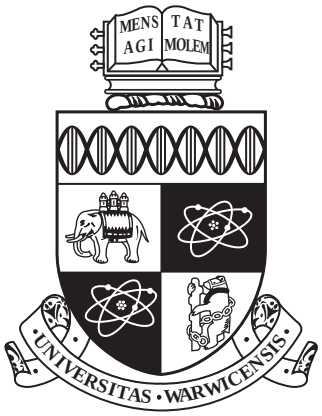


Performance Engineering Unstructured Mesh, Geometric Multigrid Codes

by

Richard Arthur Bunt

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

September 2016

Contents

Acknowledgements

Throughout the duration of my Ph.D., I have received assistance, advice and encouragement from many people. In this section I would like provide my warmest thanks to the most notable of these individuals.

To begin, I would like to thank my supervisor, Prof. Stephen Jarvis for his efforts in setting this project up and most of all for his invaluable advice. Additionally, I would like to thank him for his appearances prior to this Ph.D. – most notably for his energetic delivery of lectures and for supervising my third year project during my undergraduate course.

I would like to thank all the members (past, present and honorary) of the High Performance and Scientific Computing Group at the University of Warwick: Dr. David Beckingsale, Dr. Robert Bird, Dr. Adam Chester, James Davis, James Dickson, Dr. Simon Hammond, Dr. Matthew Leeke, Tim Law, Dr. John Pennycook, Stephen Roberts, Dr. Phillip Taylor and Dr. Steven Wright for their comments and suggestions on the research presented in this thesis. I would like to reiterate my thanks to Dr. Steven Wright for all his hard work managing the High Performance and Scientific Computing Group.

Without the Herculean effort of the administrative staff at the Computer Science department, it would not run nearly as smoothly. Therefore I must offer my thanks to Dr. Roger Packwood (specifically for, but by no means limited to loaning out an external drive bay for three years), Dr. Christine Leigh, Catherine Pillet, Lynn McLean, Ruth Cooper, Gillian Reeves-Brown and Jane Clarke.

The work in this thesis was conducted in collaboration with Rolls-Royce, so I would like to take this opportunity to thank the brilliant individuals that I encountered during various conference calls and visits to Derby. I would like to offer special thanks to Dr. Yoon Ho and Matthew Street for their questions directed at my monthly progress presentations and suggestions for further work.

The experiments presented in this thesis were conducted on a variety of computers in the United Kingdom and here I would like to acknowledge their use:

- Access to Minerva and Tinis was provided by the Centre for Scientific Computing at the University of Warwick.
- This work used the ARCHER UK National Supercomputing Service (`http://www.archer.ac.uk`).
- We acknowledge use of Hartree Centre resources in this work. The STFC Hartree Centre is a research collaboratory in association with IBM providing High Performance Computing platforms funded by the UK's investment in e-Infrastructure. The Centre aims to develop and demonstrate next generation software, optimised to take advantage of the move towards exa-scale computing.

Finally, I would like to offer my greatest thanks to Nick, Elaine, Stephanie and my sister Rachel for their support throughout this Ph.D., most notably for forcing me to take a break, by either taking me somewhere without a laptop or keeping me too busy.

Declarations

This thesis is submitted to the University of Warwick in support of my application for the degree of Doctor of Philosophy. It has been composed by myself and has not been submitted in any previous application for any degree. The work presented (including data generated and data analysis) was carried out by the author except in the cases outlined below:

- The performance model in Chapter 4 was in part formulated and described by Dr. John Pennycook.

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

Chapter 4 R. A. Bunt, S. J. Pennycook, S. A. Jarvis, L. Lapworth, and Y. K. Ho. Model-Led Optimisation of a Geometric Multigrid Application. In *Proceedings of the 15th High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing 2013 (HPCC&EUC'13)*, pages 742–753, Zhang Jia Jie, China, November 2013. IEEE Computer Society, Los Alamitos, CA [23]

Chapter 5 R. A. Bunt, S. A. Wright, S. A. Jarvis, M. Street, and Y. K. Ho. Predictive Evaluation of Partitioning Algorithms Through Runtime Modelling. In *In Proceedings of High Performance Computing, Data, and Analytics (HiPC'16)*, pages 1–11, Hyderabad, India, December 2016. IEEE Computer Society, Los Alamitos, CA [24]

Sponsorship and Grants

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

- The University of Warwick Postgraduate Research Scholarship (2012-2015).
- The Royal Society Industry Fellowship Scheme (IF090020/AM).
- Bull/Warwick Premier Partnership (2012-2014).
- Sponsored Research Agreement between University of Warwick and Rolls-Royce plc (2014).

Abstract

High Performance Computing (HPC) is a vital tool for scientific simulations; it allows the recreation of conditions which are too expensive to produce in situ or over too vast a time scale. However, in order to achieve the increasing levels of performance demanded by these applications, the architecture of computers has shifted several times since the 1970s. The challenge of engineering applications to leverage the performance which comes with past and future shifts is an on-going challenge. This work focuses on solving this challenge for unstructured mesh, geometric multigrid applications through three existing performance engineering methodologies: instrumentation, performance modelling and mini-applications.

First, an auto instrumentation tool is developed which enables the collection of performance data over several versions of a code base, with only a single definition of the data to collect. This information allows the comparison of prospective optimisations (e.g. reduced synchronisation), and an assessment of competing hardware (e.g. Intel Haswell/Ivybridge).

Second, this work details the development and use of a runtime performance model of unstructured mesh, geometric multigrid behaviour. The power of the model is demonstrated by i) exposing a synchronisation issue which degrades total application runtime by 1.41× on machines which have poor support for overlapping communication with computation; and, ii) accurately predicting the negative impact of the geometric partitioning algorithm on executions using 512 partitions.

Third, a mini-application is developed to provide a vehicle for optimising and porting activities, where it would be prohibitively time consuming to use a large, legacy application. The use of the mini-application is demonstrated by examining the impact of Intel Haswell's fused multiply and advanced vector

extension instructions on performance. It is found that significant code modifications would be required to benefit from these instructions, but the architecture shows promise from an energy perspective.

Abbreviations

AOA Angle of Attack.

API Application Programmer Interface.

BSP Bulk Synchronous Parallel.

CFD Computational Fluid Dynamics.

CPU Central Processing Unit.

DoE Department of Energy.

FLOP/s Floating Point Operations per Second.

FMA Fused Multiply-Add.

GPU Graphics Processing Unit.

HDF5 Hierarchical Data Format 5.

HPC High Performance Computing.

I/O Input/Output.

ILP Instruction Level Parallelism.

IMB Intel MPI Benchmark.

MLP Memory Level Parallelism.

MPI Message Passing Interface.

NUMA Non-uniform Memory Access.

OPlus Oxford Parallel Library for Unstructured Solvers.

PAL Performance Architecture Laboratory.

PAPI Performance Application Programming Interface.

PPN Processors Per Node.

PRAM Parallel Random Access Machine.

RIB Recursive Inertial Bisection.

SAXPY Single-Precision AX Plus Y.

SMP Symmetric Multiprocessing.

SMT Simultaneous Multithreading.

SST Structual Simulation Toolkit.

UMA Uniform Memory Access.

List of Figures

List of Tables

Listings

CHAPTER 1

Introduction

Simulation is an important cornerstone of scientific experimentation and has found widespread use in a variety of domains including engineering, physics and economics. This is because simulations can be constructed to represent environments that would be prohibitively expensive, impossible or simply against the law to physically create. Additionally, simulations can allow the measurement of variables which would be challenging to record in practice. Furthermore, simulation techniques are able to emulate processes faster than they occur in realtime, allowing the outcome of these processes to be predicted ahead of time. While some scientific simulations can be run on a computer which might be found under an office desk, achieving the level of detail required for state of the art applications in a time frame where the results are still of value, typically requires execution on large parallel computers.

These parallel computers are housed not under desks, but in dedicated facilities with their own cooling and power systems, and the most powerful are several orders of magnitude more performant than any desktop computer. Floating Point Operations per Second (FLOP/s) are one standard metric for quantifying the performance of parallel computers – this metric indicates how many calculations (e.g. addition, multiplication) can be performed on floating point numbers per second. The TOP500 list has been tracking the computing power available on the fastest 500 computers from the last decade; from this it can been seen that the computational power of machines has been increasing exponentially since recording started in 1994, to the point where the most performant of these machines can perform 93 petaFLOP/s (10^{15} FLOP/s) [118]. This regular increase in computational power has enabled scientific simulations to run at both

greater capacity and capability.

The scientific simulations which run on these machines are the product of years of research and are invaluable for their respective use cases, therefore keeping them functioning on modern hardware is of paramount importance. However, while hardware has progressed rapidly over the last four decades, these simulations have largely remained unchanged and are often still written using the programming languages around at the time of their inception. This, along with their size makes adding new features, moving the applications to upcoming parallel architectures, and running with larger degrees of parallelism a significant time investment. It is therefore necessary to further the development of performance engineering techniques, such as performance modelling (models which allow reasoning about performance) and mini-applications (distilled versions of larger codes which are more tractable to experiment with) to aid with overcoming these challenges.

This thesis represents over three years of performance engineering work on HYDRA, a geometric multigrid, unstructured mesh code developed by Rolls-Royce. HYDRA exemplifies the aforementioned challenges; it has existed through over four decades of advances in parallel hardware and supporting tool chains; it is written in FORTRAN77; and, it is tens of thousands of lines of code in size. This work sees the construction of a runtime performance model of HYDRA, which has primarily been used to i) validate HYDRA's performance on several machines at supercomputing centres around the United Kingdom, and ii) to assess the continued suitability of competing partitioning algorithms. Additionally this work presents the development of a mini-application; a smaller code capturing only HYDRA's primary computational behaviours. This is used to assess the impact of the hardware features available from the next generation of Intel Central Processing Units (CPUs) on codes such as HYDRA.

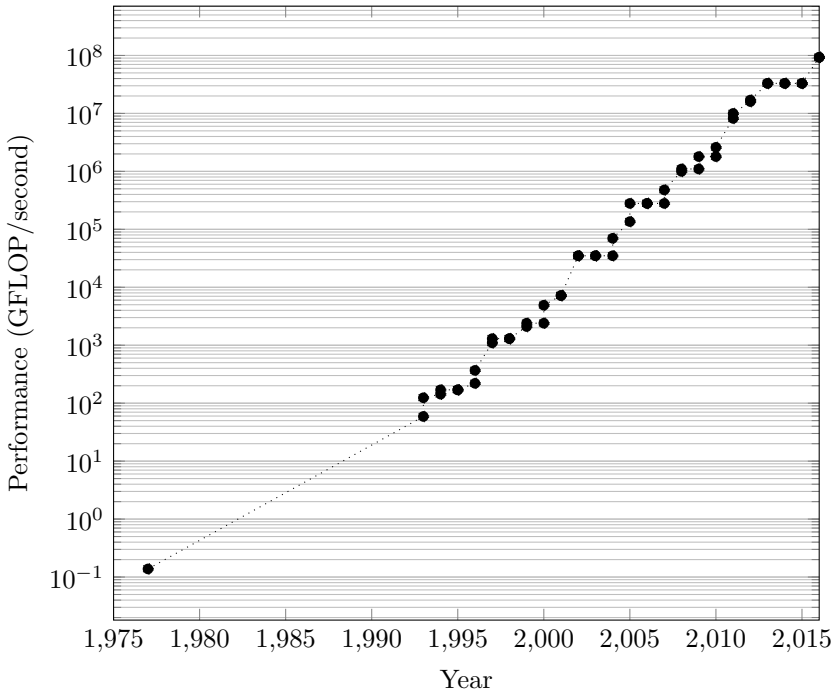


Figure 1.1: Visualisation of top supercomputer performance over time (uses data from [118, 145])

1.1 Motivation

Since the introduction of the TOP500 list [118] in 1994, the performance of the world's fastest computers has been increasing exponentially, from just under 100 gigaFLOP/s (10^9) to almost 100 petaFLOP/s in 2016 (see 1994 to 2016 in Figure 1.1). Up until 2006, this increase in performance was primarily enabled by increases in transistor densities, which was predicted to double every 18-24 months [120]. These performance increases were courtesy of Dennard scaling: as the size of a transistor is reduced, power consumption is also reduced meaning that a chip with double the density of transistors, can operate using the same power as one with half the density (but the same area) [43]. Additionally, shrinking transistors also reduces the switching delay which results in higher clock speeds [51]. What this meant for the custodians of scientific simulations, is that a reduction in time to solution could be expected every time new hardware was procured with little or no effort from them.

However, post 2006 Dennard scaling began to breakdown due to physical limits being reached (e.g. size of the transistor). This caused CPU designers to shift their attention to optimising the execution architecture to maximize the amount of instructions per cycle. This has resulted in a transition to using CPUs with multiple execution cores, which allows the CPU to run at a lower clock speed while being able to process the same number of instructions. In addition to this, the execution cores have become more complex with many more opportunities for exploiting parallelism such as wide vector units and Simultaneous Multithreading (SMT). Further to this, the recent trend has been to use many-core architectures, which trade a few complex cores for many simple cores. Unfortunately, performance increases from these architectural changes are not free; therefore application developers must actively invest time performance engineering their application to take advantage of them [146].

These, and future architectural shifts, coupled with the fact that many scientific simulations were originally written for the vector machines of the 1970s (see 1977 in Figure 1.1), has made it increasingly difficult to continuously evaluate and prepare a code for performant future use. Many High Performance Computing (HPC) centres are therefore turning to performance engineering tools and methodologies, such as predictive performance modelling and mini-applications to facilitate system evaluation, to aid in the comparison of multiple candidate machines, to investigate optimisation strategies, and to act as a vehicle for porting codes to novel architectures. This thesis focuses on the application of instrumentation, the extension of runtime performance modelling techniques and mini-applications (a relatively new form of performance engineering) to support the shift of large, legacy, unstructured mesh, geometric multigrid applications to new machines.

1.2 Problem Statement

This thesis builds upon both the performance engineering techniques of runtime performance modelling and mini-applications in the context of unstructured mesh, geometric multigrid codes; applications which represent their input geometries at varying resolution. An unstructured mesh can be represented as a graph ($G = (V, E)$), where V is a set of nodes and $E = \{(a \in V, b \in V)\}$ is a set of edges, which are arbitrary pairs of nodes. The class of simulation which is examined by this thesis, performs computation by iterating over E and accessing the nodes a and b as defined by the pair. Further to this, geometric multigrid applications operate on a list of these graphs $M = [G_0, ..., G_n]$, where the results of computation over the edges of one graph of propagated to the others. In this thesis the aim is to overcome some of the challenges associated with performance engineering this class of code.

The a and b of any edge in any G are accessed indirectly (i.e. through the edges) and are unlikely to be contiguous in memory; this posses a challenge to computers which rely on this to achieve consistent performance. This uncertainly in performance is a barrier predicting the runtime of computations over edges, as the processing time of each edge can vary. This thesis aims to quantify the impact of this uncertainty on model accuracy and propose a solution.

Unstructured meshes pose an additional challenge to modelling when attempting predict runtime at large scale. This is because selecting an optimal partitioning of a graph is an NP-Hard problem, and the generation of these partitions must be performed by a partitioning algorithm to approximate the solution [20]. This means that in order to obtain a representative partitioning for a particular scale of execution, a partitioner must first be run at that scale. This severely limits the predictive power of any performance model of unstructured mesh codes. This thesis aims to develop methods to gather representative partitioning data which can be used by performance models of unstructured mesh applications.

The development and validation of mini-applications is a challenging process, as it is not well defined in the literature and differs depending on the intended use case of the application. This thesis contributes to this gap, by detailing the experience of developing, validating and using a geometric multigrid, unstructured mesh mini-application.

The research presented in this thesis is performed as part of a project to support the continued development of HYDRA, a geometric multigrid, unstructured mesh code developed by Rolls-Royce. This code has been developed over the last four decades and as such it contains a vast amount of company knowledge. This thesis demonstrates the impact of the developed tools, performance model and mini-application by aiding Rolls-Royce with the continuing challenge of moving HYDRA onto machines with more modern architectures and larger degrees of parallelism.

1.3 Contributions

Specifically, this thesis and its products make the following contributions:

- The development of an automatic instrumentation process is presented which allows for unprecedented flexibility in terms of implementation language, the language to be instrumented and the instrumentation to be inserted. This process is implemented and then demonstrated by applying a common set of instrumentation to three different variants of HYDRA: two being chronological releases and the third being optimised to reduce synchronisation costs. The data from which is then used to compare historical changes in performance and to quantify the impact of the optimisation. In work that is not presented in this thesis, the auto-instrumentation tool has been used in support of collecting performance data from HYDRA on a variety of platforms, such as Intel Haswell/Ivybridge, IBM Power 8, BlueGene/Q so that their potential may be compared. It is expected that this tool will be applied to other codes at Rolls-Royce, such as PRECISE,

in the future;

- The development of a runtime performance model of an unstructured mesh code is detailed, which is able to capture the expected scaling behaviour of HYDRA, and its proprietary communications library, the Oxford Parallel Library for Unstructured Solvers (OPlus), when running on a grid with one level. Through the introduction of a small number of additional model terms, this model is generalised to multigrid simulations. The simple relationship between these two models significantly reduces the complexity of benchmarking a new platform, as it enables the extrapolation of complete production runtimes using data collected from the execution of small single-level datasets. The use of the performance model is demonstrated by identifying a synchronisation issue which degrades performance by up to 29.22% on machine configurations that exhibit poor support for overlapping computation with communication. An optimisation is then applied which decreases the cost of communication and synchronisation by 3.01× and total runtime by up to 1.41×. That it is possible to accelerate HYDRA to such a degree demonstrates both the accuracy of the model and the importance of reassessing whether an application's original design assumptions still hold on new hardware/software configurations. Given this accuracy, the model has been put into use by Rolls-Royce to confirm the correct strong scaling behaviour of HYDRA on small-scale evaluation hardware;

- The analytical runtime model for multigrid applications is further generalised to support multiple cycle types (e.g. V-Cycle, W-Cycle) and a variable number of time steps per iteration. Additional details are incorporated in to the performance model: buffer pack/unpack costs, runtime costs from all 300+ loops in the code base, and performance information for different memory access patterns. These additional details are validated on up to 1,024 cores of a Haswell-based cluster, using both a

geometric partitioning algorithm and ParMETIS to partition the NASA Rotor37 input deck, with a maximum absolute error of 12.63% and 11.55% respectively. Additionally, the performance model's accuracy is reported on 1,008 cores of an Ivybridge-based cluster (ARCHER). These additions to the model allow a wider range of workloads to be successfully modelled;

- Moses, an unstructured domain decomposition mini-application, is developed which is able to convert partitioning data from multiple algorithms/libraries (ParMETIS, METIS, Scotch) at varying scale (up to 100,000 partitions) for use with the runtime performance model. This data allows predictions to be made at scale without first running HYDRA at scale to collect set sizes. Runtime predictions made using this data have an error in runtime of at most 7.31% over 512 processes, when compared against predictions made with empirically collected partitioning data. The use of Moses is demonstrated in conjunction with the runtime performance model to predictively compare the relative effect on HYDRA's runtime of using Scotch, ParMETIS, METIS and a geometric partitioning algorithm on up to 30,000 cores. Using the runtime model in conjunction with Moses, predicts that the geometric partitioning algorithm will cause reduced performance in HYDRA at 512 processes when compared to ParMETIS;

- Finally, a mini-application is introduced which operates on datasets with the following properties: i) unstructured mesh, ii) geometric multigrid, and iii) a variable number of neighbours per node. It is validated using two previously developed techniques which have not previously been applied to this class of code. These techniques provide evidence of the similarity between the mini-application and the parent code in terms of their shared memory scalability. To conclude, the use of the mini-application is demonstrated by quantifying the impact of the new hardware features introduced to the Intel Haswell platform over Intel Ivybridge for geometric multigrid, unstructured mesh applications. It is found that FM-* instruc-

tions and the AVX2 features have a limited impact on performance, but there is potential for Intel Haswell to deliver application results at a much lower total energy. At the time of writing this thesis, this mini-application is being used as part of procurement exercises, and it is hoped that this mini-application will be used in the future as a vehicle for optimisation, porting and execution on machines where HYDRA is not cleared for use.

1.4 Thesis Overview

The remainder of this thesis is structured as follows:

Chapter 2 contains knowledge of parallel computing essential to understanding the field: the lexical set, the opportunities for parallelism provided by the hardware, and the software which enables the usages of this hardware. This chapter continues by providing historical background to performance engineering techniques (the primary focus of this research) up to state of the art techniques and their relative merits and shortcomings.

Chapter 3 introduces the context for this research – unstructured mesh, geometric multigrid applications, specifically Rolls-Royce HYDRA, an instance of this class of code. The features and structure, and their associated performance challenges, of these codes and those specific to HYDRA are described. This introduction also details the challenges associated with collecting data from HYDRA and other unstructured mesh applications and how these were overcome using newly developed instrumentation tools. Finally this chapter presents performance information collected using these instrumentation tools, which serves as an overview of HYDRA's performance and a demonstration of the instrumentation tool's flexibility.

Chapter 4 presents the initial development of a runtime performance model of HYDRA, first for single level execution and then for multigrid execution. Validation results are presented for two representative datasets over 384

cores. The value of this model is then demonstrated by identifying detrimental communication behaviours through a discrepancy in the predicted (expected) and actual contribution to total runtime from synchronisation costs.

Chapter 5 generalises the runtime performance model to support a variable number of time steps per iteration, arbitrary multigrid cycle types and all datasets which rely solely on loops. This chapter then identifies the challenge of generating representative partitioning data for use in large scale runtime predictions, to which the construction of a partitioning mini-application is presented as a solution. The use of this mini-application is demonstrated by performing a predictive evaluation of the effect various partitioning libraries have on application runtime.

Chapter 6 details the construction, validation and use of mini-HYDRA, a mini-application designed to compliment the performance model developed in previous chapters. The mini-application is used to quantify the impact the new instructions introduced as part of Intel Haswell (Fused Multiply-Add (FMA) and AVX2) on HYDRA's inviscid flux kernel.

Chapter 7 concludes this thesis with a summary of the work carried out and a discussion of the impact this research has had on the individuals involved in this project. Additionally the limitations of this work and possible future research directions to either rectify these limitations or broaden the research are identified.

CHAPTER 2

Parallel Computing and Performance Engineering

Parallel computing was born from the need to run algorithms of i) increasing complexity to completion in a time frame such that the result is still of value, and ii) over increasingly larger input domains. As the name suggests, parallel computing enables this by running the independent computational parts of an algorithm simultaneously (in parallel), as apposed to performing calculations one by one (serially). One classical example of this involved human computers, rather than the electronic computers available today, who performed engineering calculations at the Langley Memorial Aeronautical Laboratory in the 1940s [31]. Groups of human computers existed at this laboratory in order to field these calculations in parallel, and in some cases if a group reached capacity, the calculations were sent to a secondary group.

The purpose of this chapter is to present the foundation of knowledge which the work in this thesis builds upon. Not only does this foundation give the thesis context, it also details the lexicon and concepts which are used throughout. Specifically, this foundation consists of i) a description of the physical hardware and software which comprise the domain of High Performance Computing (HPC); ii) the laws and abstract models which capture the behaviour of HPC applications; and, iii) an introduction to modelling and benchmarking the performance of HPC applications and the relative merits of these techniques.

2.1 The Composition of a Parallel Machine

Today's parallel computers often consist of electrical circuits, fibre interconnects and large cooling apparatus. These electrical computers allow many opportunities for the parallel execution of calculations, and often multiple opportunities

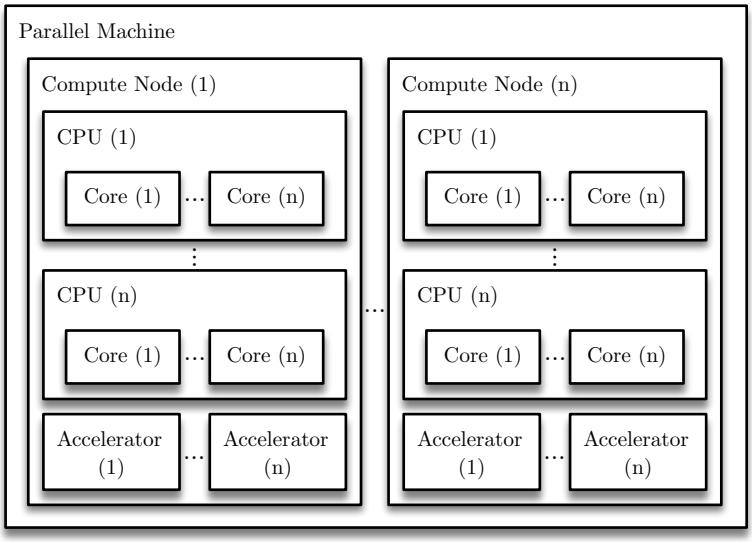


Figure 2.1: An abstract representation of a parallel machine configuration

will be exploited in order to achieve maximum benefit [144]. Opportunities for parallelism are exposed throughout all the hardware components of a parallel machine (see Figure 2.1) from the bit-level in a single Central Processing Unit (CPU) core, to the data-level where datasets are partitioned across groups of compute nodes.

2.1.1 Core

Bit-level parallelism allows the CPU core (see Figure 2.1) to push more bits around per clock cycle, typically by increasing the size of the registers. Historically, this increase in bits has moved from 4 up to 64 bits [38]. As an example of how this change can improve performance consider a CPU core which operates on 32-bit registers and needs to perform an operation on 64-bit numbers – it must perform the calculation in two parts, thereby taking two clock cycles. However, a CPU capable of operating on 64-bit registers is able to complete the same calculation in a single clock cycle.

At a slightly higher level of parallelism in the CPU core, there is Instruction Level Parallelism (ILP) [139]. This is an umbrella term for all mechanisms

which allow data independent instructions to progress through the CPU core in parallel. ILP is achieved through a variety of mechanisms in order to maximise the use of the CPU core under a variety of workloads. The first ILP mechanism to mention is pipelining; this involves separating the handling of instructions into distinct stages, with separate hardware for each [148]. The most basic pipeline has separate fetch, decode and execute units. This decomposition allows the decoding of an instruction to occur in parallel with the fetch of the next instruction.

Further parallelisation can occur at each of the stages in the pipeline by duplicating and providing specialised execution units – this is known as a superscalar architecture [148]. An example of this is the Intel Haswell CPU core which, has two execution units capable of performing an Fused Multiply-Add (FMA) instruction. In order to maximise the use of these additional execution units, techniques are employed such as dynamic [87] and speculative execution [148]. The combination of these techniques allows the CPU core to begin two FMA operations simultaneously [103] and would benefit the computation defined by Equation 2.1 as the partial results a and e have no data dependencies and can therefore be performed simultaneously.

$$\begin{aligned} a &= b * c + d; \\ e &= f * g + h; \\ r &= a + e; \end{aligned} \tag{2.1}$$

In the case of the decoder, not only can they fetch and decode multiple instructions simultaneously, they can also employ macro-fusion to exploit parallelism at the micro-operation level (the sub-operations which form instructions). This is achieved by fusing the micro-operations from different instructions into a single, but more complex micro-operation [62]. Given a decoding unit which can decode three instructions and one macro-fused instruction, the application

of macro-fusion can increase the number of decodes from four to five per cycle, thereby reducing the pressure on this unit.

Another opportunity for parallelism in the CPU core is provided by the presence of vector units. These enable data level parallelism by handling dispatched vector instructions that apply the same operation to several data items in parallel [38]. One such example of a vector instruction is *vaddpd*, which adds two sets of 64-bit doubles [90]. The number of operations these instructions perform in parallel depends on the width of the registers they use. While in the past, vector lengths were large [145], the trend at the time of writing this thesis is for CPU cores to have vector lengths of four or eight 64-bit doubles [89]. In addition to enabling arithmetic in parallel, vector instructions exist to push data, such as the `vmovapd` instruction, which moves packed and aligned 64-bit floating point values [90].

Parallelism is not limited to arithmetic operations; there are also opportunities in the CPU core for Memory Level Parallelism (MLP) [148]. This mechanism allows multiple memory operations to occur simultaneously or overlap. CPU cores with MLP often exploit this by prefetching data elements from slower to faster access memory before the data is required by an instruction, in the hope it will decrease cache misses and increase achieved memory bandwidth. MLP is important as it helps maximize the use of the available memory bandwidth, which can be a bottleneck for performance [165, 167]. Essentially, as a higher percentage of an application's arithmetic is parallelised, the more memory bandwidth is required to feed the calculations until this becomes the bottleneck.

2.1.2 CPU

At the next level of the hardware hierarchy (Figure 2.1) there is Symmetric Multiprocessing (SMP), a configuration consisting of multiple CPU cores. This configuration is capable of exposing thread and task level parallelism [131] by hosting multiple threads of execution, each of which exposes the parallelism

detailed in Section 2.1.1. At the most basic level, a CPU core offers Simultaneous Multithreading (SMT) (multiple threads of execution per core) to increase the use of a superscalar architecture, by increasing the sources of instructions [131] e.g. Intel Hyperthreading [153]. At the other end of the spectrum there are SMP systems with multiple CPUs, each with multiple cores, which can in turn support multiple threads of execution. At this point, it becomes important to consider the arrangement of these components as this impacts the efficiency of communicating data between them.

There are two classes of SMP system: Non-uniform Memory Access (NUMA) and Uniform Memory Access (UMA). In an UMA system, CPU cores can access their local shared memory at the same latency and bandwidth as the shared memory located on other cores. In a NUMA system CPU cores can access their local memory faster than the local memory of other CPU cores [131]. The configuration shown in Figure 2.1 gives rise to two CPU NUMA regions: communication between cores on the same CPU and between cores on different CPUs. Due to the difference in communication time between NUMA regions, locating data close to the CPU core which is operating on it can make a huge difference to performance [15, 168]. Typically parallel machines use NUMA due to its favourable memory size scaling over UMA [131].

2.1.3 Accelerators

The inclusion of accelerator cards in parallel machines has become increasingly popular in recent years and in fact two of current top three in the TOP500 list (Tianhe-2 and Titan) feature them (using Intel Xeon Phis and Nvidia Teslas respectively) [118]. This increase in popularity is due to the potential of accelerator cards to deliver power efficient computations [133], in an attempt to overcome one of the challenges in HPC as outlined by the Department of Energy (DoE) [76].

The basic concept behind the Graphics Processing Unit (GPU) and the Intel Xeon Phi is to provide many simple processing elements to expose a large

amount of fine-grained parallelism to computing applications. This is in contrast to a CPU which contains fewer general purpose cores. In the case of the Intel Xeon Phi (Knights Corner) it has 61 cores, whereas an equivalent CPU, the Intel Xeon E5-2670 has 8 [75]. The trade-off is that the cores on the Intel Xeon Phi are simpler (e.g. they use in-order execution) and are lower clocked compared to their CPU counterparts. The strength of the simpler cores is that they have a larger degree of SMT, longer vector registers, they are greater in number and have increased memory bandwidth [75]. The potential for individual applications to benefit from this configuration is usually non-obvious and must be assessed by way of an in-depth performance study on a per accelerator basis.

2.1.4 Compute Node

At the final level of the hardware hierarchy (see Figure 2.1) there are multiple connected physical machines, each containing multiple CPUs thus further scaling the capacity for thread and task level parallelism. This configuration is known as a distributed memory system [131].

As with the CPU level of the hierarchy, it is important to consider the arrangement of the components (compute nodes) and additionally the medium by which they are connected as this impacts performance [131]. Another consideration is the method by which a problem is partitioned between the distributed memory as this influences factors such as idle time due to load imbalance and time spent communicating data.

2.2 Parallel Data Decompositions

Many HPC applications simulate physical processes such as heat transfer and fluid flow; examples of such applications include LAMMPS [137], a molecular dynamics simulation, and OpenFOAM [92], a fluid dynamics simulation. Simulations usually represent their domains using either a structured or unstructured

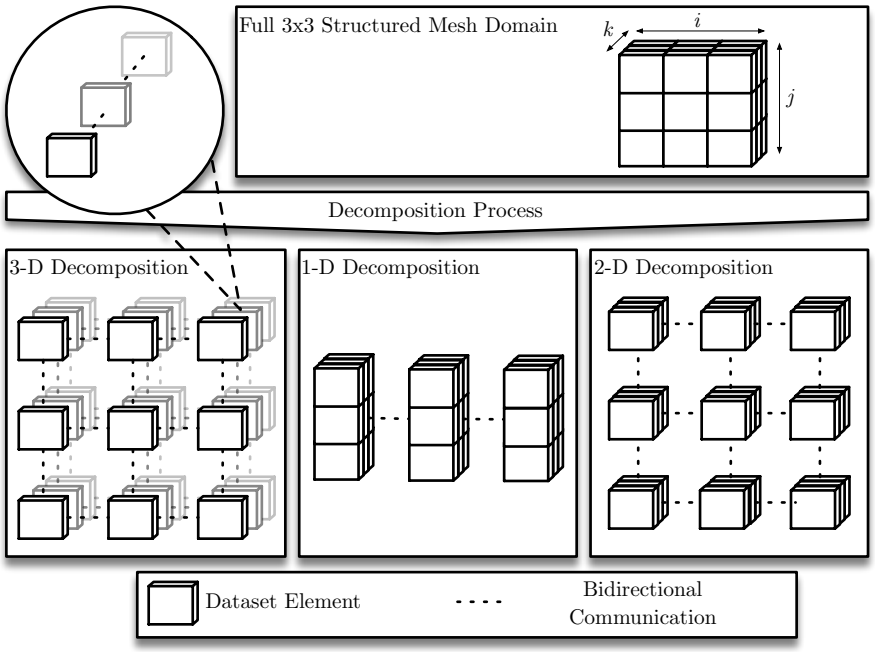


Figure 2.2: 1-, 2- and 3-D structured mesh decompositions

mesh which must be partitioned between compute nodes prior to execution on a distributed memory machine (Section 2.1.4). The process used for partitioning can impact performance because it influences both the ratio of computation to communication and the quality of the load balancing. Furthermore, the selection of partitioning process depends on whether it is to be used on a structured or unstructured mesh.

2.2.1 Structured Mesh

A structured mesh is a regular arrangement of points with a topology that is apparent from their position in space. An example demonstrating this is given in the top pane of Figure 2.2, where the quantities of the simulation are represented as averages over each sub-cuboid.

Decomposing a structured mesh is achieved using either a 1-, 2- or 3-D decomposition, as visualised by the lower set of three panes in Figure 2.2. The height, width and depth (i, j, k) are divided by some function of the total num-

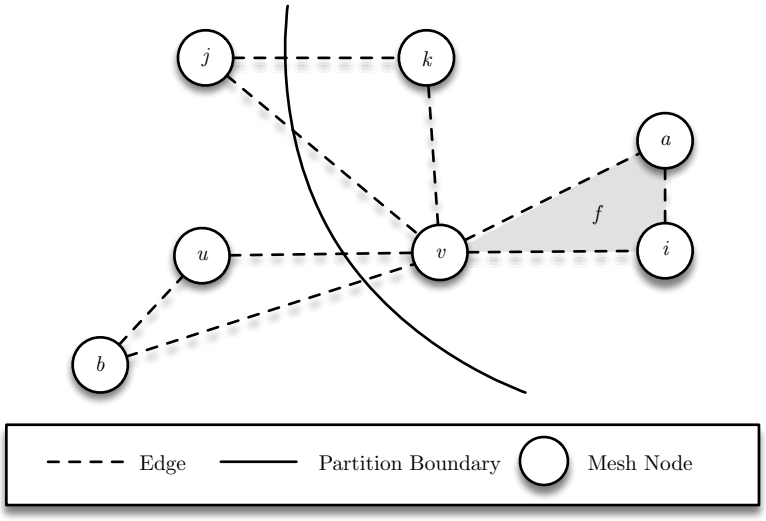


Figure 2.3: Partitioning an unstructured mesh

ber of threads available. Each of these decompositions achieves a perfect load balance, although in practice this only occurs when the number of processors is an exact divider of one of the domains sides (i, j or k), a square number or a cubic number, in the case of 1-, 2- and 3-D decompositions respectively. However, the ratio of communication to computation does change depending on the chosen decomposition, with a 3-D decomposition theoretically being the most favourable [105].

2.2.2 Unstructured Mesh

An unstructured mesh (see Figure 2.3) is a collection of nodes (e.g. j and u), edges (e.g. between u and v) and faces (f), with the nodes being at an arbitrary position in space. The flexibility of the unstructured mesh allows complex geometries to be represented and the simulation resolution can be increased (and also the computation time) only in regions of importance by altering the density of mesh points [114].

The neighbours of a node in an unstructured mesh are not implicitly defined, as is the case for a structured mesh code where the neighbours can be determined

using offsets to the array indices. This means that an explicit list of neighbours must be maintained, so that when computation over nodes is performed (e.g. the accumulation of fluxes) data can be read from the required locations. This has implications for the spacial and temporal locality of accesses when gathering unstructured mesh features from memory, as a mesh node's neighbours can appear at arbitrary strides either side of its memory location.

Unlike a structured mesh, which can often be divided using algebra, an unstructured mesh must be load balanced between distributed memory locations using specialised algorithms which are the result of numerous bodies of research [63, 97, 128, 152, 164]. These algorithms are necessary to partition unstructured meshes because it is not immediately clear how the mesh should be divided, due to the arbitrary position and the number of neighbours of its nodes. For example in Figure 2.3, mesh nodes u and b are in partition zero and mesh nodes v, a and i are in the first partition, with a total of two edges cut on level zero. However, some other arbitrary line could be drawn to partition the mesh nodes.

Ideally both the number of edges cut and the load imbalance between the number of nodes in each partition should be minimised, as a high value of the former means increased communication time, and an high value of the latter results in increased processor idle time. Finding an optimal partition is NP-Hard and therefore these algorithms serve to find approximations to the optimal solution [20].

2.3 Parallel Programming Laws and Models

When running and developing HPC applications it is only natural to want to quantify their performance (e.g. time to results, use of parallelism). This section covers the most basic performance metrics and laws which govern their behaviour.

2.3.1 Speedup

$$S(n) = \frac{R_a}{R_b} \tag{2.2}$$

Speedup ($S(n)$ in Equation 2.2) is the ratio of two performance values, R_a and R_b, where n is the variant between these values [131]. Typically R_a and R_b are application runtimes from either, two different variants of the same application, or the same variant run using different degrees of thread level parallelism. In the latter case, n would be the number of threads R_b was executed with. As an example, consider an application that can exploit thread level parallelism which has had its runtime recorded using both a single thread and four threads. Applying Equation 2.2 to these runtimes, with R_a being the single threaded runtime and R_b being the runtime when using four threads, will yield the value $S(4)$ which indicates how many times faster (or slower) the multi-threaded execution was compared to the serial run. While this is a useful summary metric, especially when comparing the effectiveness of different code optimisations (e.g. with and without using vector units) care must be taken when reporting speedup in the case where multiple code optimisations have been applied, as it becomes unclear which optimisation the speedup came from.

2.3.2 Parallel Efficiency

$$P_e(n) = \frac{S(n)}{n} \tag{2.3}$$

Parallel efficiency ($P_e(n)$ in Equation 2.3) codifies the extent to which an application utilises opportunities for parallelism as opportunities are increased (n) [131]. Equation 2.3 has speedup as one of its constituents. This captures the performance change when using different degrees of thread level parallelism. This speedup value is normalised by the degree of thread level parallelism used as this is the speedup expected from an application which fully uses the available parallelism. This metric can be used to compare the scalability of two different

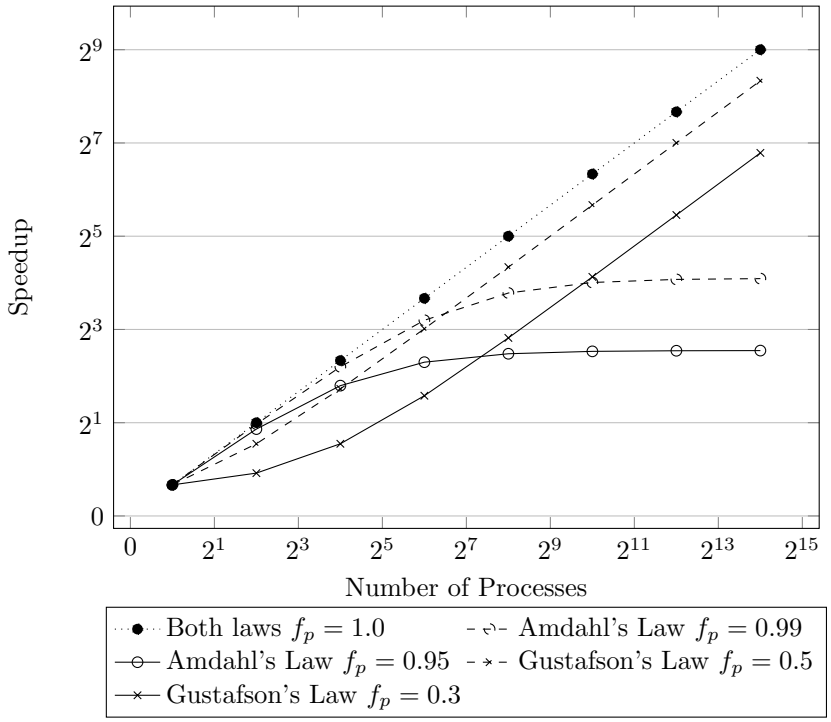


Figure 2.4: Amdahl's Law and Gustafson's Law for varying values of f_p

applications, or quantify whether the cost of additional resources required to make additional parallelism available to the application is worth the increase in performance.

2.3.3 Amdahl's Law

$$S_t(s) = \left(\frac{1}{f_s + \frac{f_p}{s}} \right), \qquad S_t(s) \leq \frac{1}{f_s} \tag{2.4}$$

While the metrics of speedup ($S(n)$) and parallel efficiency ($S_e(n)$) capture observed performance, neither attempt to bound performance, whereas Amdahl's law [6] (Equation 2.4) can provide an upper bound on speedup. It does this for an entire application run (S_t) in terms of the fraction of parallel code (f_p) and serial code (f_s), by pushing the speedup of the parallel region (s) to infinity. By priming Equation 2.4 with various fractions (f_p equal to 1.0, 0.99 and 0.95), the maximum achievable speedup for a given f_p can be compared. Figure 2.4 shows

that given an application where all of the code can be parallelised, the maximum achievable speedup is linear in the number of threads. For example, with a fixed work load a single thread must perform all the work, but two threads can perform half of all the work simultaneously; this is known as strong scaling. It should be noted that this bound on speedup is not strictly true, as super linear speedups can be observed in practice due to the degree of available parallelism not being the only factor influencing application performance; typically super linear speedups are observed as a result of the decreasing data associated with each thread fitting into successively smaller (but faster) levels of cache. Equation 2.4 further indicates that for a given fraction of parallel code, the runtime will approach the runtime of the serial region as the degree of parallelism tends to infinity. Even by having a serial region of 1% vastly impacts the overall parallel efficiency of the application.

2.3.4 Gustafson's Law

$$S_t(s) = f_s + s(1 - f_s) \tag{2.5}$$

Gustafson's law [69] considers the bound on speedup for an entire application (S_t) from a different perspective. Gustafson argued that in practice strong scaling was not the norm, and that the problem size increased with increasing degrees of parallelism (weak scaling); this behavior is codified in Equation 2.5 and plotted for varying proportions of parallel code in Figure 2.4. From this figure it can be seen that even at much lower proportions of parallel code (f_p equal to 0.3 and 0.5), the predicted bound on speedup is much more optimistic than the bound predicted by Amdahl's law for considerably higher proportions of parallel code (f_p equal to 0.95 and 0.99).

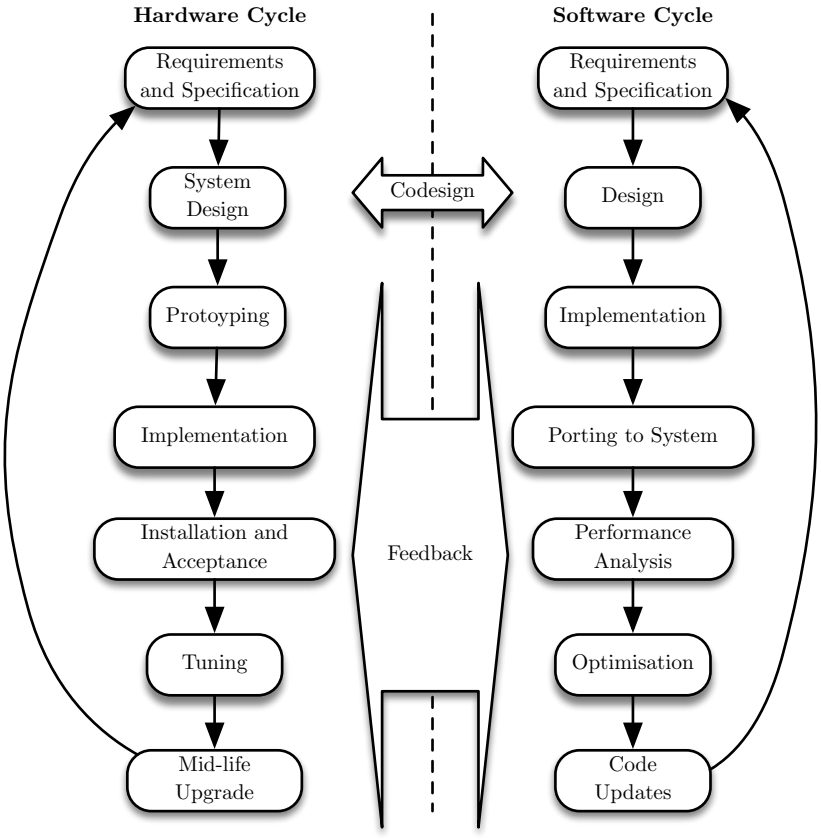


Figure 2.5: Hardware and software life cycle (adapted from [11])

2.4 Performance Engineering

HPC applications are complex pieces of software which are often tens of thousands of lines of code long, developed over several decades, and are run with hundreds of thousands of degrees of parallelism. These codes are of high utility and therefore it is important to continually assess and improve their performance on current hardware through optimisation and code updates, and also prepare for future hardware (see Figure 2.5).

The hardware upon which HPC applications are run is just as complex as the applications themselves, so it is also important to continually assess the health of this hardware. As the hardware ages, there will come a time when it

is necessary to replace it (see Figure 2.5). At this time is essential to have a tool box of methods which allow the comparison of candidate replacement hardware at the lowest level of parallelism, to indicate how well a particular application will perform on a new architecture and/or use finer grained parallelism, and can validate the performance of an application on new hardware.

Performance engineering is a field which encompasses many mature techniques (e.g. benchmarking, mini- and compact applications, profiling and analytical performance modelling) developed to support each stage of this complex cycle of developing and maintaining HPC systems and applications.

2.4.1 Profiling and Instrumentation

Profiling an application is the act of extracting performance data from a running application. Examples of data which can be collected about an application include: the time duration and frequency of code blocks (e.g. gprof [68]), cache hits and number of floating point operations (e.g. Performance Application Programming Interface (PAPI) [18]), power draw (e.g. RAPL [39]) and memory usage (e.g. Valgrind massif [129]). These metrics can be used to identify targets for optimisation and parallelisation. For example, a routine which uses memory inefficiently may demonstrate a high memory watermark or a high cache miss rate, and a routine which would benefit the most from parallelisation efforts would dominate runtime.

There are two main classes of profiling: statistical and tracing. Statistical profiling periodically samples performance values whereas tracing collects all data relating to a particular value (e.g. gprof [68]). This means that while the statistical profiler collects incomplete data, and may even overlook performance values entirely, it has the potential to exhibit lower overhead when running at scale than a tracer (e.g. Sun Studio [91]). In the case of determining the frequency and duration of code regions, a statistical profiler will periodically sample data at the position of execution and then compile these samples in to a report. In contrast, a tracer would record the entry and exit times of code

regions.

Another consideration to take into account is the method of instrumentation by which values are collected. There are three main approaches to instrumentation (as mentioned by Chittimalli and Shah [33]), these are binary, byte code and source code instrumentation. The use of byte code instrumentation is immediately ignored, as it applies to code running in a virtual machine, which does not apply to the codes in this thesis. Source code based instrumentation requires the augmentation of an applications source code, which can happen before (e.g. PAPI [18]) or during compilation (e.g. gprof [68]). Static binary instrumentation (e.g. Dyninst [160]) augments the compiled binary, naturally this technique does not require the code to be recompiled. Similarly, dynamic binary instrumentation (Intel PIN [106] and Valgrind [129]) does not require a recompilation but may introduce runtime overheads.

2.4.2 Benchmarks, Mini- and Compact-Applications

The comparison of hardware choice and software optimisation approaches is another core part of performance engineering. To enable such comparisons a vast number of benchmarks, mini- and compact applications have been developed. Benchmarks are applications which exercise and subsequently quantify the performance of a particular subsystem, or group of subsystems belonging to a parallel machine, under a defined workload. Benchmarks come in a variety of forms. At the lowest level, benchmarks can exercise and measure the performance of a single subsystem (e.g. main memory [115], cache [121] and inter-compute node interconnect [88]) and at the highest level, there are benchmarks which capture and asses the performance of common computational patterns, for example Single-Precision AX Plus Y (SAXPY) [45, 122], or even the behavior of a specific parent application (mini- and compact-applications) [110, 134, 140, 156].

There are several benchmarks which facilitate comparison of computational power between different machines or single CPUs, but the most famous has to be LINPACK [45] due to its prominent role in ordering the TOP500 list [118]. The

LINPACK benchmark solves dense systems of linear equations and is therefore Floating Point Operations per Second (FLOP/s) heavy. While this makes it an excellent compute benchmark, doubts about its usefulness have been raised as it is only representative of a single class of code. This has led to the development of further benchmarks [46, 47, 109]. Other compute benchmarks include the Livermore Loops [54] and the Thirteen Dwarfs [7].

Power is another metric by which to compare machines and individual CPUs, and is becoming an increasingly important concern. The FIRESTARTER benchmark can be used extract "near-peak power consumption" from a particular CPU [70]. However, in the case of a machine comparison, the amount of useful work done needs to be taken into consideration, meaning FLOP-per-Watt [53] is a better method of comparing two architectures in terms of power.

As with the CPU, several benchmarks exist for characterising the different levels of a memory system. For measuring the peak memory bandwidth from main memory, the STREAM benchmark can be used [115]. This delivers result in MB/s for four common memory access patterns: copy, scale, add and triad. A second benchmark exists, Cachebench, for measuring the memory bandwidth from individual levels of cache to the CPU [121]. Both of these benchmarks can be used to compare the memory bandwidth of several candidate machines and can give an indication of how well a code which is bottlenecked by memory bandwidth will perform (Section 2.1.1).

Benchmarks also exist to quantify the performance of shared and distributed memory computations. At the shared memory level there is OpenMP for which there is a benchmark to measure the overhead from using various operations such as loop iteration scheduling and synchronisation [8, 21, 22, 58]. On the distributed memory side, benchmarks such as Intel MPI Benchmark (IMB) [88] and SKaMPI [142, 143] collect bandwidth and latency figures for various message sizes. These Message Passing Interface (MPI) benchmarks can be used to compare different MPI implementations (e.g. MPICH2, Intel MPI, OpenMPI) and different interconnects (e.g. Infiniband and Cray Aires).

A more recent trend has been to develop so called mini-applications, which capture a subset of the key performance characteristics (e.g. memory access patterns) from a class of applications or a parent code. Numerous mini-applications have been developed, some of which have been released as part of projects such as the Mantevo Project [77] and the UK Mini-App Consortium [158]. Mini-applications from these repositories and other standalone mini-applications have been used in a variety of contexts: i) exploring the suitability of new architectures for molecular dynamics codes [134], ii) examining the scalability of a neutron transport application [156], iii) to aid the design an development of a domain specific active library [140] and, iv) exploring different programming models [110].

2.4.3 Modelling Parallel Computation

The formation of Amdahl's law in 1967 was a just a starting point for the development of further abstract and concrete models governing and describing the performance of both parallel hardware and applications. While Amdahl's law contains just a few parameters to represent both the software (fraction of parallel code) and the hardware (number of processors), newer models have potentially hundreds, characterising compute, communication behaviour and synchronisation behaviour. The increase in parameters allows more complex hardware and software behaviours, but brings with it a decrease in tractability. However, performance models still provide a useful platform from which to perform analyses.

Parallel Random Access Machine (PRAM)

In 1978, PRAM was proposed as one of the first models of parallel computing. It was developed due to the need for a framework with which to develop parallel algorithms while avoiding the nuances of real hardware [56]. This model has the follow constituent parts: an unbounded set of processors $P = p_0, p_1, ...$ which are each able to run `LOAD`, `STORE`, `ADD`, `SUB`, `JUMP`, `JZERO`, `READ`, `FORK` and `HALT`

instructions; an unbounded global memory and processor local memory, both capable of storing non-negative integers; a set of input register $I_0, I_1, ..., I_n$; a program counter; a per processor accumulator, also capable of storing non-negative integers; and, a finite program A comprising the aforementioned instructions. Computation then proceeds as follows:

1. The input is placed in the input registers: one bit per I_k.
2. All memory is cleared and the length of the program (A) is placed into the accumulator of P_0.
3. Instructions are then executed simultaneously by each $p \in P$.
4. The `FORK` instruction can be used to spawn computation on an idle processor.
5. Execution stops when either a `HALT` instruction is executed by P_0 or a write instruction is executed on the same global memory location simultaneously by multiple processors.

There are several deficiencies with PRAM, the first of which is the lack of cost associated with communication. This prevents the model from being an accurate representation for NUMA machines which have multiple communication layers (core-to-core, socket-to-socket and node-to-node). This lack of cost associated with communication time additionally prevents the scaling behaviour of an application from being modelled when run at large scale, where communication costs can dominate.

Bulk Synchronous Parallel (BSP) Computation Model

BSP was developed in 1990 by Leslie Valiant with the same underlying goal as PRAM [150, 161], to provide a model which allows researchers to independently develop parallel algorithms and hardware. BSP overcomes the primary flaw with PRAM: there was no cost parameter for communication events, which limited the accuracy of PRAM when costing various algorithmic and hardware

choices. BSP execution proceeds in supersteps (S), each of which consists of three sub-steps which are described below along with the relevant modelling parameters.

1. Simultaneous computation on local processors W_i, where this is the cost of computation in instruction rate per processor.

2. Data transfer between local processors $h_{s,i}g$, where g is a measure of network permeability under continuous traffic to uniformly random address locations and $h_{s,i}$ is the number of messages or some function of this and the size of the messages per processor [150].

3. Barrier synchronisation (l).

With values for the aforementioned parameters, Equation 2.6 can be used to compute the cost of an algorithm (C) expressed using BSP.

$$C = \sum_{s \in S} (\max_{i \in P}(w_{s,i}) + \max_{i \in P}(h_{s,i}g + l)) \tag{2.6}$$

From examining Equation 2.6, two observations about the performance of BSP applications can be drawn. First is that load balance is important for both computation and global communication since they are both maximums over all processors and second, minimising the number of supersteps will have a positive effect on performance as it reduces the amount of global synchronisation [150].

As reported by Skillicorn *et al* the BSP model has been used to the benefit of several application's performance. Most notably is the work by Hill *et al* where the authors develop a BSP model of a multigrid application [78]. This model is then used to predictively assess the impact of different network choices on the runtime of the code.

One of the main weaknesses of BSP is its disregard to data locality. This presents an issue for HPC applications as data locality is important to make use of hardware features such as caches and vector units, not using either of these

can lead to significant slow downs. However, the issue of locality was addressed in an extension to BSP by Tiskin [155].

The LogP Model

The PRAM and BSP models are high level abstractions of parallel computation and over simplify the detail of applications and hardware, specifically in the area of communication, which permits the development of algorithms which would perform poorly on real hardware. Culler *et al* address this issue with their LogP model [37], which (as given by its name) has the following parameters:

L An upper bound on the latency or delay incurred in sending a point-to-point communication containing a small number of words.

o The length of time a processor is engaged in message sending activities (overhead).

g The maximum interval between consecutive messages (gap). The reciprocal of g is the per-processor memory bandwidth.

P The number of processors. For simplicity, all local operations complete in unit time.

Given values for these parameters in cycles, a communication graph can be can be constructed indicating the cost of a particular communication pattern. The authors go to great lengths to differentiate LogP from PRAM and BSP by pointing out the flaws in these previous models. Most notably the authors point out that PRAM does not penalise algorithms which use an excessive amount of communication whereas LogP discourages this. While BSP also discourages a large amount of interprocess communication, it mandates that only h messages can be sent and received by processors in any superstep, where as LogP allows more fine-grained control of messages and hence the ability to represent more complex schedules of messages [37].

The LogP model has been successfully used by the authors, and by many other researchers to investigate the performance of algorithms [28, 50, 82, 86, 95]. Building on the success of this model, further extensions have been made: LogGP, which incorporates terms for long messages [5], LoPC which introduces terms for contention [57] and LogfP which handles small Infiniband messages [83].

2.4.4 Analytical Modelling

The BSP and LogP performance models form the basis of a much larger class of modelling techniques known as analytical modelling. These models are formed of mathematical equations, often representing the time to completion of some performance property (e.g. communication and computation time) of an algorithm or application. These models are typically less constrained than BSP as there is no framework other than mathematics itself to work within. This gives them the power to represent complex behaviours not possible within a tight framework.

The next major milestone in the development of analytical modelling techniques was detailed by Adve in 1993 [2, 3]. The basic idea is the same as previous models, in that the total cost (T_{cost}) of an application can be represented as the sum of its costs (as codified in Equation 2.7).

$$T_{cost} = (C_{computation} + C_{communication} - C_{overlap}) + C_{synchronisation} + C_{overhead} \tag{2.7}$$

Each of these sub-term are often backed by their own models, for example one might use a LogP (see Section 2.4.3) style model or a latency/bandwidth model [79] to calculate $C_{communication}$. The $C_{overlap}$ term is in recognition that performance improvements can be gained from overlapping communication and computation. This can alternatively be implemented by taking the maximum of computation time that can be overlapped and communication time. Additionally, terms such as $C_{computation}$ are often maximums over processors in order to

capture the critical path of an application. The $C_{synchronisation}$ term is used to represent delays due to load imbalance between different processes and global synchronisation. Finally, $C_{overhead}$ is there to represent costs such as resource contention. The use of this style of model has been documented extensively in the literature.

One such body of work carried out by the Performance Architecture Laboratory (PAL) at the Los Alamos National Laboratory details the usage of modelling techniques throughout the hardware life cycle (see Figure 2.5). The authors of this work use performance models: i) during system design to examine hybrid accelerated systems [9]; ii) during implementation to predict the performance of a large scale systems from small scale evaluation hardware [11]; iii) during procurement to compare several vendor offerings [11]; iv) during post-installation to validate machine performance [101, 136]; v) to aid in optimisation [94]; vi) to support machine maintenance [11]; vii) during upgrades [41, 99]; and, many other pieces of work [10, 85, 100]. This body of work exemplifies the use of not just analytical modelling but performance modelling in general.

Another body of analytical modelling work was carried out by Gahvari *et al*, in which the authors apply a model to an algebraic multigrid application executing on a range of architectures available at the Lawrence Livermore National Laboratory (including a Blue Gene/P and a Blue Gene/Q) [59, 60, 61]. This work demonstrates the usefulness of using modelling to understand the scalability of algebraic multigrid applications and the use of hybrid OpenMP/MPI programming.

Work from the University of Warwick also specialises in the creation of analytical performance models. Their work has produced models for many classes of MPI based codes which have been applied to several of the use cases defined by PAL [13, 14, 40, 123, 124, 135].

The primary disadvantage of analytical modelling is the time they take to construct; however due to their analytical nature these models are fast to evaluate and are therefore highly tractable. This, along with their utility, more than

offsets the time investment required to development them.

2.4.5 Simulation

In contrast to analytical models, which consist of equations representing performance costs, simulations are virtual environments (often created in software) which emulate the execution of an application. Simulations are employed for many of the same reasons as analytical models: to give guidances as to an applications performance on existing and future hardware. There exists two main methods for performing a simulation:

Trace Driven Simulators replay traces collected from an execution of an application. Examples of such simulators include the SimpleScalar [25] tools set which can execute instruction level traces, MemSpy [112] which operates on traces of memory accesses, and PSINS [154] an MPI trace executor. Trace execution presents several difficulties relating both to the size of traces and extrapolating traces of runs at large scale from smaller scale.

Execution Simulations are driven by a representation of the application to be simulated written in a simulation language. Examples of such simulators include PACE [130], its successor WARPP [72], LogGOPSim [84] (an MPI execution simulator) and the Structual Simulation Toolkit (SST) [159]. Representing the application in a simulation language has several benefits such as the ability to capture control flow and, in the case of WARPP, this representation is much smaller than a trace; as timing information is accumulated rather than replaying each individual instruction.

Simulations have several advantages over analytical models, in that they can represent performance features that are difficult to represent analytically, such as non-determinism and control flow. However, the ability to represent these features comes with a price; increased simulation time. Often large amounts of computing resource is needed to run these simulations which may not always be readily accessible.

2.5 Alternative Methods

Automated instrumentation techniques are a natural solution to instrumenting large code bases and have been extensively covered in the literature [19, 33, 44, 132, 149, 166]. As mentioned in Section 2.4.1, there are three main approaches to instrumentation, these are binary, byte code and source code instrumentation; however, the use of source code based instrumentation is focused upon in this thesis.

The TAU tool set is a mature performance toolkit which along with implementations of the various instrumentation techniques, includes an automatic source instrumentation tool [111, 149]. While this tool is effective for the most part, it is unable to handle FORTRAN77 and nested loops, both of which are supported by the tool developed in this thesis.

Byfl is another tool which is capable of instrumenting code bases, with what the authors term "software counters" which can be used to record data from executing code [132]. However this tool is tied to LLVM and so limits the use of high performance compilers such as those produced by Intel, Cray and PGI. However, the tool developed in this thesis is compiler agnostic.

The work which is most similar to that presented in this thesis is that developed by Wu *et al*; their tool was developed so that instrumentation written by performance engineers could be applied to multiple codes by application engineers. In the case of HYDRA this quality is highly desirable as it allows one set of instrumentation to be applied to multiple versions of the code and potentially applied by a third party. This separation of instrumentation from code also has the additional benefit that instrumentation can be transported non-securely to the location of the HYDRA source. The tool developed in this thesis differs by focusing on the needs of a performance engineer rather than end users [166] by focusing on the flexibility of instrumentation rather than usability.

The use of analytical and simulation-based performance models has previously been demonstrated in a wide range of scientific and engineering appli-

cation domains. The construction of such models can augment many aspects of performance engineering, including: comparing the expected performance of multiple candidate machines during procurement [73], improving the scheduling of jobs on a shared machine via walltime estimates [151], identifying bottlenecks and potential optimisations to evaluate their effect upon performance ahead-of-implementation [125] and post-installation machine validation [101].

One body of work similar to that presented in this thesis is described in [59, 60, 61], in which the authors apply an analytical performance model to an algebraic multigrid application executing on a range of architectures available at the Lawrence Livermore National Laboratory (including a BlueGene/P and a BlueGene/Q). The focus of these papers is on understanding the scalability of algebraic multigrid applications and the utility of hybrid OpenMP/MPI programming. This work presents a model of a *geometric* multigrid application. This differs from previous work produced at the University of Warwick (in which several models of MPI-based codes [13, 14, 40, 123, 124] were developed) in that the focus is on using the performance model to asses the suitability of different partitioning algorithms/libraries at varying scale.

Giles *et al* have published several papers on the design and performance of the Oxford Parallel Library for Unstructured Solvers (OPlus) and its successor, OP2 [26, 35, 65, 66, 67, 126]. One of these papers details the construction of an analytical performance model of a simple airfoil benchmark (consisting of ≈2 K lines of code), executing on commodity clusters containing CPU and GPU hardware [126]. The performance model achieves high levels of accuracy, but does not support multigrid execution. This thesis details the construction of a performance model for a significantly more complex production application (consisting of ≈45 K lines of code), and presents model validations for datasets with multiple grid levels. This work additionally differs by using data provided by a mini-application in the performance modelling process.

Numerous mini-applications have been developed, some of which have been released as part of projects such as the Mantevo Project [77] and the UK Mini-

App Consortium [158]). Mini-applications from these repositories and other standalone mini-applications have been used in a variety of contexts: i) exploring new architectures, as demonstrated by Pennycook *et al* who use miniMD, a mini-application version of LAMMPS, to explore the performance of a molecular dynamics code on Intel Xeon Phi [134]; ii) examining the scalability of a neutron transport application [156]; iii) to aid the design and development of a domain specific active library [140] and, iv) exploring different programming models [110]. This work aims to provide further evidence as to the success of the mini-application approach by detailing its usage in examining new hardware features.

Benchmarks and mini-applications representing Computational Fluid Dynamics (CFD) codes already exist, but are missing some or all the application characteristics that need to be captured in order to represent unstructured mesh, geometric multigrid applications. LULESH is a hydrodynamics mini-application representative of ALE3D, however it does not utilise a geometric multigrid solver [96]. Likewise, both the Rodinia CFD [34] and AIRFOIL [65] codes developed by Corrigan *et al* and Giles *et al* respectively, utilise an unstructured mesh but are also lacking a geometric multigrid solver. Furthermore, the Rodinia CFD code does not support datasets which represent geometries using nodes with a variable number of neighbours. The HPGMG-FV benchmark, while not a CFD code, does utilise a geometric multigrid solver, however; it operates on a structured mesh [1]. The mini-application developed in this work supports datasets with the following characteristics: i) geometric multigrid, ii) a variable number of neighbours per node and iii) unstructured, all of which contribute to the spacial and temporal memory access behaviour of HYDRA.

Another body of work which is similar to that in this thesis and which it builds upon, deals with the validation of a mini-applications performance characteristics against those of the parent code. The technique employed by Tramm *et al* involves comparing the correlation of parallel efficiency loss to performance counters for both the mini-application and the original code [156].

Previously this technique has been applied to mini-applications of a neutron transport code [156] and a Lagrangian hydrodynamics code [12]; in this thesis it is applied to a different class of application. Messer *et al* develop three mini-applications and use a comparison between the scalability of the mini-application and the original code as evidence of their similarity [117]. However, the authors focus on distributed memory scalability whereas in this work the focus is on intranode scalability (OpenMP and SIMD).

2.6 Summary

This section has covered the anatomy of a parallel machine and the various forms of parallelism which can be exploited by HPC applications from the CPU core to the level of the distributed memory machine. It has been noted that multiple forms of parallelism must be exploited in order to achieve the maximum potential of any given machine.

Additionally in this section various performance engineering techniques have been covered (e.g. profiling, benchmarking and performance modelling) and how they are essential due to the complexity of the software, hardware and their respective life cycles.

CHAPTER 3

Computational Fluid Dynamics, HYDRA and Tools

One use of High Performance Computing (HPC) resource is to run Computational Fluid Dynamics (CFD) simulations and in order to construct an analytical runtime model or a performance proxy of such a simulation, the following are necessary: i) knowledge of the codes structure and to a lesser extent the underpinning mathematics; ii) specifications of the parallel architectures to be run on; and, iii) access to suitable software tooling. In this section, this prerequisite knowledge and the tools used/developed are detailed.

3.1 Computational Fluid Dynamics

Versteeg and Malalasekera define CFD as the "analysis of systems involving fluid flow, heat transfer and associated phenomena such as chemical reactions by means of computer-based simulation" [162]. The Navier-Stokes equations (in various forms) mathematically describe the flow of fluids and underpin CFD. For problems where the viscosity of a fluid can be ignored, the Euler equations can be used, which are a simplification of the Navier-Stokes equations [127]. These equations are used the following high level description of the mathematics of fluid dynamics due to their relative simplicity. The purpose of this description is not to give a rigorous example of how a simulation is formulated from first principles, but to give the reader a feel for how these simulations operate.

$$\frac{\delta(\rho u)}{\delta x} + \frac{\delta(\rho v)}{\delta y} = 0 \tag{3.1}$$

$$\frac{\delta(\rho u^2)}{\delta x} + \frac{\delta(\rho uv)}{\delta y} = -\frac{\delta p}{\delta x} \tag{3.2}$$

$$\frac{\delta(\rho uv)}{\delta x} + \frac{\delta(\rho v^2)}{\delta y} = -\frac{\delta p}{\delta y} \tag{3.3}$$

Euler's equations (Equations 3.1, 3.2, 3.3) represent the relationship between velocity (u and v), pressure (p) and density (ρ) in a moving fluid. Specifically, there is the continuity equation to enforce the conservation mass and an equation of momentum for each spacial dimension that is to be simulated (in this case two). These equations, along with an equation to relate pressure and the density of a gas [127], when discretized can be used as the starting point of constructing a 2D-simulation of fluid flow.

CFD simulations are evolved in a series of time steps, starting from an initial state (e.g. arrays of pressure, density, momentum and density energy) by the application of the discretized equations to each volume, to a final state, which is also a set of simulation properties. On such output data, analyses and visualisations can be performed, such as that represented in Figure 3.1 [119]. This shows Mach contours on a 3D bypass duct [119] which were gathered from a multigrid application with similar properties to the application considered in this thesis.

3.1.1 Uses of CFD

CFD simulations have many advantages – they can save both time and money by permitting the fast navigation of design spaces [147] without the cost of producing scale models and purchasing wind tunnel time [163]. Additionally, these methods can be used to model variables which would be difficult to measure directly [74] or to simulate conditions which would be impractical to create [162]. Due to these advantages CFD simulations have been applied to a plethora of real world problems in domains such as sport (e.g. cycling [42, 74, 108] and motor racing [4, 98, 102]), and aircraft design [32, 93] to name just a small subset.

Lukes *et al* detail the development of cycling aerodynamics over a period of 50 years and mention a study performed in 2002 by Hanna *et al*, who use

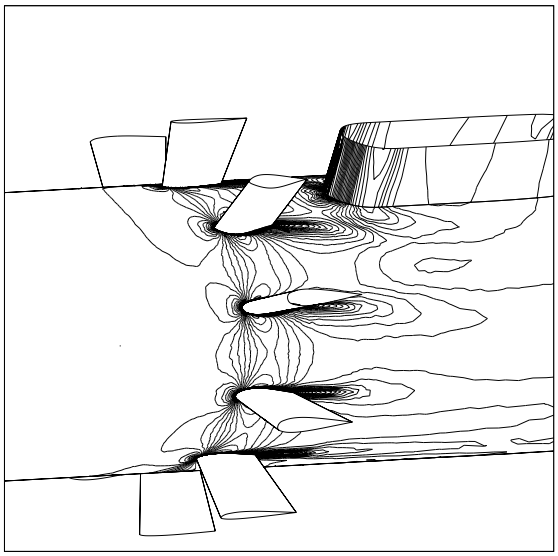


Figure 3.1: 3D Bypass duct with Mach number contours [119]

CFD methods to propose that using rear disc wheels was not as effective as first thought [74, 107]. Furthermore, Defraeye *et al* use CFD techniques to analyse the drag of cyclist in different positions [42]. The computational models built for this work potentially could be used to suggest favourable positions to adopt while racing.

Likewise, CFD methods have also played an important role in motor racing. For example, Kieffer *et al* use the Star-CD CFD code to examine the relationship between the Angle of Attack (AOA) and the lift and drag coefficients for the front and rear wings of a Formula Mazda car. Furthermore, the authors use their computational model to quantify the impact of the ground on the velocity and pressure around the front spoiler and propose using their computational model to examine how changing the height of the front spoiler affects these properties [102]. Kellar *et al* also report using a CFD simulation to optimise a front spoiler [98].

Another area where CFD methods are prevalent is the aerospace industry. Rolls-Royce use CFD codes to simulate the flow of fluids in and around some of their commercial products to optimise engine designs. One example of such

Listing 3.1: Pseudo-code for HYDRA's smooth loop

```
1  for iter = 1 to niter do
2
3     if iter == 1 then
4        call jacob   // Jacobian preconditioning
5     end if
6
7     for step = 1 to 5 do
8
9        if dissipative flux update then
10          call grad    // compute gradient
11          call vflux   // accumulate viscous fluxes
12          call wfflux  // modify viscous wall fluxes
13          call wvflux
14       end if
15
16       call iflux      // accumulate inviscid fluxes
17       call srcsa      // Spalart-Allmaras source term
18       call update     // update flow solution
19
20    end for
21
22 end for
```

an optimisation is detailed by Duta *et al* who reduce the effects of high cycle fatigue [49]. One of the main sets of applications, tools and libraries involved in such simulations is illustrated in Figure 3.2 and is the application which the research in this thesis is based upon.

3.1.2 HYDRA

HYDRA [104] is a suite of nonlinear, linear and adjoint solvers developed by Rolls-Royce in collaboration with many UK universities. The reader is referred to previous works for more information [29, 30, 48, 64, 119].

The focus of this thesis is on HYDRA's nonlinear solver, which is henceforth referred to as "HYDRA". Specifically HYDRA's smooth loop, which accounts for the vast majority of its execution time, is examined. This smooth loop contains the following functions: `vflux`, `iflux`, `jacob`, `grad`, `wfflux`, `wvflux`, `srcsa` and `update`), which are each called approximately 1–5 (Runge-Kutta [27]) times per iteration. Note that the loop does not include HYDRA's main input/output (I/O) or setup routines. Pseudo-code for this routine is shown in Listing 3.1. In this thesis results are mostly from from HYDRA 6.2.23. However, results from HYDRA 7.3.4 are occasionally presented when, for example

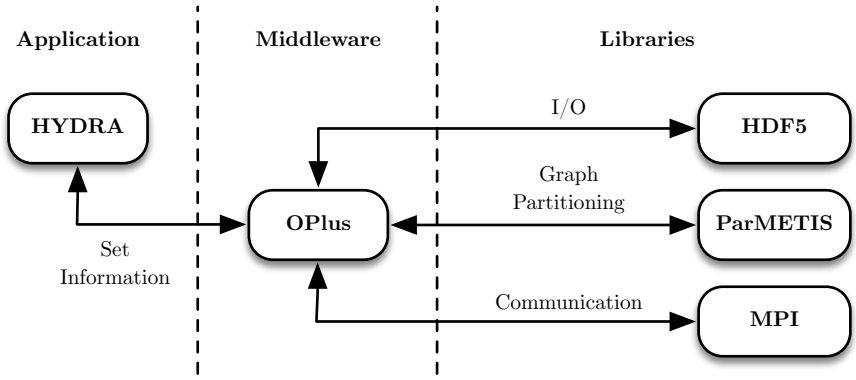


Figure 3.2: Interaction between HYDRA, OPlus and MPI

examining historical performance or the applicability of the work in this thesis to newer versions of HYDRA.

3.1.3 Multigrid

HYDRA employs multigrid methods which are designed to increase the rate of convergence for iterative solvers, and possess a useful computational property; the amount of computational work required is linear in the number of unknowns [157]. Multigrid applications operate on a hierarchy of grid levels; this thesis is concerned with geometric multigrid, wherein each grid level has its own explicit mesh geometry, and the coarse levels of the hierarchy are constructed based upon the geometry of the finest level.

Starting at the finest level, multigrid applications use an iterative smoothing subroutine to reduce high frequency errors. Low frequency errors are then transferred to the next coarsest level (restriction), where they appear as high frequency errors and can thus be more rapidly smoothed by the same subroutine. Error corrections from the smoothing of coarse levels are then transferred back to finer levels (prolongation). The order in which prolongations and restrictions are applied is know as a cycle, of which this thesis considers two types: V-cycles and W-cycles as these are both available in HYDRA. Figure 3.3(a) and Figure 3.3(b) visualise several V-cycles and two W-cycles respectively.

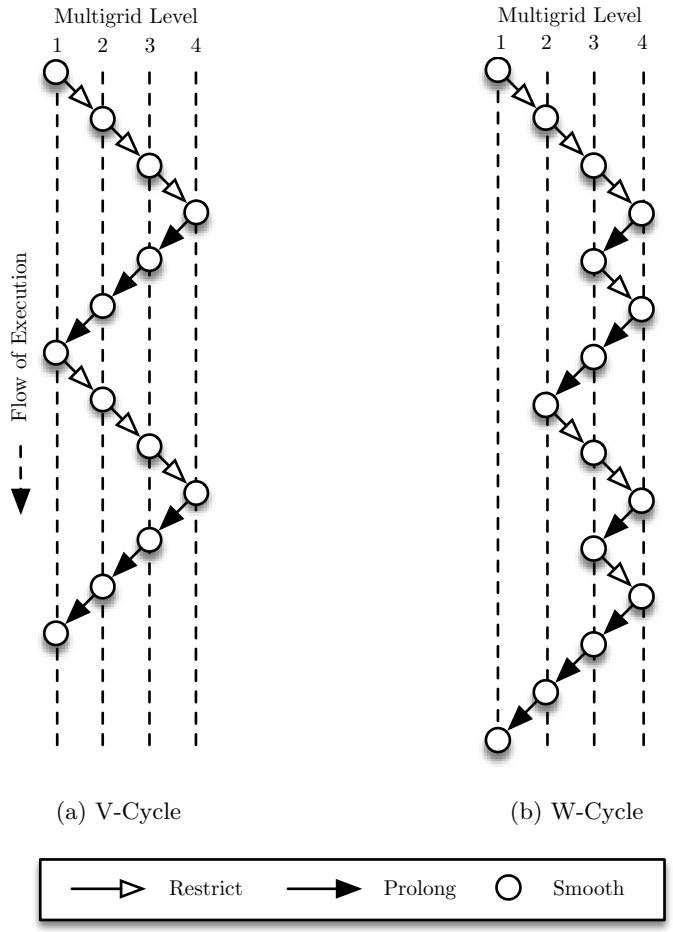


Figure 3.3: Level transition pattern for (a) two V-cycles and (b) one W-cycle

3.1.4 OPlus

HYDRA uses the Oxford Parallel Library for Unstructured Solvers (OPlus), which was designed to allow a single source code to be recompiled for serial or parallel execution, acting as a middleware that completely hides other library calls (e.g. to Message Passing Interface (MPI) or OpenMP) and the low-level implementation of a code's parallel behaviour from the programmer (see Figure 3.2) [26, 36]. Subroutines in the user source code (in this case, HYDRA) are defined as operations over user-defined data sets (e.g. nodes, edges, faces) and the library schedules the computation accordingly. When running serially,

Listing 3.2: Code-snippet from `vflux`, a typical OPlus parallel loop

```
1  do while (op_par_loop(edges, istart,
2  |          ifinish))
3  |
4  |  call op_access('read', ewt, 3,
5  |                 edges, 0, 0, ...)
6  |  call op_access('read', x, 3,
7  |                 nodes, ne, 2, ...)
8  |  ...
9  |  call op_access('update', vres, 6,
10 |                 nodes, ne, 2, ...)
11 |
12 |  do ie = istart, ifinish
13 |
14 |    i1 = ne(1,ie)
15 |    i2 = ne(2,ie)
16 |    call vflux_edge(ewt(1,ie),
17 |                    x(1,i1), x(1,i2),
18 |                    vres(1,i1), vres(1,i2))
19 |
20 |  enddo
21 |
22 |enddo
```

OPlus uses a standard loop to execute the subroutine for each set element; when running in parallel, the set elements (and their computation) are partitioned over multiple nodes. OPlus is also responsible for handling the halo exchanges at the boundaries between processor domains, i.e there are no calls to the MPI library or any other communication library within the HYDRA source code as the MPI calls are handled by OPlus.

Listing 3.2 gives an example of an OPlus parallel loop. It should be noted that nothing in the source code (besides the name of the `op_par_loop` function) suggests that this loop will be executed in parallel. The inner loop (from `istart` to `ifinish`) is a standard Fortran `do` loop, which calls a subroutine for each edge in the unstructured mesh.

In order to schedule such a loop for parallel execution, OPlus requires that the programmer declare how each data array will be accessed, via calls to `op_access`. Firstly, they must declare an access type for each array, read, write, or read/write ("update"). OPlus then attaches a "dirty bit" to each array, based upon these access modifiers; if an array is declared as being "write" or "update", then execution of the loop will invalidate any copy of the data held on neighbouring processes. Secondly, the programmer must specify whether the

array is to be accessed directly (i.e. the array index is the loop counter) or indirectly (i.e. the array index is the result of a look-up, based on the loop counter); such information allows OPlus to reason about whether a given loop requires data only from local set elements, or is likely to access data residing on another processor.

When combined with the set partitioning, these access descriptors permit OPlus to determine which iterations of the inner loop:

1. Can be executed prior to communication;
2. Require communication with neighbouring processors to ensure correctness; and,
3. Should be executed redundantly on multiple processors to avoid additional communication steps.

The set elements corresponding to such iterations are referred to henceforth as independent, halo and execute set elements respectively. When OPlus is operating using MPI communication proceeds as follows:

1. Wait until each outstanding send has completed (using `MPI_Waitany`) before packing send buffers.
2. Initiate a swap of halo information with each neighbouring processor (`MPI_Irecv` and `MPI_Isend`).
3. Carry out independent computation.
4. Wait until each outstanding receive has completed (again using `MPI_Waitany`) before unpacking receive buffers.
5. Carry out dependent and redundant computation.

The `do while` loop surrounding the computation enables OPlus to iterate over the three distinct subsets of elements in a way that is transparent to the programmer. The `op_par_loop` call returns true for a certain number of calls

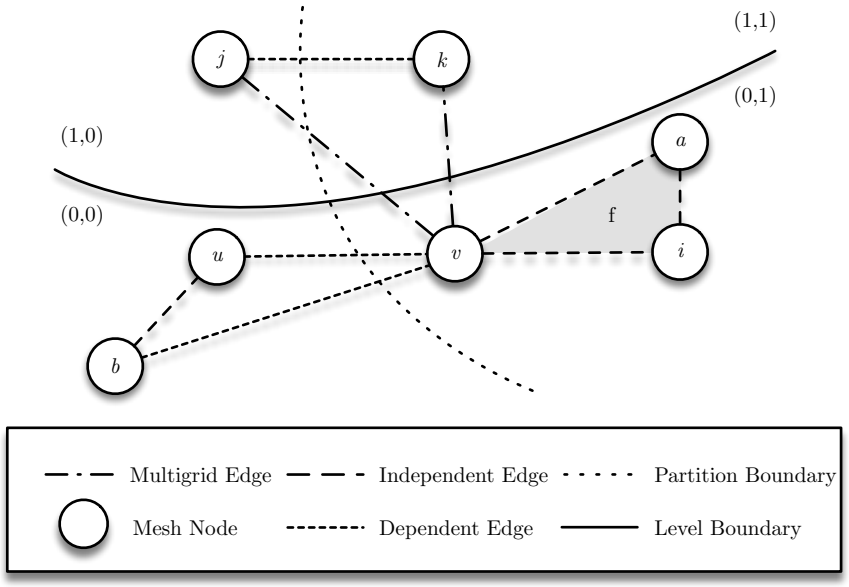


Figure 3.4: Abstract representation of an unstructured mesh dataset over two multigrid levels

(thus continuing the while loop) and sets the values of `istart` and `ifinish` to different values each time (thereby controlling the set elements executed by a given iteration).

The experiments in this thesis use OPlus version 6.03 in conjunction with HYDRA 6.2.23, however in some instances a modified version of OPlus 6.03 is used which contains a distributed memory communication optimisation. Additionally, a more recent version of OPlus (version 6.12) is used in conjunction with HYDRA 7.3.4.

3.1.5 Datasets

HYDRA represents its input geometries using an unstructured mesh (see Section 2.2.2) and since HYDRA employs a multigrid solver (see Section 3.1.3) the input geometry is replicated from a fine to a coarse resolution for levels one to n of the multigrid. Consecutive levels are linked using a set of edges (distinct from the edges which connect mesh nodes on the same level) which represent how the

physical properties of the simulations are prolonged and restricted between the multigrid levels; Figure 3.4 visualises this additional type of edge. For example the edge between j and v in Figure 3.4 is a multigrid edge.

Experiments in this thesis use two different datasets. First, the LA Cascade dataset [80, 81] which is a mesh of ≈105K nodes and ≈305K edges (at the finest level) representing a rectilinear cascade of turbine blades. Second, the NASA Rotor37 dataset [141], a larger mesh of ≈746K nodes and ≈2.2 million edges (at the finest level) representing an axial compressor rotor (Rotor37 (2M)). Additionally, a larger variant of Rotor37 (Rotor37 (8M)) is used which has ≈8 million nodes and ≈24.8 million edges. Both datasets have four levels, and follow similar execution paths through HYDRA. The primary difference identified between the two datasets is the inclusion of an additional OPlus loop not called by LA Cascade, due to the introduction of a non-zero rotational parameter by Rotor37.

In Chapter 4 the runtime performance model is validated on both LA Cascade and the smaller Rotor37 dataset. In Chapters 5 and 3 the larger Rotor37 dataset is used for all experiments. Finally, the LA Cascade dataset is used for all experiments in Chapter 6.

3.1.6 Mesh Partitioning Libraries

As mentioned in Section 2.2.2, when using unstructured mesh datasets on distributed memory machines they must be partitioned using a partitioning algorithm. OPlus has been developed such that different unstructured mesh partitioning algorithms and libraries can be integrated and used, such as the ParMETIS [97] library, the PTScotch [128] library and the Recursive Inertial Bisection (RIB) algorithm [63, 152, 164].

All HYDRA experiments in Chapters 3 and 4 use the RIB implementation built into OPlus. HYDRA experiments in Chapter 5 use ParMETIS 3.1 and RIB. Partitioning data is also collected from METIS 5.1.0, PT-Scotch 6.0.4 and Scotch 6.0.4, but is not used in conjunction with HYDRA/OPlus.

Hardware	
	Napier
Nodes	360 × E5-2697 v2
Cores per node	24
Frequency (GHz)	2.7
RAM per node (GB)	64
Interconnect	FDR Infiniband
Software	
Operating System	Red Hat 4.4.7
Compilers	Intel 16
MPI	Intel MPI 5.1.1
Storage	GPFS
Resource Manager	SLS

Table 3.1: Hardware/software configuration of Napier

Hardware	
	Power8
Nodes	1 × Power8E
Cores per node	20
Frequency (GHz)	3.69
RAM per node	16GB-1TB
Interconnect	N/A
Software	
Operating System	Red Hat Enterprise Linux 7
Compilers	XLF 15.1.1 and XLC 13.1.1
MPI	MPICH 3.1.2
Storage	N/A
Resource Manager	N/A

Table 3.2: Hardware/software configuration of Power8

3.2 Parallel Machine Resources

Throughout this thesis data is presented from several parallel machines (see Tables 3.1, 3.2, 3.3, 3.4, 3.5). For the most part, these are standard commodity clusters which fit the description and examples provided in Section 2.1 (e.g. Intel Central Processing Unit (CPU) with an Infiniband network). The only exceptions to this are ARCHER (Table 3.3), which uses the custom Cray Aries interconnect [52] and the Power8 machine 3.2, which holds two Power8 CPUs [55].

Minerva is a now decommissioned cluster formerly based at the University of Warwick, built from Intel Westmere-based CPUs and connected with a QDR Infiniband network (see Table 3.4 for more information). Data collected from

Hardware	
Nodes	4544 × Intel E5-2697 v2
Cores per node	24
Frequency (GHz)	2.7
RAM per node (GB)	64
Interconnect	Aries Interconnect
Software	
Operating System	CLE
Storage	Lustre
Resource Manager	PBS Professional

Table 3.3: Hardware/software configuration of ARCHER

Hardware	
	Minerva
Nodes	396 × Intel Xeon X5650
Cores per node	12
Frequency (GHz)	2.66
RAM per node (GB)	24
Interconnect	QLogic TrueScale 4X QDR InfiniBand
Software	
Operating System	SLES11
Compiler	Intel 13
MPI	Open MPI 1.4.3
Storage	IBM GPFS
Resource Manager	Moab/Torque

Table 3.4: Hardware/software configuration of Minerva

Minerva is used only in Chapter 4 to conduct performance model validations.

Tinis, Minerva's successor, is built from Intel Haswell-based CPUs and is connected with the same QDR Infiniband network (see Table 3.5 for more information). Data collected from Tinis is used to i) perform the performance model validations in Chapter 5; ii) validate mini-HYDRA in Chapter 6; and, iii) examine the impact of the new instructions provided by the Intel Haswell platform on HYDRA's `iflux` routine in Chapter 6.

ARCHER is a Cray XC30 MPP supercomputer maintained by the EPCC based in Edinburgh. Each of its nodes has two 12-core Intel Ivybridge CPUs and inter-node communication occurs via the custom Cray Aires interconnect [52] (see Table 3.3 for more information). Results from ARCHER were used to identify the increased buffer pack/unpack costs caused by the larger Rotor37

Hardware	
	Tinis
Nodes	200 × E5-2630 v3
Cores per node	16
Frequency (GHz)	2.4
RAM per node (GB)	32
Interconnect	QDR Infiniband
Software	
Operating System	CentOS 6.7
Compiler	Intel 15
MPI	Open MPI 1.6.5
Storage	GPFS
Resource Manager	Slurm/Moab

Table 3.5: Hardware/software configuration of Tinis

datasets, which motivated the extensions to the performance model presented in Chapter 5.

Napier is a commodity cluster formed from Intel Ivybridge-based CPUs at the Hartree Centre in Daresbury, UK (see Table 3.1). Results from this machine were collect in support of i) the work in Chapter 6, where the results are used in a comparison against results collected from an Intel Haswell machine; and, ii) the work in Chapter 3 where HYDRA's historical performance is examined.

The Power8 architecture was developed by IBM which supports 8-way Simultaneous Multithreading (SMT). Results from this chip were collected for use in Chapter 3 to examine the impact of varying the degree of SMT on HYDRA's kernels.

3.3 Auto-instrumentation

Having the appropriate tools to gather performance information from HYDRA is essential for the construction of performance models and the development of mini-applications. However instrumenting HYDRA poses several challenges.

The size of the code base presents a major hurdle to instrumentation as it contains over 300 loops. Although, as mentioned in Section 3.1.2, HYDRA spends the vast majority of its runtime in a small subset of its loops, the loops

which are invoked depends on the input deck (i.e. Rotor37 uses additional loops over LA Cascade) and the command line options used. Hand instrumenting all of these loops is infeasible especially when instrumenting varying subsets of these loops to reduce instrumentation overhead. The time taken to instrument the entire code base by hand is further exacerbated by the fact that HYDRA is in continuous development, so multiple versions exist (both releases with new features and optimised variants) which need to be instrumented.

Restrictions on where the code can be run, compiled and by who adds additional complexity to applying instrumentation to the code base as it is often not possible for a performance engineer to place instrumentation in themselves. This issue is also compounded by the fact that there are multiple versions of the code, so often a performance engineer has never seen the source.

HYDRA's reliance on OPlus (see Section 3.1.4) to perform parallel computations further complicates the process of instrumentation, as its Application Programmer Interface (API) increases the complexity of the code's structure, as it requires the use of nested loops and hides the communication behaviour. The former increases the complexity of inserting instrumentation as any code analysis will be required to maintain state in order to handle multiple loop nests and the latter requires both HYDRA and OPlus to be instrumented.

The fact that HYDRA is a multigrid code also increases the complexity of the source and thus complicates instrumentation attempts as at any particular point during execution the code could be on any level of the multigrid. That is to say the instrumentation can be inserted such that collected performance information needs to be associated with the particular multigrid level it was collected from.

3.3.1 Instrumentation Process

The source code parsing which underpins the auto-instrumentation tool has been designed as a process, so that it can be implemented in any programming language in order to promote its use on a wide variety of platforms. This process

Listing 3.3: An example auto-instrumentation rule which matches the entry to an OPlus parallel loop, and action, to insert timers and gather the set name associated with this loop

```
1  {
2      ''regex'':
3          ''
4              (dowhile)\(hyd_par_loop\(([^,]*),
5              ([^,]*),([^,]*),([^,|\)]*)\)
6          '',
7      ''action'' : loopEnter,
8      ''state'' : loopRulesState,
9      ''log'': ''enter loop instrumentation''
10 }
```

comprises three steps, which are as follows:

1. The program code is preprocessed before compilation to remove comments, squash line continuations and remove all white space;

2. A set of condition action rules are passed over this preprocessed code, where the conditions are segments of code (e.g. loop bounds) and the actions insert instrumentation (e.g. to call a timer or record a variable's value over time);

3. Finally the code is post-processed to return the original formatting (white space and comments).

The tool first preprocesses the program code based on features common to the vast majority of languages used for scientific codes (Fortran and C variants), so that adding support for new languages consists only of defining the tokens for comments, new lines and line continuations. This avoids the need to add an entirely new lexer and parser when adding support for additional languages, which is a task methods reliant on the ROSE compiler [138] would require.

The rule conditions are regular expressions which are used to match the preprocessed source lines, this means most of the language specific details are encoded in the rule sets themselves rather than in a lexer and parser. An example rule is provided in Listing 3.3; this rule matches the entry to OPlus parallel loops and additionally defines regular expression groups (donated by rounded

brackets) so that static information from the OPlus API can be extracted and used by the action.

The actions range in complexity from simple (e.g. inserting lines of code before and after a match) to complex (e.g. running arbitrary code with which to alter the source). In the case of the example in Listing 3.3, the action is a call to a function which records the entry to an OPlus loop in the state dictionary (`loopRulesState`) and stores the values from the regular expression groupings to disk. The hyd_par_loop API call encodes the set which the loop iterates over, this information is collected by this rule as it is essential for performance modelling activities. Each action may also log output, this feature is required for confirming that all the instrumentation has been inserted, when applying an existing rule set to a different version of the code.

The final step is arguably the most important, as it maintains the readability of the code and allows the correctness of the instrumentation to be verified by performing a `diff` between the instrumented and non-instrumented source.

A realisation of this process was developed in Python, and overcomes the challenges with instrumenting HYDRA as follows.

1. The size of the code. The instrumentation tool is a compiler wrapper which modifies the input source code according to a set of rules, i.e. the same source modifications can be applied over an arbitrarily sized code base.

2. There are restrictions on where HYDRA can be compiled and run. The auto-instrumentation is primed with a rule file, which can be freely distributed (unlike the HYDRA source) to the location of the source, rather than needing to send the instrumented source code to performance engineers, or the performance engineers to the source code.

3. There are several challenges related to the complexity of the source code, such as the age of the language HYDRA is written in, the reliance on a domain specific library, the existence of nested loops and the fact it is a

Loop Name	Identifier
vflux	FLUX2
iflux	FLUX6
jacob	UPDATE12
grad	FLUX8
wfflux	FLUX3
wvflux	FLUX4
srcsa	FLUX27
update	UPDATE8

Table 3.6: Mapping between loop names and the identifiers assigned by the auto-instrumentation tool

multigrid code. The auto-instrumentation tool overcomes these challenges by allowing a large amount of flexibility in both the rules and the actions.

The overhead of this auto-instrumentation on an application's runtime is tied directly to the instrumentation/library calls that are being inserted, so does not in and of itself add any measurable overhead.

3.4 Auto-instrumentation Case Studies

In order to demonstrate the effectiveness of the auto-instrumentation tool it is used to aid in the execution of two performance studies. The first analyses the impact of increased parallelism opportunities made available through SMT on HYDRA's (version 6.2.23) runtime when using the Power8 architecture, and the second examines how HYDRA's runtime performance has changed through subsequent releases (versions 6.2.23 and 7.3.4). Both of theses studies stress the auto-instrumentation tool in a different way, with the former testing its ability to operate problem free on a completely alien platform and the latter stresses its ability to operate over different versions of the code base.

In the following case studies, HYDRA's loops are assigned identifiers by the auto-instrumentation tool and not by the names used in Section 3.1.2. A mapping between these identifiers is provided in Table 3.6 so that the names can be related back to their descriptions in Listing 3.1. Furthermore, the loop identifiers change depending on a loop's position within a source file, therefore

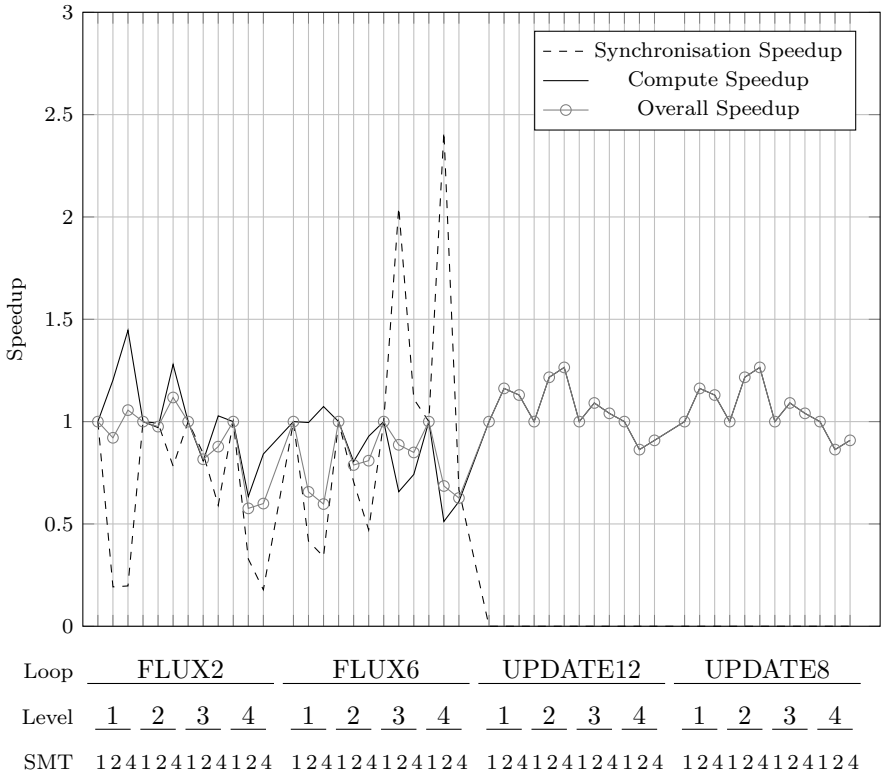


Figure 3.5: HYDRA's per OPlus loop speedup when compared to SMT1 on Power8 (`FLUX2`, `FLUX6`, `UPDATE12`, `UPDATE8`)

the auto-instrumentation tool may assign different identifiers to the same loop depending on the version of HYDRA. Where a comparison between different versions of HYDRA is performed, the loop identifiers have been changed to match those of the older version.

3.4.1 Effect of Power8 SMT Degree on Runtime

The results in this section were collected from the IBM Power8 machine defined in Table 3.2, executing HYDRA primed with the Rotor37 (8M) dataset. Figure 3.5 and Figure 3.6 show the speedup of the most expensive loops in HYDRA, broken down by compute and synchronisation/communication cost, when using varying levels of SMT. The confidence intervals for these results are given in Table 3.7; there is no confidence interval for SMT level 4 as multiple

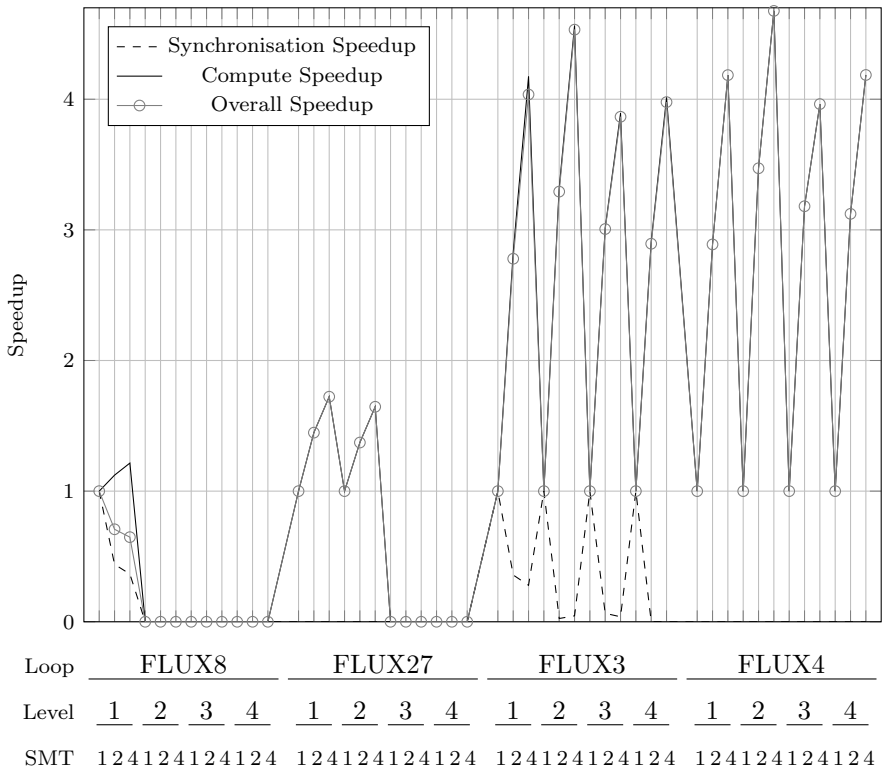


Figure 3.6: HYDRA's per OPlus loop speedup when compared to SMT1 on Power8 (`FLUX8`, `FLUX27`, `FLUX3`, `FLUX4`)

SMT	Confidence Interval
1	10.87
2	17.65

Table 3.7: Confidence intervals for HYDRA's runtime on Power8 at different SMT levels

runs were not obtained due to restricted time on the machine. However, this is not expected to be an issue due to the low variance observed for the results from SMT levels 1 and 2.

The usage of SMT on the Power8 architecture has a positive impact on the compute (sum of independent, halo and execute times) performance, especially for `FLUX2`, but this improvement diminishes and in some cases reverses (as is the case for `UPDATE12`) on the coarser levels of the multigrid (Figure 3.5). However, for loops which have associated communication activities (e.g. `FLUX2` and `FLUX6`, `FLUX8`), increasing the degree of SMT also increases synchronisation cost (see

Version	Confidence Interval
6.2.23	13.22
7.3.5	27.86

Table 3.8: Confidence intervals for HYDRA's Runtime on Napier

Figure 3.5 and Figure 3.6). This increased synchronisation cost outweighs the gains from the reduced computation time in the case of both `FLUX6` and `FLUX8`. Despite this, `FLUX2` does experience an overall improvement at SMT4 on levels one and two of the multigrid (Figure 3.5), most likely because it is the most compute heavy routine in HYDRA. Overall, these changes in performance result in an slowdown when increasing the degree of SMT used.

The display of data in Figure 3.5 and Figure 3.6 demonstrates the ability of the auto-instrumentation to enable easy data collection despite the challenges. From the data itself; the reduction in compute time is promising and warrants further investigation towards reducing the synchronisation costs on this architecture. One avenue to reduce these overheads would be to construct an OpenMP version of HYDRA rather than relying on MPI for intranode communication. Furthermore the results have highlighted the varying responses to SMT levels on a per loop, per level basis, this information could be used to intelligently vary degrees of parallelism during execution.

3.4.2 Highlighting Historical Performance Differences

Figure 3.7, Figure 3.8, Figure 3.9 and Figure 3.10 show the runtime of the most expensive loops broken down by the region of compute, communication+sync and multigrid level (see Section 3.1.4) for versions 6.2.23 and 7.3.4 of HYDRA. When considering just the two most expensive loops (in terms of runtime) it is apparent that `FLUX2` and `FLUX6` are quicker (1.14× and 1.72× respectively) due to reductions in synchronisation cost. By examining the most expensive loops from version 6.2.23 it could be concluded the code has undergone positive changes in terms of performance; however, this improvement is not reflected in the overall runtime of both versions. This analysis highlights that the increase

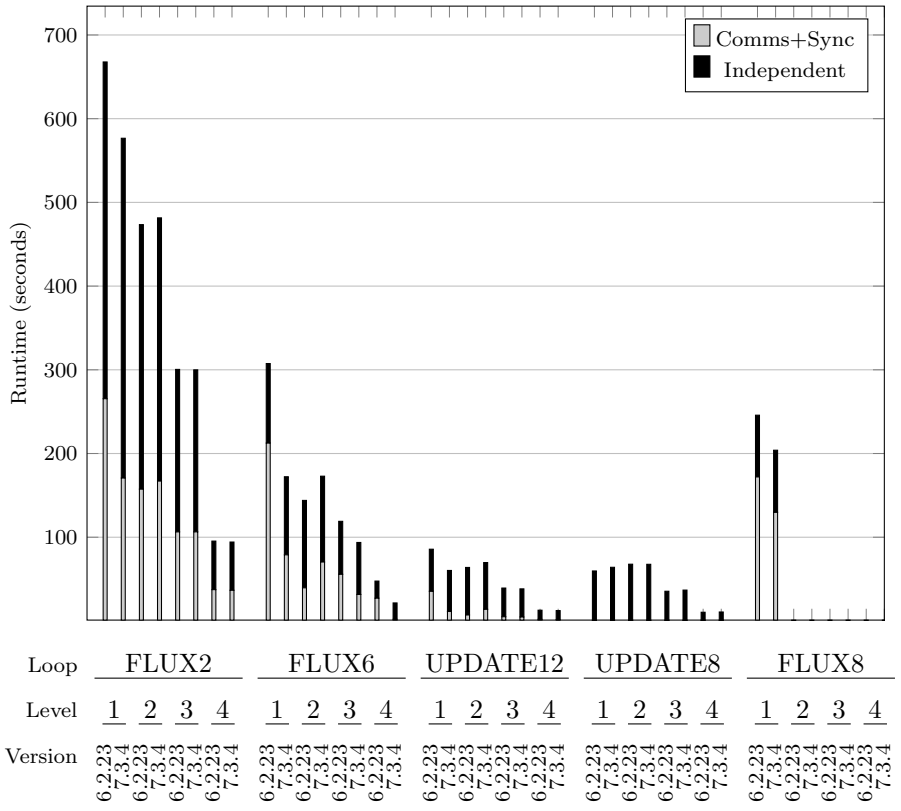


Figure 3.7: Per loop comparison between HYDRA 6.2.23 and HYDRA 7.3.4 on Napier for independent compute and comms+sync (`FLUX2`, `FLUX6`, `UPDATE12`, `UPDATE8`, `FLUX8`)

in runtime is due to the fact that the synchronisation cost for previously inexpensive loops has been increased by several orders of magnitude (`FLUX25` and `BCS7`). The confidence intervals for these results are given in Table 3.8.

3.4.3 Other Uses

This instrumentation tool has been a cornerstone to the collection of results in this chapter and those which follow. In Chapter 4 the tool is used to collect the performance data (e.g. loop time, communication time, synchronisation time, set and halo sizes) for use with a runtime performance model. Additionally the tool was used to effortlessly instrument two different variants of HYDRA (one being optimised to reduce synchronisation cost) in preparation for a comparison

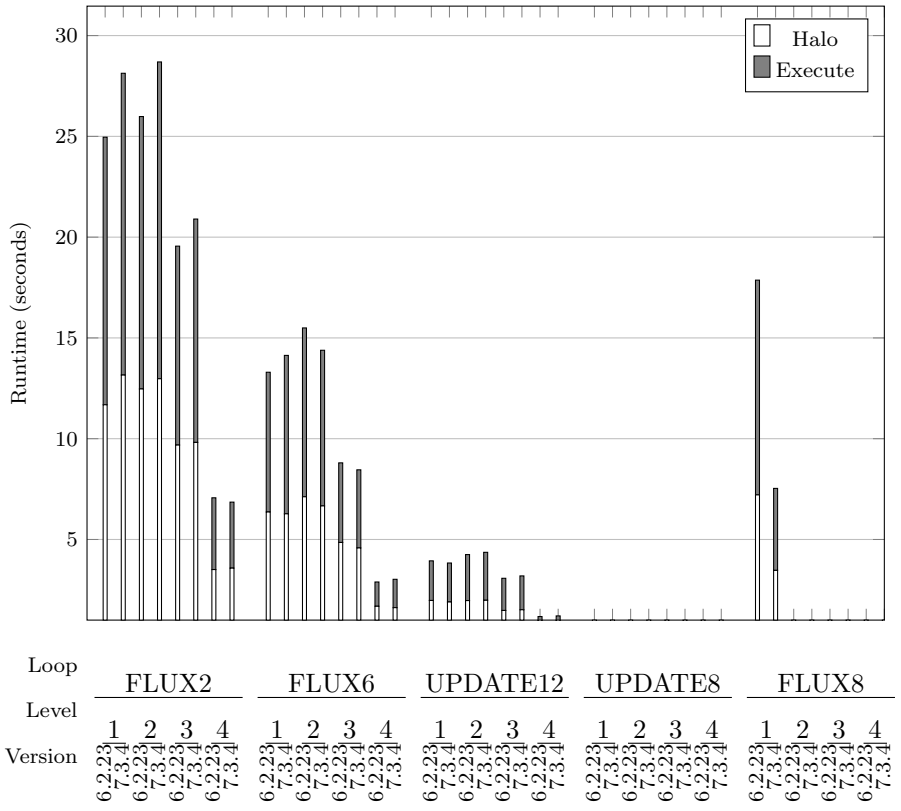


Figure 3.8: Per loop comparison between HYDRA 6.2.23 and HYDRA 7.3.4 on Napier for halo and execute compute (`FLUX2`, `FLUX6`, `UPDATE12`, `UPDATE8`, `FLUX8`)

between versions. In Chapter 5 the instrumentation rules are extended to collect additional information from OPlus (buffer pack/unpack time) as it became apparent that these costs needed representation during performance modelling activities on larger datasets.

In experiments which are not detailed in this thesis, the auto-instrumentation has been used with a single rule-action set to instrument three different chronologically released versions of HYDRA. The instrumentation has enabled a strong scaling study to be performed, revealing how its performance has changed (on a per loop basis) over several generations. Additionally, the same rule-action set has been used to successfully instrument and collect performance data from HYDRA on a variety of platforms including an IBM BlueGene/Q

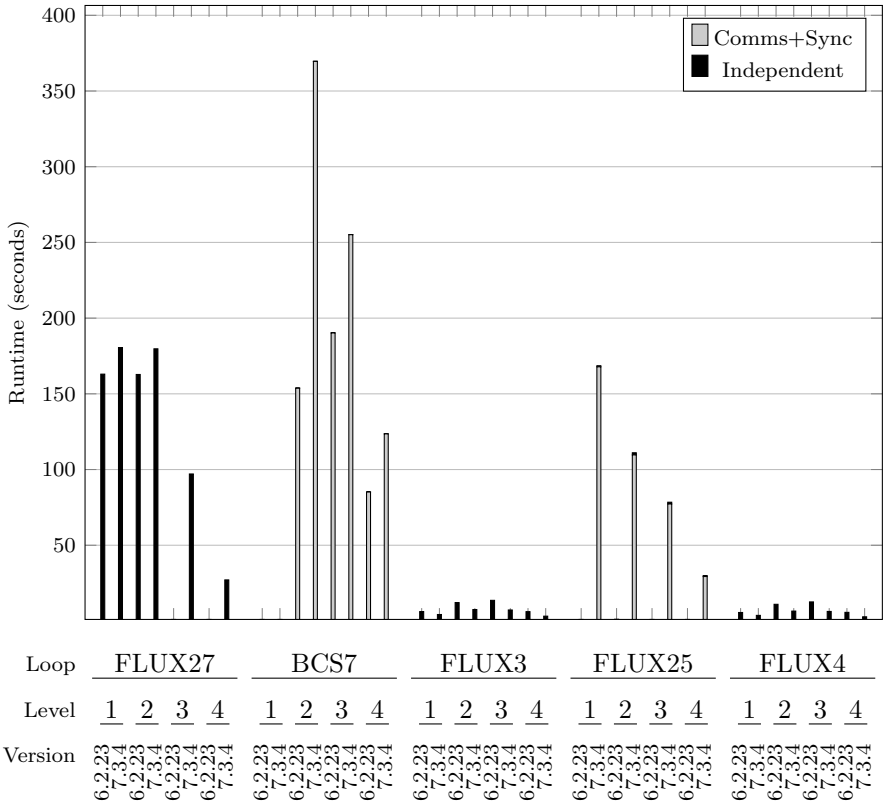


Figure 3.9: Per loop comparison between HYDRA 6.2.23 and HYDRA 7.3.4 on Napier for independent compute and comms+sync (`FLUX27`, `BCS7`, `FLUX3`, `FLUX25`, `FLUX4`)

and an Ivybridge-based cluster. Furthermore, the tool has been used to instrument HYDRA using a variety of libraries such as the Performance Application Programming Interface (PAPI) [18], perf-dump and a custom timing library.

3.5 Summary

This chapter began by introducing CFD as the study of fluid dynamics by way of computer simulations, and detailing the use that these simulations bring to various domains such as sports and aeronautics. This plethora of applications motivates the focus on this class of codes.

Next HYDRA was introduced, a CFD code developed by Rolls-Royce, and its proprietary communications library (OPlus) as the code bases that will be

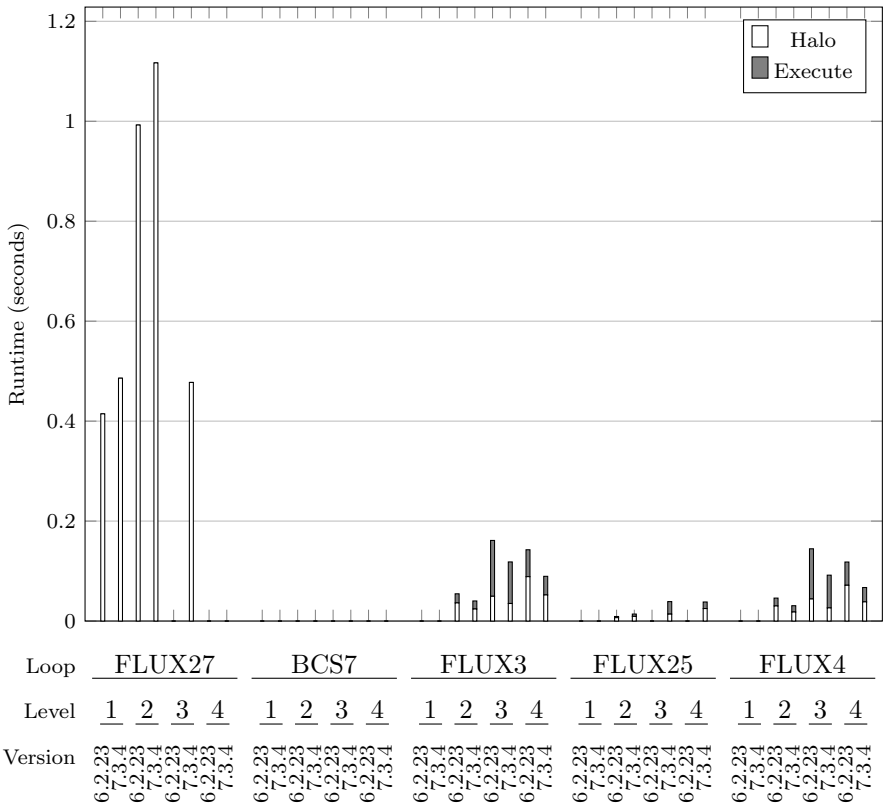


Figure 3.10: Per loop comparison between HYDRA 6.2.23 and HYDRA 7.3.4 on Napier for halo and execute compute (`FLUX27`, `BCS7`, `FLUX3`, `FLUX25`, `FLUX4`)

used as part of the case studies presented in this thesis. HYDRA is categorised as a geometric multigrid, unstructured mesh code and each of these properties and their potential performance implications on current generation hardware is discussed. Additional details specific to the operation of HYDRA and OPlus are also presented, as a detailed understanding of a codes operation assists in the application of performance modelling techniques; the focus of this thesis.

Lastly, an auto-instrumentation tool is developed to overcome the challenges associated with instrumenting HYDRA. The design of this tool is presented along with a demonstration of its use through two case studies: i) an assessment of the impact of the different SMT modes available on the Power8 platform on HYDRA's runtime; and, ii) an analysis of HYDRA's performance over different versions of the code.

CHAPTER 4

Model-led Optimisation of an Unstructured Multigrid Code

This chapter details the construction of an analytical performance model of HYDRA, a production non-linear multigrid solver used by Rolls-Royce for computational fluid dynamics simulations. The model captures both the computational behaviour of HYDRA's key subroutines and the behaviour of its proprietary communication library, OPlus, with an absolute error consistently under 16% on up to 384 cores of an Intel X5650-based commodity cluster.

A demonstration is provided of how the performance model can be used to highlight performance bottlenecks and unexpected communication behaviours, thereby guiding code optimisation efforts. Informed by model predictions, an optimisation is implemented in OPlus that decreases the communication and synchronisation time by up to 3.01× and consequently improves total application performance by 1.41×.

4.1 Experimental Setup

The hardware/software configuration for all experiments are listed in Table 3.4. For the single level model validations in Section 4.2, the results are gathered using the LA Cascade dataset (see Section 3.1.5). For the multilevel model validations in Section 4.4 the NASA Rotor37 (2M) dataset is also used.

Term	Definition
	Hardware Parameters
P	Set of processors.
α_{intra}	Intra-node (memory) latency.
β_{intra}	Inverse of intra-node (memory) bandwidth.
α_{inter}	Inter-node (network) latency.
β_{inter}	Inverse of inter-node (network) bandwidth.
	Dataset Parameters
n_{cycles}	Number of cycles (assumes convergence not reached).
n_{start}	Number of smoothing iterations in the first cycle.
n_{crs}	Number of smoothing iterations on the coarsest level.
n_{pde}	Number of partial differential equations (PDEs).
	Measured Parameters
$W_{g,l}$	Grind-time per set element in loop l.
$N_{i,l}$	Number of independent set elements in loop l.
$N_{h,l}$	Number of dependent (halo) set elements in loop l.
$N_{e,l}$	Number of redundant (execute) set elements in loop l.
B_l	Number of bytes sent per set element in loop l.
$n_{\text{calls},l}$	Number of calls to loop l in each smoothing iteration.
	Derived Parameters
C_r	Communication cost for reductions.
C_l	Communication cost for loop l.
W_l	Wall-time for loop l.
W_{smooth}	Wall-time for a single smooth iteration.
W_{total}	Total wall-time.

Table 4.1: Description of compute and communication model terms for a single-level HYDRA run

4.2 Single-Level Model

4.2.1 Model Construction

Model construction is begun by modelling the expected behaviour of HYDRA, based on examination of its source code and its intended design. Specifically, two assumptions are made:

1. The communication cost of exchanging halo data for a given loop will be hidden behind the independent compute for that loop as stated by the OPlus design [26].

2. All loops proceed in a bulk synchronous parallel (BSP) fashion.

Based on these assumptions, a description of a loop's (l) runtime can be formed

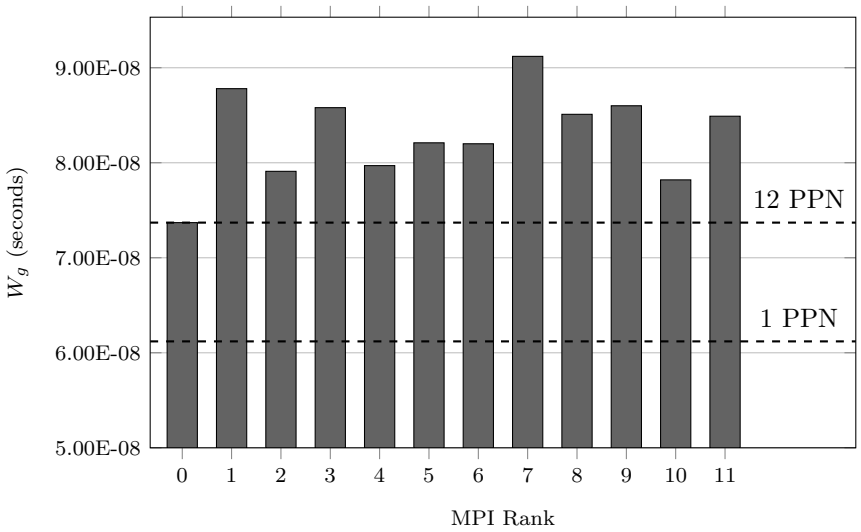


Figure 4.1: Comparison of `iflux` W_g values for 1 and 12 PPN on a single node

as follows:

$$\begin{aligned} W_l = \max(N_{i,l} \times W_{g,l}, C_l) \\ +(N_{h,l} + N_{e,l}) \times W_{g,l} \end{aligned} \tag{4.1}$$

That is, the maximum of the compute time for N_i independent set elements and the communication time for $N_h + N_e$ halo/execute elements (to account for potential overlap), plus the compute time for halo/execute elements (which are dependent on communication and thus cannot be overlapped).

Compute costs are calculated using grind-times (W_g values), which represent the processing time associated with a single loop iteration (and hence a single set element). These are too small to be measured directly due to the resolution of the timers available. Therefore, they are instead calculated from the execution times of complete loops (W_l values). For runs with more than one Processors Per Node (PPN), it is necessary to account for the fact that Message Passing Interface (MPI) tasks will contend for a node's shared resources (e.g. DRAM bandwidth); as shown in Figure 4.1, the performance effects of such contention can be significant, increasing grind-times (and hence loop times) by ≈30%. Contention effects may also differ per core; since the aim is to predict

the best-case, the minimum observed W_g is used (marked in Figure 4.1 by the 12 PPN dashed line) in the model. These W_g values are recorded empirically from benchmarking runs of an instrumented version of HYDRA, which contains profiling calls associated with OPlus loops. Using Listing 3.2 as an example OPlus loop, profiling calls are placed:

1. Surrounding the entire code block to collect the loop name, start and end times; and,
2. Around the inner loop to record the number of elements in the independent, halo and execute sets.

Halo exchange communication costs are calculated using a collection of simple bandwidth/latency models [79]:

$$C_l = \alpha_{\text{intra}} + \beta_{\text{intra}} \times (N_{h,l} + N_{e,l}) \times B_l \tag{4.2}$$

where the values of α (latency) and β (reciprocal bandwidth) are recorded from the Intel MPI Benchmark (IMB) [88], and the number of bytes per element (B) is calculated from `op_access` declarations on a per loop basis. Note that the intra- and inter-node versions of these equations are the same, and that these two parameterisations are used interchangeably by the model (depending on whether the destination processor is on- or off-node). A given network interface is then modelled using a number of α and β pairs (a piecewise linear regression), to account for dynamic optimisations performed by MPI, such as the transition between the "rendezvous" and "eager" communication protocols.

The geometric meshes that HYDRA operates on are unstructured (i.e. node connectivity must be stored explicitly in a list), and thus the way that the mesh is partitioned over processors has a significant effect on execution time. In order to accurately determine the amount of computational work and message sizes per node, the N_i, N_h, and N_e values are calculated for each processor count using an instrumented version of the Oxford Parallel Library for Unstructured Solvers (OPlus).

Since a model of HYDRA's critical path is sought, the wall-times of each loop are taken as the maximum across all processors before being combined into a time per smooth iteration:

$$W_{\text{smooth}} = \sum_{l} n_{\text{calls},l} \times \max_{p \in P} (W_l) \tag{4.3}$$

where the number of calls to each loop is recorded from an instrumented version of HYDRA. This process is straightforward for loops which are called every iteration (e.g. `jacob` and `iflux` from Listing 3.1). However, loops which are called or communicate only when some condition holds are associated with a probability, e.g. `vflux` from Listing 3.1 is modelled as occurring with a probability of 0.6, and the communication frequency of `iflux` is modelled with a probability of 0.4. These conditional execution patterns are discovered by comparing the worst case number of calls and bytes sent, to the actual values reported by HYDRA on a per loop basis. If a significant difference is apparent, then a probability is derived from the code logic.

Additional communication is required at the end of each cycle, to calculate total residual error and to print messages to the screen. The costs of these communication steps are captured in a separate model term:

$$\begin{aligned} C_r = &\log_2(|P|) \times (n_{\text{start}} + (n_{\text{cycles}} - 1)) \\ &\times [(\alpha + \beta \times 78) + (4\alpha + \beta \times (80 n_{\text{pde}} + 24))] \end{aligned} \tag{4.4}$$

where: 78 is the number of bytes exchanged before a `printf` call; the calculation of residual error features four reductions of $20n_{\text{pde}}+6$ bytes each; and reductions are assumed to require $\log_2(|P|)$ messages.

To predict total execution time, the total number of smooth iterations must be calculated. The first of HYDRA's cycles always consists of n_{start} smooth iterations, and all other cycles (on a single level dataset) execute n_{crs} smooth

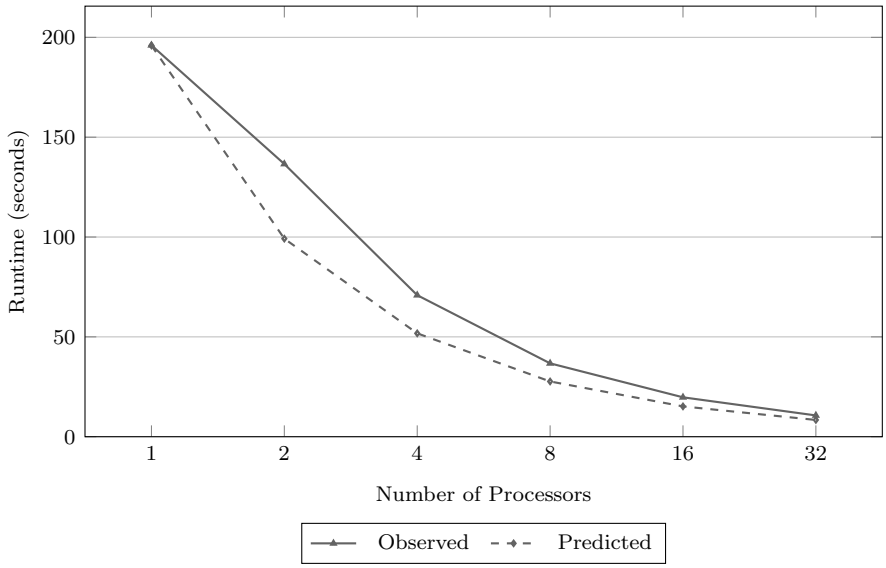


Figure 4.2: Comparison of recorded and predicted execution times for single-level runs of the original HYDRA using 1 PPN

iterations; therefore the total execution time can be modelled as below:

$$W_{\text{total}} = C_r + (n_{\text{start}} + n_{\text{crs}} \times [n_{\text{cycles}} - 1]) \times W_{\text{smooth}} \tag{4.5}$$

For clarity, a description of each term in the model can be found in Table 4.1.

The extension of the model to account for the effects of background machine noise (due to other users), cache effects (due to decreasing local problem size at scale) and the non-deterministic behaviour of MPI's collective/non-blocking communication routines is left to future work.

4.2.2 Validation

The graphs Figure 4.2 and Figure 4.3 compare observed and predicted execution times for HYDRA running for 200 cycles on a single level of LA Cascade. Results are included for two different PPN counts, to stress-test the different components of the model, the contribution of communication between MPI tasks and network contention to total execution time will be much lower for 1 PPN than for 12 PPN.

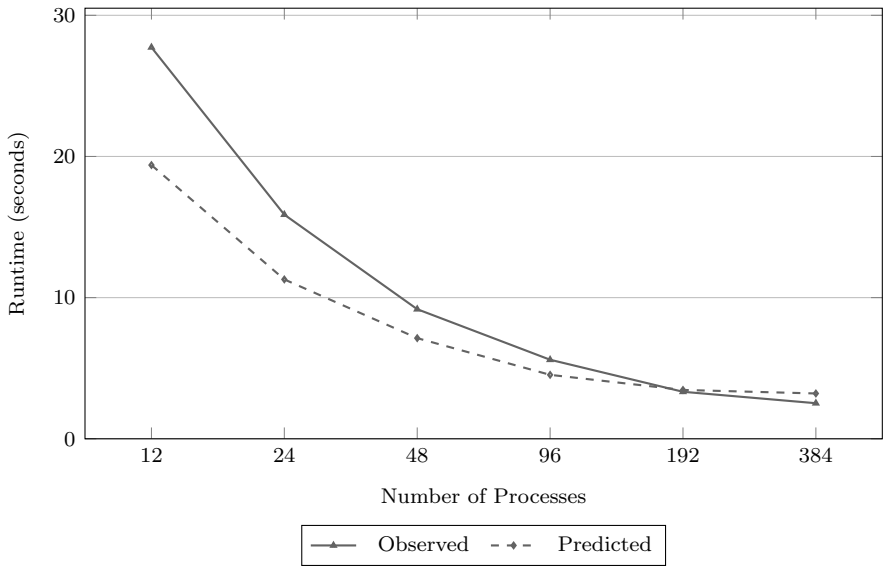


Figure 4.3: Comparison of recorded and predicted execution times for single-level runs of the original HYDRA using 12 PPN

	Multigrid				Single			
Nodes	**1PPN**		**12PPN**		**1PPN**		**12PPN**	
	Orig.	**Opt.**	**Orig.**	**Opt.**	**Orig.**	**Opt.**	**Orig.**	**Opt.**
1	6.48	59.34	3.15	1.44	2.00	2.77	3.83	5.31
2	40.80	49.97	6.98	3.30	4.58	6.34	8.79	12.18
4	44.81	35.32	3.14	0.84	1.17	1.62	2.24	3.11
8	14.47	12.61	3.03	0.58	0.80	1.11	1.54	2.13
16	6.76	5.62	0.53	1.52	2.11	2.92	4.04	5.60
32	2.94	2.40	5.48	5.80	8.04	11.14	15.44	21.40

Table 4.2: Confidence intervals for HYDRA's runtime on Minerva when using the LA Cascade dataset

The gap between observed and predicted times is significant, representing maximum model errors of 27.38% and 30.00% for 1 and 12 PPN respectively. The model's inaccuracy must therefore stem from a violation of the initial modelling assumptions; compute and communication may be failing to overlap, and/or synchronisation costs (i.e. the time processors spend waiting at barriers and blocking communication calls) may be higher than anticipated. This conclusion is drawn because the other individual model components (computation and communications) are accurate; single loop computation costs are predicted with high accuracy (96.78% on a single process), as are the individual message times predicted by the network model (the mean squared error for the intra-

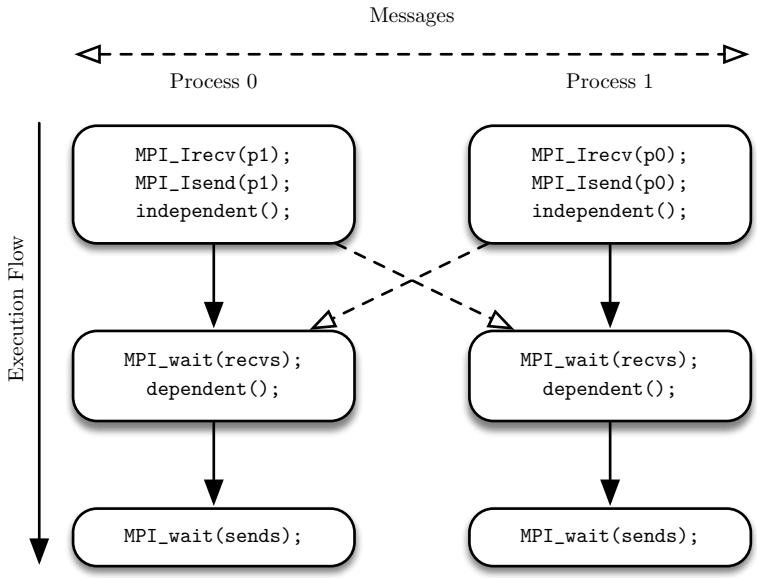


Figure 4.4: Best-case performance behaviours for the OPlus communication routines on two processors

and inter-node models are 1.04×10^{-11} and 4.78×10^{-9} respectively).

4.3 Model Analysis

4.3.1 Communication in OPlus

The best-case communication pattern for an OPlus loop (see Listing 3.2) on two processors (and the expected behaviour, if the modelling assumptions outlined during the model construction hold) is shown in Figure 4.4; the worst-case pattern (based on the most recent MPI standard [116]) is shown in Figure 4.5. The MPI standard does not say that a message must begin sending when `MPI_Isend` is called, and a successful return from the corresponding `MPI_Wait` means only that the send buffer can be re-used; therefore, it is possible for a message to be delayed until the sending processor can make no further progress. Consider the diagram in Figure 4.5: the left processor reaches its `MPI_Waitany` call before sending (but after receiving) its message; proceeds with its dependent compute,

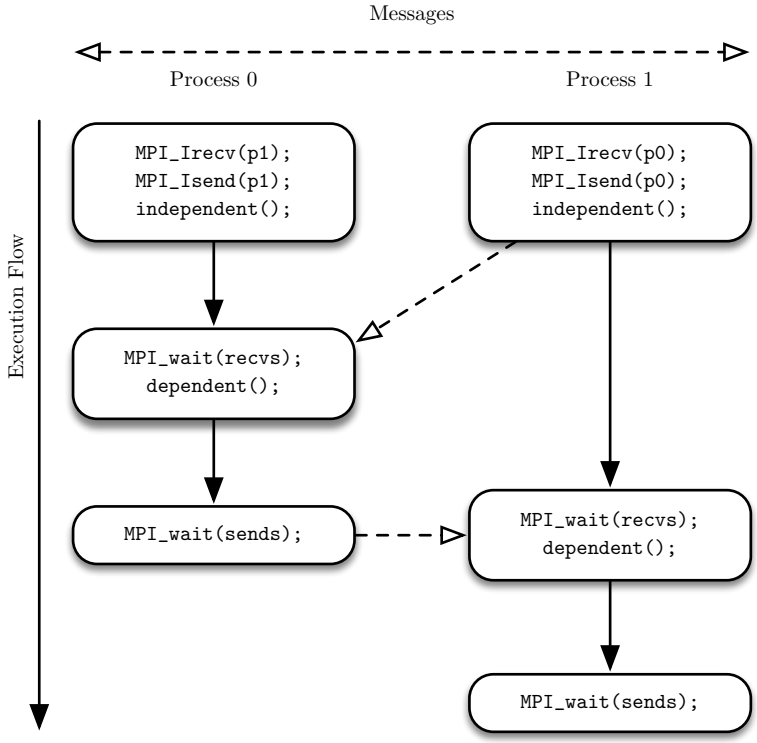
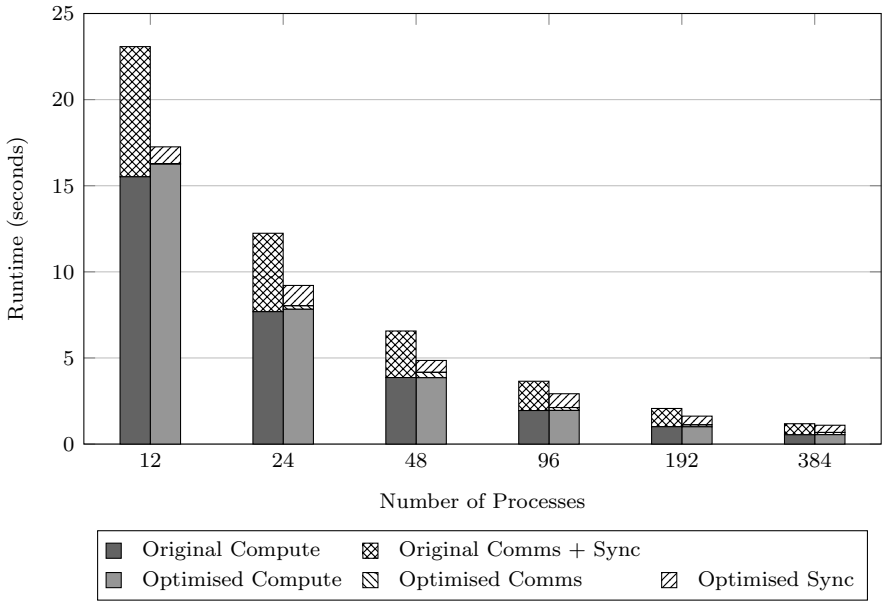


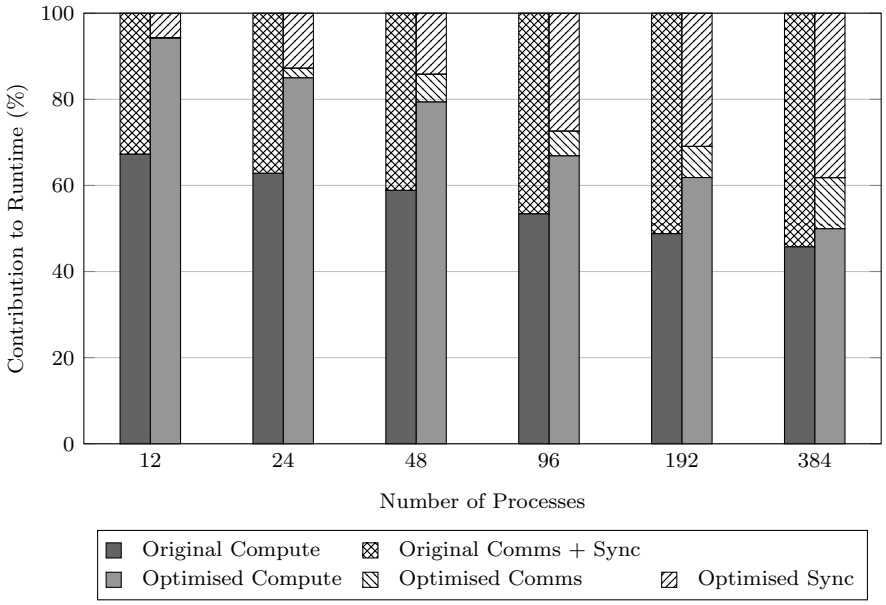
Figure 4.5: Worst-case performance behaviours for the OPlus communication routines on two processors

while the right processor remains blocked; and is only forced to send upon reaching the `MPI_Waitall` call at the beginning of its next parallel loop (which may take place after a significant amount of serial computation).

This worst-case behaviour is observed repeatedly in experiments (confirmed by MPI traces from SunStudio and manual investigation of timestamps), explaining the discrepancy between HYDRA's execution times and those predicted by the model. Specifically, due to the lack of synchronisation points between loops, it is theoretically possible for the compute costs of a single `op_par_loop` to be counted several times towards total execution time.



(a) Seconds



(b) Percent

Figure 4.6: Compute/communication/synchronisation breakdown for the original and optimised HYDRA, as: (a) time in seconds; and (b) percentage of execution time

4.3.2 Communication Optimisations

One way to fix this problem would be to ensure that non-blocking messages make progress when the application is not inside the MPI library [17]. However, this would likely have required a major overhaul to the codebases of both HYDRA and OPlus, which is beyond the scope of this work. Instead, some relatively simple alterations are made to the OPlus communication routines, to investigate the use of investing more time in improving the behaviour of non-blocking sends and receives. Specifically, these changes are as follows:

1. The non-blocking send (`MPI_Isend`) is replaced by a non-blocking synchronous send (`MPI_Issend`), thus ensuring a call to the matching `MPI_Wait` completes successfully "only if a matching receive is posted, and the receive operation has started to receive the message" [116].

2. An `MPI_Waitall` is introduced on outstanding send requests immediately before blocking on outstanding receive requests, thus helping processors to remain synchronised in keeping with the modelling assumptions.

Figure 4.6 presents a breakdown of HYDRA's execution time into compute, communications and synchronisation using 12 PPN. For the original code, the instrumentation is unable to separate communication and synchronisation costs (since the time spent in `MPI_Waitany` calls is reported); for the optimised code, the time spent in HYDRA's original `MPI_Waitany` calls are reported as communication, and time spent in the newly introduced `MPI_Waitall` as synchronisation. It is observed that the contribution of communication and synchronisation to HYDRA's execution time is decreased by the optimisations, in all configurations tested. However, the cost of synchronisation remains significant (as a fraction of execution time) even in the optimised code. A small fraction of this cost is attributed to the overhead of `MPI_Waitall`, but it is believed that most of this is caused by a combination of load imbalance between processors and communications that fail to overlap. Further investigation into this matter is necessary.

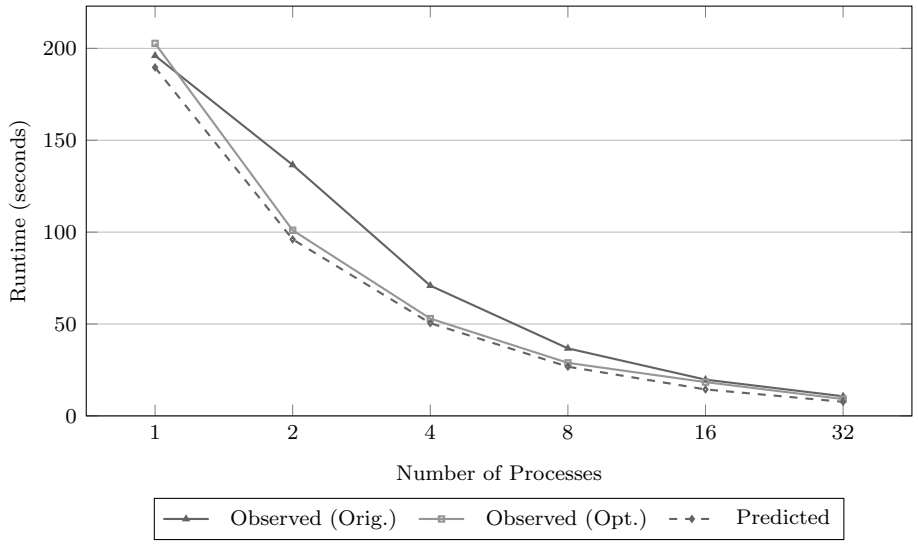


Figure 4.7: Comparison of observed and predicted execution times for single-level runs of the optimised HYDRA, using 1 PPN

Since the optimised version of HYDRA uses `MPI_Issend`, IMB was modified to use this non-blocking call instead of a regular blocking send and used to re-benchmark the network. With these new latency and bandwidth figures in place (as shown in Figure 4.7 and Figure 4.8), the performance improvements that are observed following optimisation are in keeping with those predicted by the model. Performance is improved by up to 1.45× in the case of single level runs, and is most noticeable for 12 PPN (since communication costs are higher for these runs).

It is acknowledged that the optimised version of the code may well be slower than the original at a sufficiently high core count. However, HYDRA is run at Rolls-Royce as a capacity workload; most jobs are scheduled on between one and four densely packed compute nodes, where the optimisation is shown to have the greatest effect. It believed that the results in this section adequately demonstrate the role that performance modelling can play in identifying unusual performance behaviours, and highlight that improving the communication behaviour of OPlus is clearly a critical direction for future research.

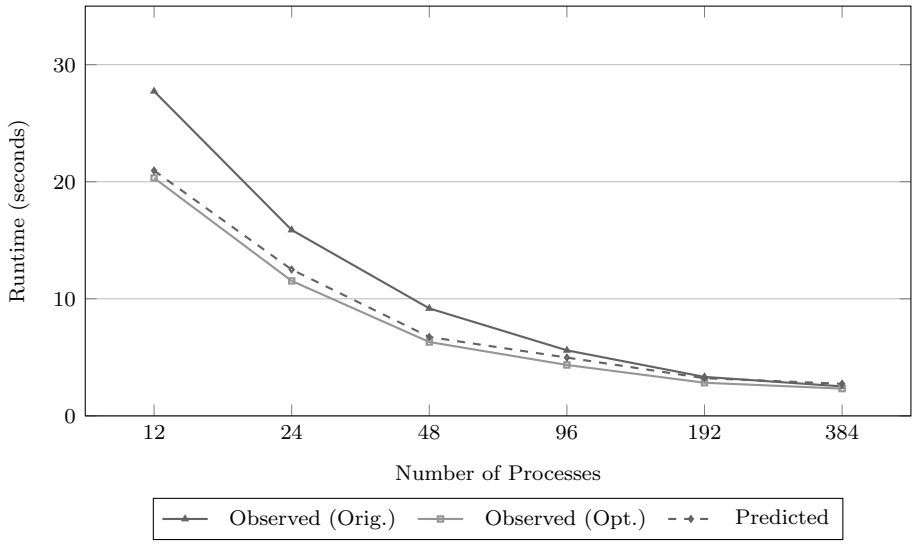


Figure 4.8: Comparison of observed and predicted execution times for single-level runs of the optimised HYDRA, using 12 PPN

4.4 Multigrid Model

4.4.1 Model Construction

In order to extend the single level model discussed in Section 4.2 to support multigrid execution, there are a number of terms that must be able to take different values for different levels of the multigrid. A description of these model terms is given in Table 4.3.

The simplest change to the model introduces separate grind-times per grid level (to account for changes in smooth behaviour, such as using first- or second-order smoothing), and separate N_i, N_h and N_e values per grid level (to account for each level having its own mesh geometry):

$$C_{l,L} = \alpha_{\text{intra}} + \beta_{\text{intra}} \times (N_{h,l,L} + N_{e,l,L}) \times B_l \tag{4.6}$$

$$\begin{aligned} W_{l,L} = &\max(N_{i,l,L} \times W_{g,l,L}, C_{l,L}) \\ &+ (N_{h,l,L} + N_{e,l,L}) \times W_{g,l,L} \end{aligned} \tag{4.7}$$

This change also affects Equation 4.3, which is updated to capture the wall-time

Term	Definition
	Dataset Parameters
n_{levels}	Number of levels in the multigrid.
n_{fmg}	Level below which first-order smoothing is used.
n_{pre}	Number of pre-smoothing iterations.
n_{post}	Number of post-smoothing iterations.
	Measured Parameters
$W_{g,l,L}$	
$N_{i,l,L}$	
$N_{h,l,L}$	Same as in Table 4.1, but specific to level L.
$N_{e,l,L}$	
$n_{\text{calls},l}$	
	Derived Parameters
R_{calls}	Number of additional calls caused by restrict.
P_{calls}	Number of additional calls caused by prolong.
$n_{\text{smooth},L}$	Number of smoothing iterations on level L per cycle.
$C_{l,L}$	
$W_{l,L}$	Same as in Table 4.1, but specific to level L.
$W_{\text{smooth},L}$	

Table 4.3: Description of additional model terms required to support multigrid HYDRA runs

per smooth iteration using a separate parameter for each level:

$$W_{\text{smooth,L}} = \sum_{l} n_{\text{calls},l,L} \times \max_{p \in P} (W_{l,L}) \tag{4.8}$$

The number of times that each level is smoothed during multigrid execution depends upon the cycle type in use, which in this case is the V-cycle (Figure 3.3(a)). Specifically, the finest level of the grid is always smoothed n_{pre} times; the coarsest level is always smoothed n_{crs} times; and intermediate levels are called n_{pre} times prior to restriction and n_{post} times prior to prolongation:

$$n_{\text{smooth},L} = \begin{cases} n_{\text{pre}} & L = 1 \\ n_{\text{pre}} + n_{\text{post}} & L < n_{\text{levels}} \\ n_{\text{crs}} & L = n_{\text{levels}} \end{cases} \tag{4.9}$$

As before, the first of HYDRA's cycles runs n_{start} smooth iterations before beginning the V-cycle. However, the level these start-up iterations operate on

Nodes	Multigrid			
	1PPN		12PPN	
	Orig.	Opt.	Orig.	Opt.
1	12.21	11.35	15.73	21.80
2	8.13	8.14	11.29	15.64
4	37.94	14.09	19.53	27.07
8	20.98	5.77	8.00	11.09
16	9.68	13.21	18.31	25.37
32	8.00	4.46	6.18	8.56

Table 4.4: Confidence intervals for HYDRA's runtime on Minerva when using the Rotor37 dataset

is now specified at runtime by a parameter (n_{fmg}). This change, together with the other equations detailed in this section, give rise to the updated form of Equation 4.5 seen below:

$$\begin{aligned} W_{\text{total}} =& C_r + \left(n_{\text{start}} \times W_{\text{smooth},n_{\text{fmg}}}\right) \\ &+ \sum_{L} \left[(n_{\text{cycles}} - 1) \times n_{\text{smooth},L} \times W_{\text{smooth},L}\right] \end{aligned} \tag{4.10}$$

The network model assumes no computation-communication overlap, as results from a comparison between the original version of HYDRA and a blocking implementation showed that this was not occurring in all cases.

The extension to multigrid also introduces restrict and prolong terms, as shown in Equation 4.11.

$$R_{\text{calls}} = P_{\text{calls}} = (n_{\text{levels}} - 1) \times (n_{\text{cycles}} - 1) \tag{4.11}$$

All other aspects of the model remain the same, and is it possible to collect all of the empirical benchmark information required for the predictive modelling of multigrid datasets from the execution of single-level benchmarks.

4.4.2 Validation

When reading the accuracy figures in this section, it is important to understand that the experiments were performed on a shared (and heavily contended) resource; validating a performance model on such a machine tends to increase

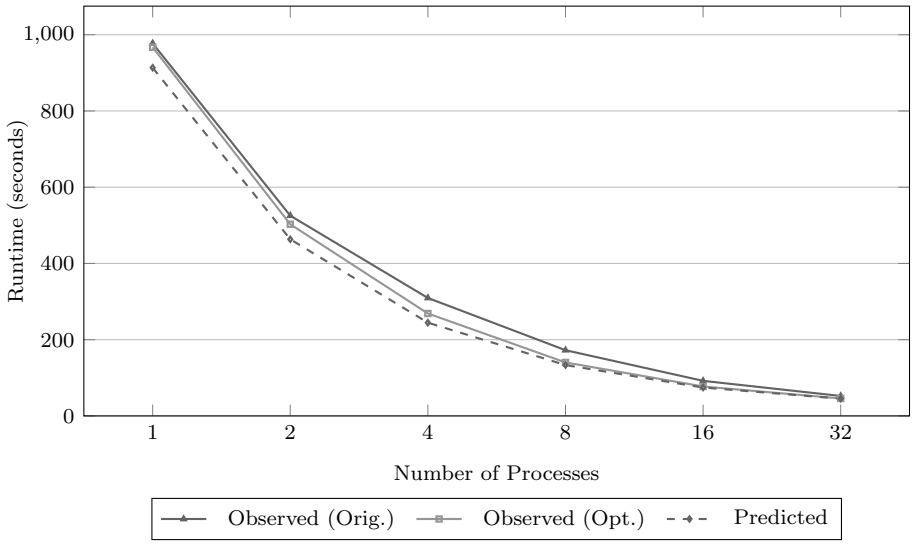


Figure 4.9: Comparison of observed and predicted execution times for multigrid runs of the optimised HYDRA using 1 PPN

error due to varying machine load. All experiments are repeated multiple times and the minimum, mean and maximum observed runtimes are reported. It is observed that HYDRA's runtime can vary by up to 22.42% from the mean, and that a distinct change of behaviour in model error occurs when moving from single to multiple packed nodes for both datasets (as this is the point at which the model switches from using solely intra-node communications).

A degree of inaccuracy in the model should also be expected due to other potential issues that are not captured by the model, such as: process placement on the host machine (e.g. a poor allocation by the scheduler will potentially impact not only execution time, but also the mesh partitioning) and the non-deterministic behaviour of MPI's non-blocking communications. Furthermore, it is noted that modelling a code's strong-scaling performance (i.e. the change in execution time occurring from an increase in the number of processors solving a problem of fixed size) is harder than modelling its weak-scaling performance (i.e. the change in execution time occurring from an increase in total problem size, for a fixed amount of work per processor).

Figure 4.9, Figure 4.10 and Table 4.5 present model validations for the op-

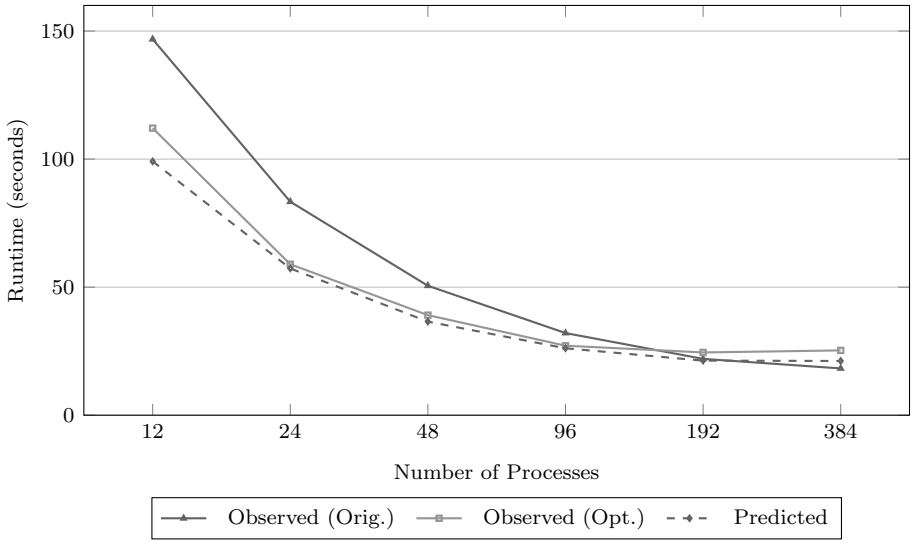


Figure 4.10: Comparison of observed and predicted execution times for multigrid runs of the optimised HYDRA using 12 PPN

Nodes	Original (s)			Optimised (s)			Pred. (s)	% Error Original			% Error Optimised		
	Min	Mean	Max	Min	Mean	Max		Min	Mean	Max	Min	Mean	Max
1 PPN													
1	975.05	977.29	981.66	920.63	967.18	981.18	913.51	-6.31	-6.53	-6.94	-0.77	-5.55	-6.90
2	499.97	525.34	541.60	463.74	502.82	514.73	463.38	-7.32	-11.80	-14.44	-0.08	-7.84	-9.98
4	286.25	309.44	331.98	240.98	268.65	277.02	244.58	-14.56	-20.96	-26.33	1.49	-8.96	-11.71
8	163.06	172.40	177.83	132.87	140.30	145.74	133.44	-18.17	-22.60	-24.96	0.43	-4.89	-8.44
16	89.25	92.01	96.15	74.45	77.19	80.18	74.46	-16.57	-19.07	-22.56	0.01	-3.54	-7.14
32	50.87	52.31	53.87	44.24	45.76	46.69	45.32	-10.92	-13.36	-15.87	2.44	-0.96	-2.94
12 PPN													
1	145.37	146.81	148.58	111.43	112.09	112.90	99.12	-31.81	-32.48	-33.29	-11.04	-11.57	-12.21
2	81.12	83.37	88.24	57.39	58.95	60.76	57.30	-29.36	-31.27	-35.06	-0.16	-2.80	-5.69
4	48.86	50.55	52.06	38.56	39.07	39.42	36.54	-25.22	-27.71	-29.81	-5.23	-6.47	-7.32
8	30.83	32.08	33.92	26.89	27.14	27.48	26.13	-15.24	-18.54	-22.96	-2.83	-3.72	-4.92
16	21.76	22.00	22.30	23.59	24.51	25.14	21.31	-2.07	-3.15	-4.44	-9.67	-13.08	-15.24
32	16.41	18.27	22.00	22.99	25.30	28.91	21.18	29.06	15.88	-3.74	-7.89	-16.30	-26.76

Table 4.5: Model validation for multigrid runs of HYDRA on the LA Cascade dataset (Confidence intervals are in Tables 4.2 and 4.4)

timised version of HYDRA, running a multigrid simulation (with four levels) using the LA Cascade dataset. When comparing Figure 4.10 to Figure 4.8, it is observed that multigrid execution times follow a very similar trend to, and are ≈4–6× larger than, those of the corresponding single-level benchmark. This trend indicates the potential for predicting the execution time of a multigrid run from benchmarks of a single level.

The amount by which the model over-predicts increases with the number of processors for 1 PPN runs. Since LA Cascade is a relatively small dataset, when running at scale (32 nodes) it is decomposed to the point at which the partitions fit into cache; this leads to a decrease in W_g values, and thus a violation of the original modelling assumptions. Even at scale, LA Cascade's runtime is still dominated by computation (7.63× larger than communication time), and the increasing model error thus stems primarily from the compute component of the model (the error of which increases from 0.69% to 3.85%).

It is noted that the model error for 12 PPN runs is also increasing with processor count. However, in this case the model is under-predicting. The compute model remains accurate for 12 PPN runs (13.42% error), and the number of messages (and bytes) accounted for by the network model is within 1.05% of those observed in HYDRA; therefore the increasing model error is attributed to the increasing synchronisation costs arising from running at scale (Figure 4.6). Accounting for such synchronisation costs in the model is clearly a direction for future research.

The point at which the optimisation is no longer valid is exposed in Table 4.5, and occurs when running HYDRA using LA Cascade on 192 cores. At this scale, the compute is such a small portion of total runtime (24.42%) that the potential benefit from keeping compute synchronised is outweighed by the cost of synchronisation. These results suggest that scale should be taken into account when selecting the communications strategy for HYDRA.

The model error for Rotor37 is larger than for LA Cascade, increasing from 0.87% to 7.79% for 1 PPN, and from 6.14% to 11.82% for 12 PPN. This increased

error is due to the communication component of the model. The compute error remains low (1.08% and 16.05% for 1 and 12 PPN runs respectively), highlighting this component's ability to adapt to different data sets which follow similar execution paths, whereas analysis of the message sizes used in the communication component highlights an imbalance in their distribution. To prepare the model for additional datasets, it must be extended to include all of HYDRA's loops and their influence on communication behaviour.

4.5 Summary

The chapter details an analytical performance model of Rolls-Royce HYDRA executing two unstructured multigrid datasets, LA Cascade and Rotor37, on up to 384 cores of an Intel X5650-based commodity cluster. The model has an average predictive accuracy of ≈90% for configurations matching those employed by Rolls-Royce.

Using the performance model as a guide of expected (and desired) runtime behaviour, a synchronisation issue in HYDRA's OPlus communication library was identified, which leads to a performance degradation of up to 29.22% on Minerva (see Table 3.4). An optimisation is proposed and implemented that improves HYDRA's strong-scalability significantly on this platform, bringing observed execution times in line with those predicted by the model. This optimisation was incorporated into a production version of HYDRA, which exhibited a ≈7% improvement in runtime when using datasets more representative of actual workloads (than LA Cascade or Rotor37) and on hardware in use by Rolls-Royce. While this is a smaller improvement in runtime than observed using LA Cascade or Rotor37, this speedup still has the potential to improve the throughput of HYDRA workloads at Rolls-Royce.

The results in this chapter therefore illustrate the role of performance modelling in the analysis and optimisation of legacy codes, and highlight the importance of revisiting such applications to assess whether their design assumptions

Nodes	Original (s)			Optimised (s)			Pred. (s)	% Error Original			% Error Optimised		
	Min	Mean	Max	Min	Mean	Max		Min	Mean	Max	Min	Mean	Max
1 PPN													
1	1367.00	1373.90	1379.46	1368.87	1375.11	1380.45	1298.27	-5.03	-5.50	-5.89	-5.16	-5.59	-5.95
2	737.27	742.03	745.57	697.78	702.40	706.09	657.55	-10.81	-11.38	-11.81	-5.77	-6.38	-6.87
4	400.26	411.81	438.98	363.99	373.34	378.37	339.25	-15.24	-17.62	-22.72	-6.80	-9.13	-10.34
8	226.23	235.70	247.64	197.11	199.28	203.00	180.89	-20.04	-23.25	-26.95	-8.23	-9.23	-10.89
16	135.43	139.80	145.31	112.11	117.01	125.59	101.96	-24.72	-27.07	-29.84	-9.06	-12.87	-18.82
32	74.83	79.73	82.99	65.16	67.23	69.71	57.53	-23.12	-27.84	-30.67	-11.71	-14.43	-17.47
12 PPN													
1	214.74	215.45	216.36	177.99	179.50	182.05	168.60	-21.49	-21.75	-22.07	-5.27	-6.07	-7.39
2	121.17	122.02	122.36	112.78	113.40	114.24	100.28	-17.24	-17.82	-18.04	-11.09	-11.57	-12.22
4	70.36	71.27	72.22	64.06	67.00	75.56	54.28	-22.85	-23.84	-24.84	-15.27	-18.99	-28.16
8	43.34	43.97	44.85	38.02	38.48	39.10	32.32	-25.44	-26.51	-27.95	-15.00	-16.02	-17.36
16	25.40	26.31	26.83	22.29	24.02	24.66	19.86	-21.80	-24.49	-25.96	-10.88	-17.30	-19.45
32	17.32	19.55	24.34	15.48	19.37	23.72	13.40	-22.63	-31.44	-44.95	-13.43	-30.83	-43.50

Table 4.6: Model validation for multigrid runs of HYDRA on the Rotor37 dataset

still hold on new software/hardware configurations. Also, it is shown that it is possible to reason about the performance of a geometric multigrid application based upon the execution of small, single-level benchmarks, and hence demonstrate a way to minimise the amount of benchmarking required when evaluating new machines.

In the next chapter, the model is i) further generalised to support multiple multigrid cycle types and Runge-Kutta invocations with a variable number of iterations; ii) extended to include additional costs, such as buffer pack/unpack times; and, iii) augmented with a mini-application representative of HYDRA's partitioning behaviour. In this way, the model will apply to a wider range of workloads and along with the mini-application, will be able to perform predictions without first needing to run an instrumented version of HYDRA at scale to collect partition sizes.

CHAPTER 5

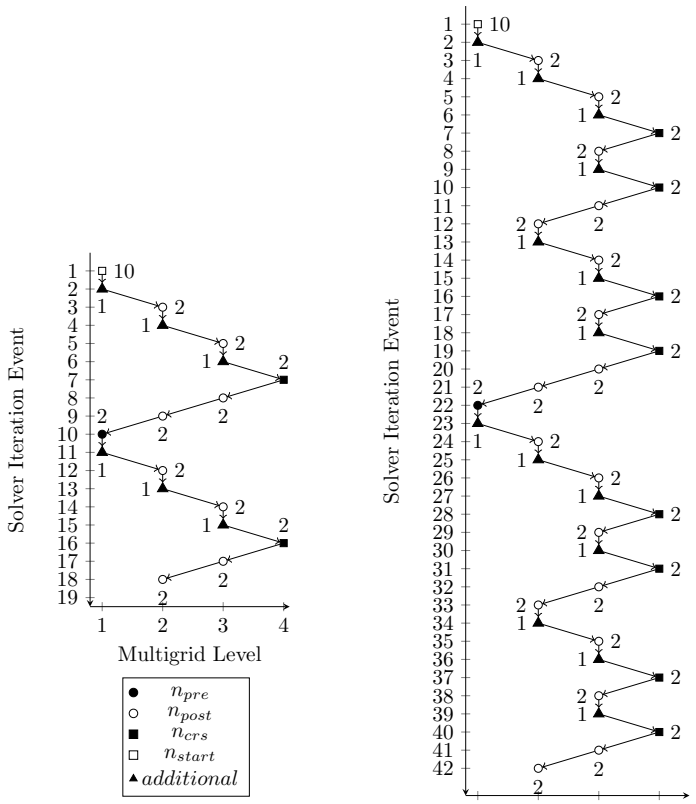
Enabling Model-led Evaluation of Partitioning Algorithms at Scale

While the performance model developed in the previous chapter was successful at identifying detrimental communication behaviour, limitations prevented the model from delivering runtime predictions across the desired range of input parameters and scale: i) the performance model was lacking complete analytical support, which restricted the set of tasks runtime predictions could be performed for (e.g. different multigrid cycle types); ii) the dataset coverage was limited due to the performance model only being primed from a subset of HYDRA's loops; and, iii) the model was reliant on partitioning data, which could only be collected empirically from HYDRA when running at scale.

First the inclusion of additional runtime costs to the performance model are validated on up to 1,024 cores of a Haswell-based cluster, using both a geometric partitioning algorithm and ParMETIS to partition the input deck, with a maximum absolute runtime error of 12.63% and 11.55% respectively. Second, the development of a mini-application representative of the mesh partitioning process internal to HYDRA is detailed. This mini-application is able to generate partitioning data that is usable with the performance model to produce predicted application runtimes within 7.31% of those produced using empirically collected data. Next, a demonstration of the performance model is given by performing a predictive comparison of several partitioning algorithms on up to 30,000 cores. The performance model is able to correctly predict the ineffectiveness of the geometric partitioning algorithm at 512 cores on the Rotor37 dataset.

Term	Parameter Definition
	Indicies
g	Grind time (loop time divided by total iterations)
p	Process identifier
l	Loop identifier
u	Unpack time
i	Independent elements
h	Dependent elements
e	Redundant compute elements
	Dataset Parameters
n_{levels}	Number of levels in the multigrid.
n_{cycles}	Number of multigrid cycles to perform.
n_{fmg}	Level below which first-order smoothing is used.
n_{pre}	Number of pre-smoothing iterations.
n_{post}	Number of post-smoothing iterations.
n_{rk}	Number of Runge-Kutta iterations.
n_{start}	Number of starting iterations.
n_{crs}	Number of smoothing iterations to perform at the coarsest level of the multigrid.
	Measured Parameters
$W_{g,p,l,L}$	Grind-time per level, per set element in loop
W_u	Grind-time for unpacking an element
$N_{i,p,l,L}$	Number of independent set elements in loop.
$N_{h,p,l,L}$	Number of dependent (halo) set elements in loop
$N_{e,p,l,L}$	Number of redundant (execute) set elements in loop.
	Derived Parameters
R_{calls}	Number of additional calls caused by restrict.
P_{calls}	Number of additional calls caused by prolong.
I_L^{post}	Calls caused by n_{post} input parameter on level L.
I_L^{pre}	Calls caused by n_{pre} input parameter on level L.
I_L^{crs}	Calls caused by n_{crs} input parameter on level L.
I_L^{start}	Calls caused by n_{start} input parameter on level L.
$W_{p,l,L}$	Walltime per process, per loop, per level.
W_{mg}	Total runtime of the multigrid solver.
$C_{l,L}$	Communication cost for loop l per level.

Table 5.1: Description of performance model terms from Chapter 4 and the additional terms required for multiple cycle types, pack/unpack costs and multiple compute regions



(a) V-Cycle (b) W-Cycle

Figure 5.1: Trace of solver iteration events ($n_{cycles} = 3$)

5.1 Runtime Model for Multigrid Applications

The generality and tractability of HYDRA's existing performance model is increased to overcome the aforementioned shortcomings. First the construction of further analytical equations are presented, which describe HYDRA's function invocations. Following this, it is shown how these new equations operate with the model developed in Chapter 4. Next, it is demonstrated how the new analytical equations can be adapted to allow for runtime predictions when using other types of multgrid cycle, thereby increasing the applicability of the model. The modelling terms used throughout this chapter are defined in Table 5.1; this table repeats parameters defined in Tables 4.1 and 4.3 for the readers convenience,

however it does not repeat the hardware parameters listed in Table 4.1.

5.1.1 Model of Solver Steps

HYDRA's smooth routine invokes a number of solver iterations. These depend on HYDRA's current position in the multigrid cycle (labeled in Figure 5.1) and directly affect the total number of solver steps. To parameterise the model, HYDRA's source code is examined from both the solver and Runge-Kutta loop bounds to the input deck, and in doing so the following parameters are identified as influencing the loop bounds: n_{crs}, n_{pre}, n_{post}, n_{cycles}.

To aid the development of equations for the number of solver iterations (per multigrid level) in terms of these parameters, a trace of solver iteration events and the multigrid level they originate from is collected. Figure 5.1(a) presents this trace, which is used as a guide for further code inspection, by creating a mapping between events in the trace and HYDRA's source code.

The first feature apparent from Figure 5.1(a), are the initial 11 iterations on the first level of the multigrid (solver iteration events 1 and 2). Through experimentation with the input deck and code inspection, it was found that the first 10 of these events can be attributed to the n_{start} parameter. The extra event is a separate feature, in which an additional iteration of the inner loop is performed only when restricting. This leads directly to Equation 5.1, where n_{start} is simply multiplied by the number of inner loop iterations (n_{rk}), and to this, a single addition iteration is added.

$$I_1^{start} = n_{start} \times n_{rk} + 1 \tag{5.1}$$

The second feature visible in Figure 5.1(a) is event 10. This feature is singled out next as it does not appear at the beginning of the previous V-cycle (solver iteration event 2). Code inspection reveals that these events are dictated by n_{pre}. Given the information that these events occur at the beginning of every V-Cycle, Equation 5.2 can be constructed. The second half of the equation

deals with the single additional iteration while restricting – both $\times 1$ terms, while unneeded, are left in to ensure a 1-to-1 mapping between the two halves of Equation 5.2 for readability.

$$\begin{aligned} I_1^{pre} = ((n_{cycles} - 2) \times n_{pre} \times n_{rk}) \\ +((n_{cycles} - 2) \times 1 \times 1) \end{aligned} \tag{5.2}$$

Next the events which occur on levels 2 and 3 of the V-cycle are examined, for both prolongation and restriction. Code inspection reveals that the number of iterations are dictated by n_{post} and as is the case with Equations 5.1 and 5.2, the additional iteration which occurs while prolonging must be accounted for.

$$\begin{aligned} I_{2,3}^{post} = (((n_{cycles} - 1) \times n_{post} \times 2) \times n_{rk}) \\ +(((n_{cycles} - 1) \times 1) \times 1) \end{aligned} \tag{5.3}$$

Finally, the events which occur on the final level of the multigrid are examined: events 7 and 16 in Figure 5.1(a). These occur once per V-cycle, therefore the equation is:

$$I_4^{crs} = (n_{cycles} - 1) \times n_{crs} \times n_{rk} \tag{5.4}$$

It should be noted that Equations 5.1-5.4, given an input deck, will predict the invocation count of `iflux`. The invocation count of the other functions will be dealt with by modelling their percentage of invocations relative to `iflux`.

5.1.2 Model Integration

Equations 5.1-5.4 are integrated bottom up, into the existing model to provide a fully analytical description of HYDRA's computation. The reader is referred to the previous chapter for the equations for communication time ($C_{l,L}$), restrict (R_{calls}) and prolong (P_{calls}) as these equations remain unchanged [23].

Equation 5.5 describes how the different types of compute (independent, halo and execute) and the communication are combined into a single walltime. To model communication-computation overlap, the larger of the independent compute and communication time is taken, and added to this, the compute which cannot be overlapped at all. This equation can easily be adjusted to produce a prediction where overlap is not assumed to occur, by replacing the maximum function with a summation.

$$\begin{aligned} W_{p,l,L} = &\max(N_{i,p,l,L} \times W_{g,l,L}, C_{l,L}) \\ &+ (N_{h,p,l,L} + N_{e,p,l,L}) \times W_{g,p,l,L} \end{aligned} \tag{5.5}$$

Finally, the runtime of all of the loops on each level of the multigrid are summed to give the predicted runtime for the solver (Equation 5.6).

$$W_{mg} = \sum_{l} \sum_{L} \max_{p \in P}(W_{p,l,L}) \times I_L \tag{5.6}$$

5.1.3 Generalisation to W-Cycles

V-Cycles are not the only pattern by which multigrid solvers can transition between levels. The process used in Section 5.1.1 to derive the equations which model a V-cycle is next applied to a W-Cycle – this lends weight to the processes applicability to arbitrary cycle types. As before, a trace of the code is plotted, but while performing W-cycles (Figure 5.1(b)).

The non-repeating features in Figure 5.1(b) (solver iteration event 1), and the frequency of steps caused by n_{pre}, are the same as for the V-cycle case, therefore Equations 5.1 and 5.2 can be reused. Next, the location where a single cycle terminates is identified (solver iteration event 21 in Figure 5.1(b) in order to construct equations for the remaining levels of the multigrid.

Equations 5.7 and 5.8 are similar to Equations 5.3 and 5.4 but are parameterised to allow operation with multiple cycle types.

$$I_{2,3}^{post} = (((n_{cycles} - 1) \times O_{2,3}^{post} \times n_{post}) \times n_{rk}) \\ +(((n_{cycles} - 1) \times O_{2,3}^{additional} \times 1) \times 1) \tag{5.7}$$

$$I_4^{crs} = (n_{cycles} - 1) \times O_4^{crs} \times n_{crs} \times n_{rk} \tag{5.8}$$

Where O_4^{crs}, O_2^{post}, O_3^{post}, $O_2^{additional}$ and $O_3^{additional}$ equal 4, 3, 6, 2 and 4 respectively for a W-Cycle and 1, 2, 2, 1 and 1 respectively for a V-Cycle. By making these improvements to the model it can support multiple cycle types (e.g. W-Cycle and V-Cycle) and a variable number of Runge-Kutta iterations. As an additional side effect the changes have increased the performance models tractability; the model's time to prediction has improved by ≈22× when predicting for runs at approximately 500 cores, and will likely improve the time to prediction at much larger scale.

5.2 Additional Performance Model Detail

First, three additional runtime costs are identified and included within the performance model: the compute and communication time for all 300+ loops in the code base, the time taken to pack and unpack data from the Message Passing Interface (MPI) library buffers at the application layer, and separate performance data for each region of compute. Second, these changes are validated over 1,024 cores by presenting the effect each adjustment has on the model's error. The performance model is further validated when using ParMETIS, rather than a geometric partitioning algorithm. Finally, the performance model's accuracy is reported over 1,008 cores when using data collected from an Ivybridge-based cluster (ARCHER).

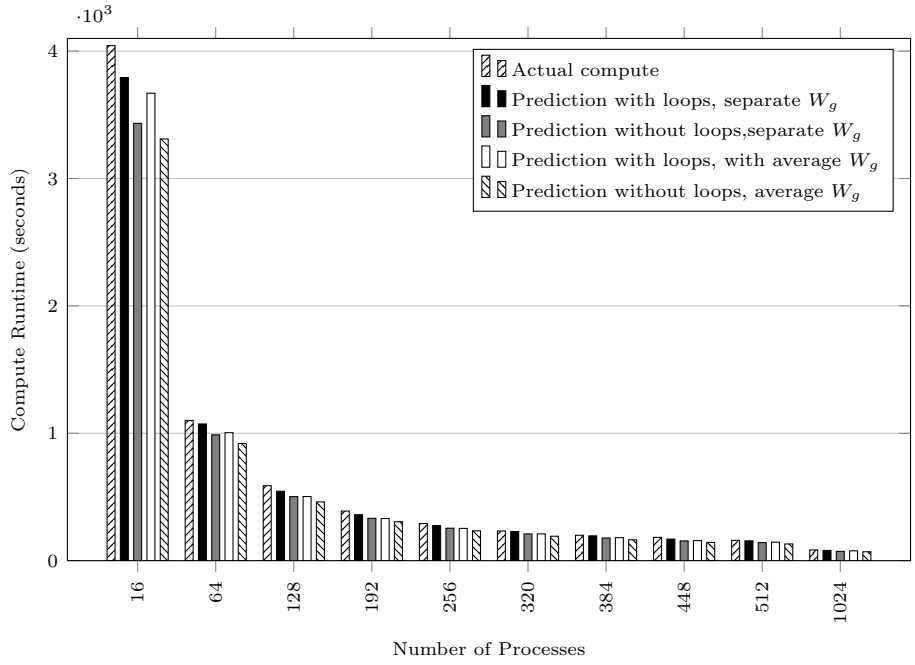


Figure 5.2: Comparison of actual and predicted compute time (Rotor37, 8 million nodes; geometric partitioning; Tinis)

5.2.1 Experimental Setup

For the model validations in this section and Section 5.3 the Rotor37 (8M) dataset (see Section 3.1.5) is used. Modelling data is collected from Tinis, a Haswell-based cluster and ARCHER, an Ivybridge-based cluster. More detailed specifications for each can be found in Table 3.3 and Table 3.5 respectively. However, all the analysis is performed using the data from Tinis.

The HYDRA experiments presented in this section and those remaining in this chapter use ParMETIS 3.1 and Recursive Inertial Bisection (RIB). Partitioning data is also collected from METIS 5.1.0, PT-Scotch 6.0.4 and Scotch 6.0.4, but is not used in conjunction with HYDRA/Oxford Parallel Library for Unstructured Solvers (OPlus). More information on partitioning algorithms can be found in Section 3.1.6.

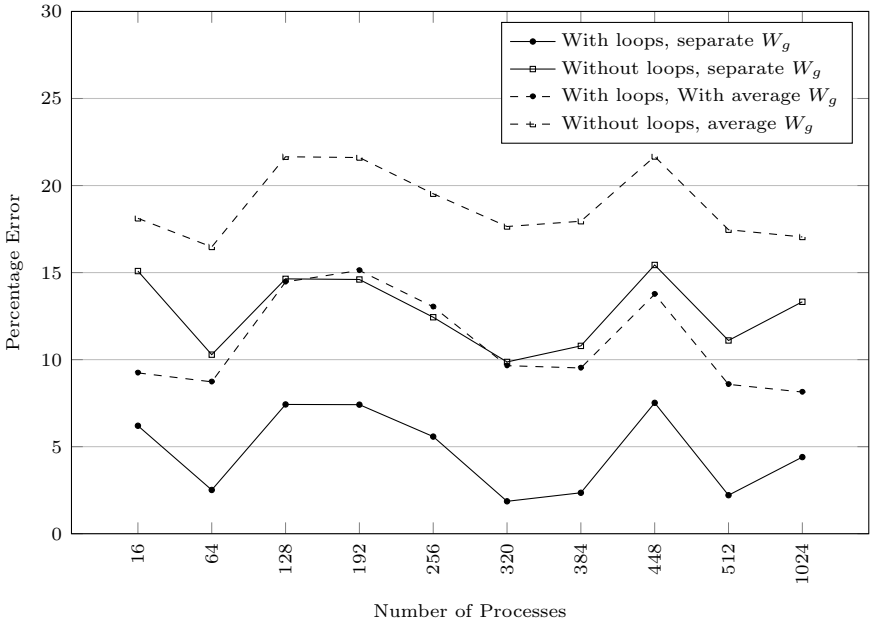


Figure 5.3: Predicted compute time percentage error (Rotor37, 8 million nodes; geometric partitioning; Tinis)

5.2.2 Region Grind-time Data

OPlus partitions HYDRA's compute into three different regions, independent, halo and execute. For each of these element types the access pattern varies, which is reflected in the timing information. The compute time (per element) for dependent elements compared to independent elements is 60.72% and 142.45% larger for `vflux` and `iflux` respectively. Without using separate timing information for the different regions a consistent under prediction in compute time is observed (average of 19.12%). However, when the performance model is primed with separate timing information for each region then the model error is reduced (see Figure 5.2) to a consistent average under prediction of 12.69%. Percentage errors at each process count are provided in Figure 5.3.

The analytical model is generalised to support these regional grind times by introducing three new terms: $W_{g,i,l,L}$, $W_{g,h,l,L}$ and $W_{g,e,l,L}$ for the independent, halo and execute regions respectively. After making this adjustment Equation 5.5 becomes Equation 5.9.

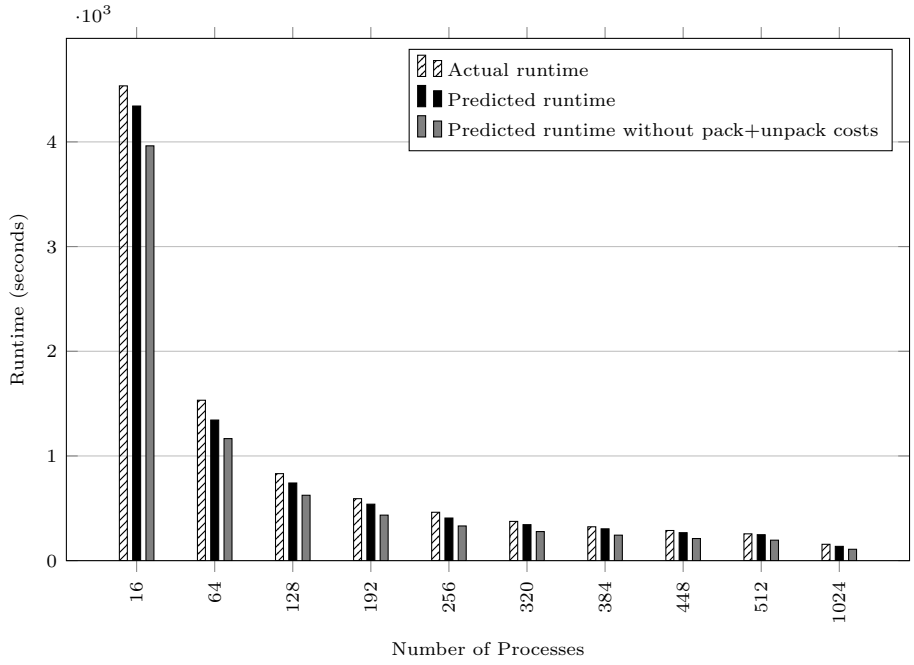


Figure 5.4: Comparison of actual and predicted runtime (Rotor37, 8 million nodes; geometric partitioning; Tinis). See Table 5.2 for confidence intervals.

$$\begin{aligned} W_{p,l,L} = &\max(N_{i,p,l,L} \times W_{g,i,l,L}, C_{l,L}) \\ &+ (N_{h,p,l,L} \times W_{g,h,l,L} + N_{e,p,l,L} \\ &\times W_{g,e,l,L}) \end{aligned} \tag{5.9}$$

5.2.3 Complete Loop Coverage

HYDRA consists of over 300 nested loops of which a subset are used by any given dataset; due to this large number of loops, using automated instrumentation tools is essential. Tools were developed in Section 3.3 to cope with the specifics of the code base which existing tools were unable to deal with (e.g. FORTRAN77 and nested loops). Naturally, full code coverage gives increased model accuracy because as it provides a complete view of HYDRA's performance. Also, it future proofs the performance model against new datasets which may exercise other regions of the code.

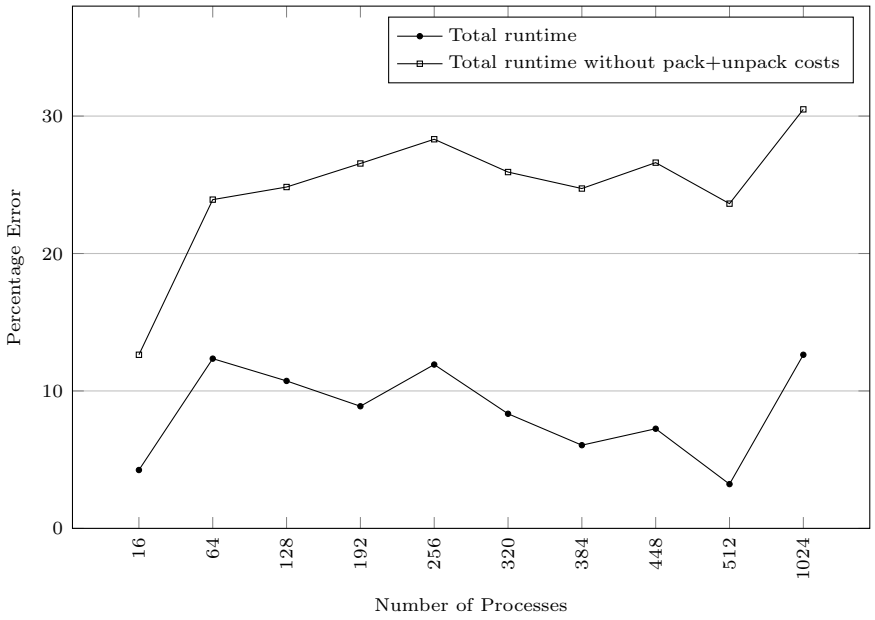


Figure 5.5: Total predicted time percentage error (Rotor38, 8 million nodes; geometric partitioning; Tinis).

With the addition of performance data from all loops in HYDRA the runtime performance model's under prediction reduces from an average of 12.69% to 4.79% (see Figure 5.2). Even with complete loop coverage there is still an error in compute time; it is suspected this is due to the modelling assumption that compute time per edges/node is the same across all processes which is not often the case. Both the complete loop coverage and the use of detailed compute data, reduces the compute error by approximately 10% at all measured scales (see Figure 5.3).

5.2.4 Buffer Pack Cost

OPlus abstracts communication between processes, and as part of this it must pack and unpack elements shared between processes into communication buffers. The cost of this operation largely depends on the number of items that need moving around, which in the previously used datasets of approximately ≈746 K nodes and ≈2.2 million edges (at the finest level) was a negligible amount. Fur-

Nodes	Geometric	ParMETIS
16	2.95	82.92
64	0.65	42.54
128	8.14	5.58
192	2.08	7.09
256	4.43	5.46
320	1.50	7.13
384	1.53	7.52
448	2.23	2.70
512	1.34	3.16
832	1.02	2.75
1024	0.77	2.05

Table 5.2: Confidence intervals for HYDRA's runtime on Tinis when using either a Geometric Partitioning or ParMETIS to partition the input deck.

ther validation work on the performance model with a larger dataset indicated parts of the code were not being modelled and subsequent performance analysis pointed to a significant buffer pack and unpack cost. With larger scale runs, increasingly larger datasets will be required (over 100 million nodes) therefore increasing this cost even further.

The analytical model is extended with Equation 5.10 to account for the buffer pack and unpack costs. Likewise, the unpack costs are accounted for on the receiving process.

$$W_{p,l,L} = N_{h,p,l,L} \times W_u + N_{e,p,l,L} \times W_u \tag{5.10}$$

Figure 5.4 shows that the runtime performance model's total under prediction is reduced from an average of 24.76% to 8.56% and at most 12.63% (see Figure 5.5 for per process errors).

The runtime performance model has also been validated with the aforementioned details on up to 1,008 cores of ARCHER, with a maximum model error of 4.72%. This provides further evidence to suggest the model is applicability across multiple generations of Intel hardware.

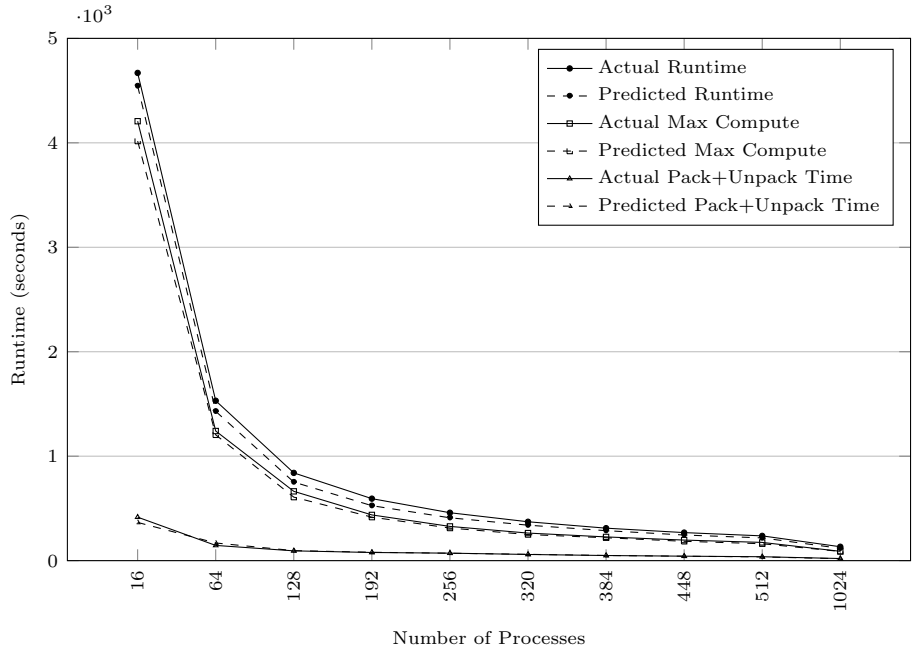


Figure 5.6: Comparison of HYDRA's actual and predicted runtime (Rotor37, 8 million nodes, ParMETIS; Tinis) See Table 5.2 for confidence intervals.

5.2.5 Performance Model Validation (ParMETIS)

Figure 5.6 presents the total runtime, max compute time and pack/unpack time for both predicted and actual executions when using ParMETIS as the partitioning algorithm. The errors for total runtime, max compute time and pack/unpack costs are on average 8.65%, 4.09% and 5.23% respectively.

The compute error consistently under predicts and the error is neither increasing or decreasing with scale, but fluctuates between under predictions of 8.37% and 1.43% (see Figure 5.7). This under prediction and fluctuation can be partially explained by a deviation from the assumption that the W_g values are similar across all processes for a given OPlus loop, multigrid level and compute region. This is not true as different processes have different access patterns, due to the nature of unstructured mesh codes.

This broken assumption manifests itself as a problem in the performance model when an average, maximum or minimum W_g is used to approximate the compute cost, as the model will always predict that the most expensive processes

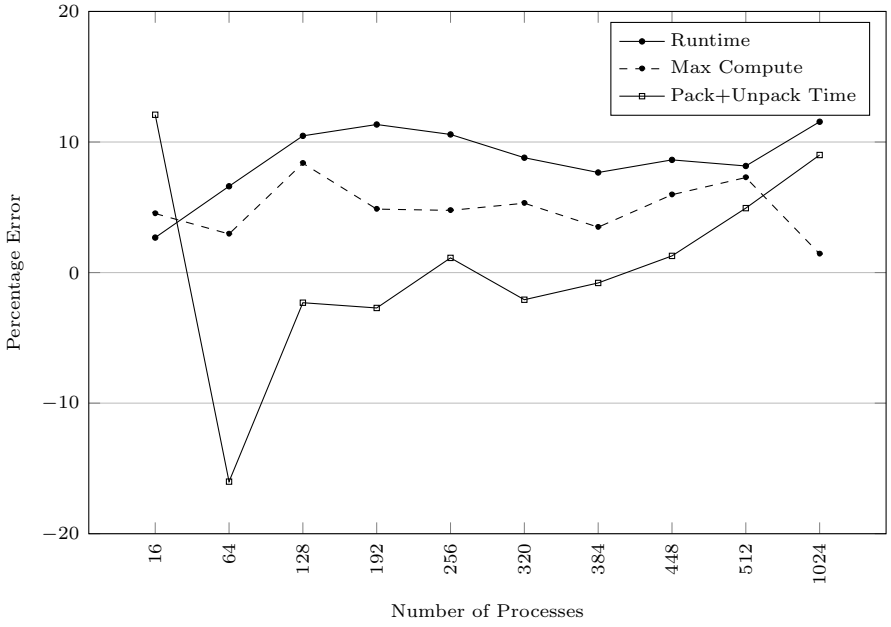


Figure 5.7: Percentage error for model costs (Rotor37, 8 million nodes, ParMETIS; Tinis) See Table 5.2 for confidence intervals.

is the one with the most elements to process. From Figure 5.8 it can seen that this assumption leads to an under prediction (except for at 320 processes) when using the average W_g and an over prediction when using the maximum W_g. For the predictions in this chapter an average over the top 50% largest W_g values is used as this is more representative of the compute costs on the critical path.

The pack and unpack error fluctuates between under predicting and over predicting. However, for the most part the absolute error is very low (less than 3 seconds for runs larger than 128 processes). Further investigation is required to identify the remaining sources of error, specifically at lower core counts, where the runtime prediction is over predicting 50.32 seconds and under predicting 23.32 seconds for runs with 16 and 64 processes respectively.

The compute under prediction leads to an under prediction of the total runtime as this is the dominant cost. Errors of between 2.68% and 11.55% are observed. This validation demonstrates the performance model's effectiveness at predicting runtime when using alternative partitioning algorithms.

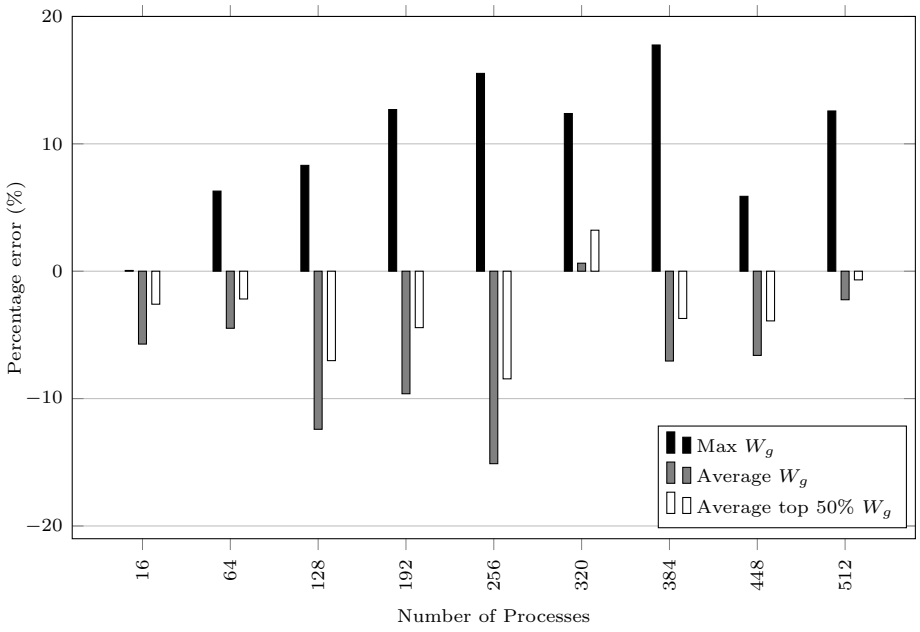


Figure 5.8: Percentage error of W_g calculation techniques for max edge compute (Tinis)

5.3 Set and Halo Size Generation

Typically in order for an analytical model to provide a runtime prediction, the size of the dataset (i.e. number of nodes, edges, cells) and message sizes must be known for a given process count. In the case of a structured mesh these sizes can be obtained using basic algebra, but for unstructured meshes these sizes depend on the partitioning algorithm (e.g. ParMETIS) and the halo exchange strategies.

Previously this data was collected empirically from HYDRA, but this approach becomes impractical for large process counts as vast amounts of hardware are required. This limits the performance model's capacity to predict HYDRA's scaling behaviour. Two applications are developed to solve this limitation: one for driving the partitioning algorithm, and one for computing the set and halo sizes (the latter will henceforth be referred to as "Moses"). With these applications, it is easier to explore different approaches (e.g. simulation, alternative partitioning algorithms, serialisation of code) to collecting partitioning informa-

tion for use in runtime predictions at scale.

5.3.1 Partitioning Mini-Driver and Mini-Application

First, a mini-driver (a framework for running specific application routines with test data) is developed, which exists to perform four tasks: i) read in the mesh files used by HYDRA for each level of the multigrid; ii) manipulate the mesh files into a form usable by the chosen partitioning algorithm; iii) invoke the partitioning algorithm; and, iv) store the resultant partitioning in a standard form, so the mini-application which is responsible for computing the halo and set sizes can operate with any chosen partitioning algorithm. This standard form is a set of tuples, which map nodes to the identifier of the partition they belong to.

To ensure the mini-driver's correctness, a comparison between the arguments to the chosen partitioning library (in this case ParMETIS) when called from HYDRA and the arguments used in the mini-driver is performed to ensure they are identical. Collecting usable partition data is only the start of the halo and set generation process; OPlus uses this data to partition the remaining sets and form the halos.

Second, a mini-application is developed which mimics the process by which OPlus uses the partitioning data to generate all other size data: the size of all sets in use by the CFD simulation (e.g. edges, nodes, faces); the number of elements which can be updated before and after communication; and, the size of the halos to communicate. To develop a fully representative version would be prohibitively time consuming, so a simplified version is built by adding only the major detail; this detail is selected based upon the largest contributors to runtime.

Using Moses and the mini-driver it is possible to generate partitioning information for up to 100,000 cores, which is usable by the runtime performance model.

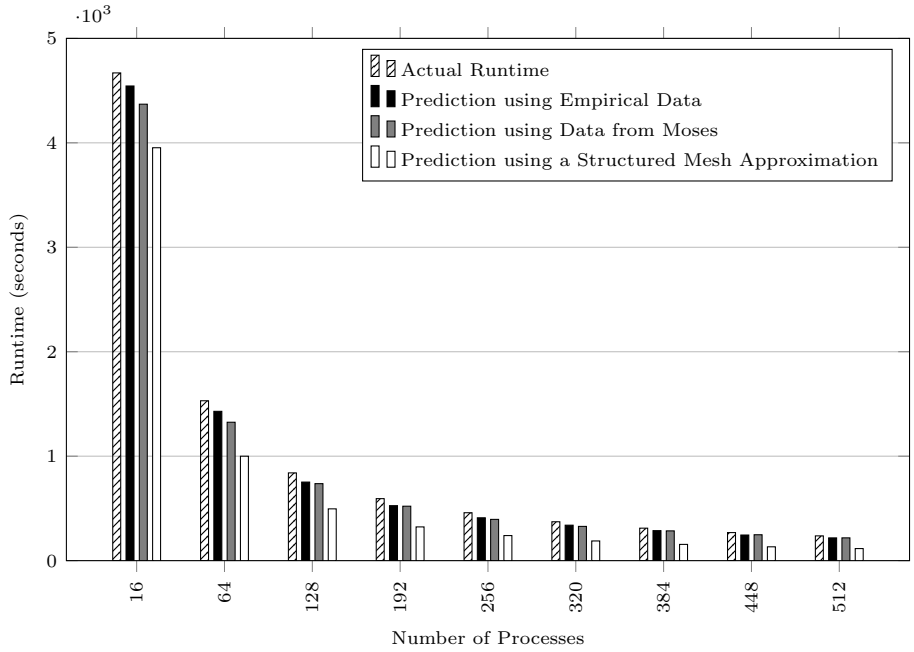


Figure 5.9: Impact of partitioning data source on model (Tinis)

5.3.2 Validation

First the level of detail included in the mini-application is quantified by performing a comparison between actual runtime, runtime predictions made using partitioning data generated by Moses and predictions made using partitioning data collected empirically from HYDRA (Figure 5.9). Additionally, a runtime prediction is plotted using a variant of the partitioning model used by Mathis *et al* [113], where the set and halo sizes are approximated to the structured mesh case. This plot is used as a baseline for runtime accuracy that can be achieved when using a simple partitioning model.

Figure 5.9 clearly highlights the large error (46.94%) in runtime that using the structured mesh approximation induces compared to using the empirically collected partitioning data. Whereas the runtime prediction made using the partitioning data generated by Moses differs by at most 7.31%. If the predicted parallel efficiencies are also examined when using both the structured mesh approximation and the data generated from Moses it is found that the former

indicates a near perfect efficiency across all ranks, but the latter is in line with the empirically predicted efficiency: an average parallel efficiency of 0.70 when using the empirically collected partition data and a parallel efficiency 0.69 when using the partitioning data generated by Moses. These results would indicate that using the data generated by Moses affords more runtime performance model accuracy than using the structured mesh data.

From the breakdown of predicted runtime costs (computation, communication, synchronisation, pack and unpack time) the two reasons why the structured mesh approximation fails to give an accurate prediction can be identified: i) it under predicts the amount of data to be sent between ranks, resulting in a lower predicted communication time and lower pack and unpack times; and ii) the lack of load imbalance reduces the cost of synchronisation on each process.

While the data generated using Moses is more representative than that from the structured mesh approximation, there are still sources of error. To identify these the set and halo sizes generated by HYDRA are compared directly with those generated by Moses. It is found that while the set sizes generated for the first level of the multigrid are of low error (0% for edges and ≈6% for nodes) this error increases up the multigrid to ≈24% and ≈35% for nodes and edges respectively. However, the upper levels of the multigrid account for a diminishing amount of the total runtime and therefore these errors have a smaller effect on predicted runtime error.

5.3.3 Predictive Analysis of Partitioning Algorithms

The use of the runtime performance model is demonstrated in conjunction with Moses to perform a predictive comparison of the effect varying partitioning algorithms have on HYDRA's runtime, for a given dataset (Rotor37) at varying scales (16-30,000 processes). Specifically, the trade off between load balancing the sets present in HYDRA (nodes and edges) and the amount of communication/pack/unpack costs is considered. The performance model is primed with compute data from a single scale (16 processes), as memory behaviours are not

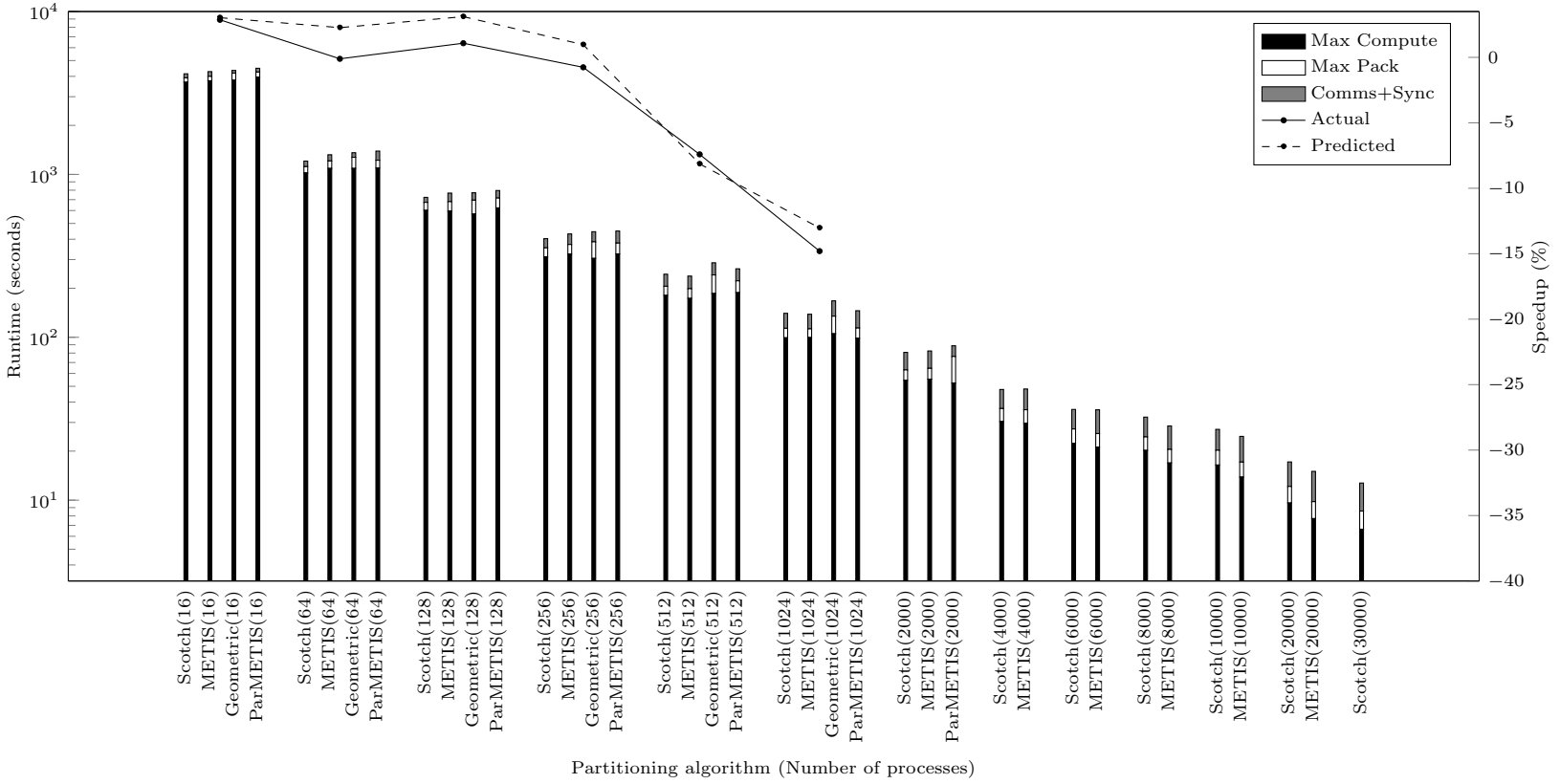


Figure 5.10: Predicted effect of partitioning algorithm on HYDRA's runtime and the speedup from using ParMETIS over a geometric partitioning

being considered.

Figure 5.10 contains this comparison of partitioning algorithms, however due to several current limitations is was not possible to collect the complete range (16-30,000) of data for all partitioning algorithms used in this work. It was only possible to collect data for up to 1,024 partitions for the geometric partitioning algorithm as currently there is no serial implementation of this algorithm with which to prime the mini-driver. Additionally, it was only possible to collect partitioning data from ParMETIS for up to 2,000 partitions as the simulations take in the order of weeks to complete. Finally, METIS is unable to partition the Rotor37 (8M) into 30,000 parts.

From Figure 5.10, it can be seen that the geometric partitioning algorithm is able to produce a partitioning with a comparable or lower predicted maximum compute when compared to the other partitioning algorithms, however it makes no consideration as to the communication time. This omission manifests itself primarily as increased time spent packing and unpacking elements for communication – 1.35× and 1.92× larger at 16 and 1,024 processes respectively when compared to ParMETIS (the next worst performing partitioning algorithm in terms of these costs). However, both the Scotch and METIS partitioning library manage the trade-off between costs as they take into account the number of edges cut, which is a proxy for communication time. This leads to predicted runtimes which are at most 1.2× better (predicted speedup of using METIS over a geometric partitioning at 1,024 cores).

Also from Figure 5.10, it can be seen that METIS consistently performs better than its parallel variant (ParMETIS) across all scales for both max compute time, and pack and unpack cost, this leads to a predicted performance improvement of up to 1.1× (at 512 cores). This performance improvement does not appear to diminish with scale.

This predictive analysis has delivered three observations, i) at the scales measured, Scotch and METIS are the better choice of partitioning algorithm when compared to ParMETIS and the geometric partitioning algorithm; ii)

the serial variant of ParMETIS produces consistently better partitions than ParMETIS itself; and, iii) the geometric partitioning invokes reduced performance in HYDRA runs of greater than 512 processes due to increasing buffer pack and unpack costs. These predictive observations will direct the focus of future work such as integrating the feature to read in pre-generated partitions into HYDRA. Especially those from Scotch and PT-Scotch, the former to see if the performance improvements at small scale hold and the later to determine if PT-Scotch is out performed by its serial variant.

Observation iii) is validated in Figure 5.10 by plotting the predicted and actual percentage runtime slow down of HYDRA when using the geometric partitioning algorithm over ParMETIS to partition the input deck (Rotor37). As can be seen from the figure, the runtime performance model in conjunction with Moses accurately predicts the downfall of the geometric partitioning algorithm at 512 processes. Indicating that this partitioning algorithm is not suitable for anything but small scale runs when using the Rotor37 dataset.

5.4 Summary

The development of a general analytical model for a multigrid code which supports multiple cycle types, a variable number of Runge-Kutta iterations and an arbitrary number of loops has been presented. These additions have increased the number of input decks the performance model is applicable to.

These additional details have been validated on up to 1,024 cores of a Haswell-based cluster, using both ParMETIS and a geometric partitioning algorithm to partition the input deck, with a maximum absolute error of 12.63% and 11.55% respectively.

The development of Moses has been presented – a domain decomposition mini-application, which is able to convert partitioning data from multiple algorithms (Scotch, METIS, ParMETIS) at varying scale (up to 30,000 cores) into data usable by the runtime performance model. It is shown that runtime pre-

dictions made using this data have a runtime error of at most 7.31% over 512 processes, when compared against predictions made with empirically collected partitioning data.

Finally, a demonstration of Moses is given; it is used in conjunction with the runtime performance model by comparing the effect of several different partitioning algorithms on HYDRA's runtime. From this analysis it is concluded that priming HYDRA with partitioning data from Scotch is worth investigating due its consistent predicted performance advantage (maximum of 1.21×) over ParMETIS. Additionally, the prediction that the geometric partitioning algorithm causes reduced performance in HYDRA at 512 processes when compared with ParMETIS is validated.

CHAPTER 6

Developing Mini-HYDRA

The ongoing and rapid cycle of hardware and software development in the High Performance Computing (HPC) space means there are always new performance opportunities to evaluate for particular classes of application. This evaluation is often a time consuming process due to the complexity of the applications involved, and the learning curve that often comes with using new architectures and tool chains. To ease this process application custodians have looked to alternative techniques to ease this burden. As has been demonstrated by previous work and the work in Chapters 4 and 5 performance modelling is one such example, which allows the reasoning about scaling behaviour.

However, this performance model is weak at reasoning about low level performance behaviours. Another relatively new device which allows for reasoning about low level performance is the mini-application – a manageable, but performance representative application which can be used to conduct rapid performance analysis of a much larger application. In this chapter a mini-application is developed for unstructured mesh, geometric multigrid codes, its scaling behaviour is validated and then its use is demonstrated by assessing the performance impact of incremental features to hardware.

6.1 Developing Mini- and Compact-HYDRA

6.1.1 Mini-HYDRA

Even though the benefits of mini-applications are clear (see Section 2.5), their development is not a well-defined process as it depends largely on their intended purpose [117]. This makes their development challenging as the purpose may

differ on a project-by-project basis, limiting the reuse of efforts. However, the literature details a list of considerations and guidelines; these are aggregated by Messer *et al* and summarised here as a set of questions for reference [117].

1. Where does the application spend most of its execution time?
2. What performance characteristics will the mini-application capture?
3. Can any part of the development process be automated?
4. How can the build system be made as simple as possible?

When these questions are answered the mini-application's developer will have a concrete understanding as to i) which aspects of the parent code the mini-application should include, and ii) the components of the supporting configuration (e.g. tools and datasets). These guidelines are applied to the development of mini-HYDRA and because the development of each mini-application is essentially unique, it is considered a valuable exercise to document the usefulness of this approach. Additionally a new consideration is proposed, which comes from the experiences developing mini-HYDRA.

An answer to question 1 is provided in Section 3.4 – the most time consuming regions of code are contained within the routines `vflux` and `iflux`. This decomposition in time reveals the routines that should be captured within the mini-application. This decomposition in time is not always necessary to highlight the regions of the parent application that should be focused on as the mini-application developer may already have a particular performance characteristic in mind that they wish to study which may not dominate runtime.

Next question 2 is answered by considering the primary purpose of mini-HYDRA – to evaluate the impact of new hardware features based on their suitability for applications such as HYDRA. This use case suggests constructing a mini-application which ignores I/O and inter-node communication performance costs and focuses on computation costs. This question guides the decomposition

Listing 6.1: Pseudo-code skeleton of the mini-application

```
1  |
2  |int updown = 0;
3  |while(i < cycle) {
4  |     // Runge-Kutta iterations
5  |     for(int j = 0; j < RK; j++) {
6  |          compute_flux_edge();
7  |          compute_boundary_flux_edge();
8  |          compute_wall_flux_edge();
9  |          update();
10 |          time_step();
11 |     }
12 |
13 |     // Multigrid logic. Move restrict or prolong?
14 |     if(updown == 0) {
15 |          level++;
16 |          restrict();
17 |
18 |          if(level == levels-1) {
19 |               updown = 1;
20 |          }
21 |     }
22 |     else {
23 |          level--;
24 |          prolong();
25 |
26 |          if(level == 0) {
27 |               updown = 0;
28 |               i++;
29 |          }
30 |     }
31 |}
```

of the code in terms of the type of performance cost rather than in time, and shifts the focus on to more specific regions of the code.

Now a new consideration is proposed: which aspects of the simulation (e.g. unstructured mesh, finite volume, multigrid) contribute to the compute behaviour within the most expensive regions of the code? This decomposition by simulation aspect provides a route for including performance characteristics within the mini-application. It is the irregular memory accesses which contribute greatly to the difficulty of running on different compute architectures. These irregular memory accesses come from two main sources: the edge updates over the unstructured dataset and the restriction and prolongation of corrections between the multigrid levels (see Section 3.1.3).

With these features in mind, mini-HYDRA is based upon an existing code developed by Corrigan *et al* as it shares simulation features with HYDRA [34].

Listing 6.2: Pseudo-code of the edge loop

```
1  |
2  |for edge e in edges {
3  |    //Acquire the indicies of the nodes at
4  |    //each end of this edge
5  |    a = e.left_index, b = e.right_index;
6  |
7  |    //Extract the node properties.
8  |    //Note that a and b can be arbitrary
9  |    density_a = variables[a*NVAR + VAR_DENSITY];
10 |    density_b = variables[b*NVAR + VAR_DENSITY];
11 |
12 |    //Perform flux calculation
13 |    ...
14 |
15 |    //Update the fluxes
16 |    ...
17 |}
```

However, as described in Section 2.5 HYDRA has several additional simulation features, which need adding to this existing code. It should be noted that the correctness of the simulation is not verified against a standard problem, as the interest is purely on the performance characteristics which are validated in Section 6.2. Furthermore there are no restrictions on where this code can be run, meaning it can be run on hardware without being subject to a lengthy approval processes first.

A skeleton for the resultant mini-application is provided in Listing 6.1 with a focus on presenting an overview of the simulation's properties. The outer loop controls, not the number of iterations, but the number of complete multigrid cycles performed. The nested loop performs the Runge-Kutta iterations, where the core of the simulation is performed. The routines of interest are the `*_edge` routines as they contain the vast majority of the memory accesses. Past the Runge-Kutta loop is the multigrid logic, which in this case hard codes a V-cycle.

In the `*_edge` routines the application originally performed computation over nodes, i.e. each node is taken in turn and updated based on the properties of it's neighbours. While this is an irregular access pattern it can make vectorisation challenging for several reasons (e.g. the variable number of nested loop iterations). This was modified to computation over edges (as this is used by the

parent code), which is demonstrated in Listing 6.2. This computation takes each edge in turn then accesses the node at each end. This arrangement removes the need for an inner loop, and introduces it's own barriers to exposing parallelism, but the key is for the challenges to be the same as for the parent code.

Support for the computational behaviours of multigrid were implemented by augmenting the construction of the Euler solver presented by Corrigan *et al* with crude operators to transfer the state of the simulation between the levels of the multigrid (see Listing 6.1). These operators are defined by Equations 6.1 and 6.2 which serve as restriction (fine to coarse grid) and prolongation (coarse to fine grid) operators respectively [16]. Where u_j^l represents simulation property u of the j^{th} node at level l and N_j^l is the set of node indices which are linked to node j at level l from $l-1$ of the grid.

$$u_j^l = \frac{\sum_{i \in N_j^l} u_i^{l-1}}{|N_j^l|} \tag{6.1}$$

$$u_{i \in N_j^l}^{l-1} = u_j^l \tag{6.2}$$

The restriction operator (Equation 6.1) primes the simulation properties with an average across nodes from the finer grid level – this mapping is defined as part of the input deck. The prolongation operator (Equation 6.2) reverses restriction by injecting the values from the coarse grid to the fine grid as dictated by the mapping.

Support for an arbitrary number of neighbours was added by modifying the bounds of the flux summation (Equation 4 in the work by Corrigan *et al*) to accommodate a variable number of components. This summation is already weighted by the surface area of the interface between nodes in the mesh, so it is not necessary to correct for additional interfaces.

6.1.2 Compact-HYDRA

To complement mini-HYDRA, compact-HYDRA is constructed. In contrast to the mini-application, contains actual kernels from HYDRA and is therefore closely representative of performance. Of course this means that compact-HYDRA is subject to certain restricts on where it can be run, but it can still be used for i) in house porting activities, and ii) as a vehicle for validating the performance characteristics of mini-HYDRA which is demonstrated in Sections 6.2 and 6.3.

Compact-HYDRA is built by manually separating the flux calculations in `iflux` from any memory allocation and usage of Cray Pointers, then an automatic FORTRAN to C conversion is applied to generate a version of this kernel in C. This reduces the chance of introducing errors during the conversion process. Next the mini-application is modified to invoke this translated kernel, as this way both applications can use a common framework for reading datasets and check-pointing. Essentially both compact- and mini-HYDRA can be primed using the same datasets; this keeps the underlying cause of irregular memory access patterns the same which aids in the comparison between the applications.

6.1.3 Supporting Tools

Part of what makes a mini-application a useful tool is its simplicity, this however can not just be restricted to the application itself and must apply to the processes surrounding the mini-application and parent application that take time (e.g. building, job submission).

The make process is simplified by removing all third-party libraries such as Hierarchical Data Format 5 (HDF5) and the communications library. These can both be safely removed as the purpose of this mini-application is not to investigate Input/Output (I/O) performance, inter-node communication performance or the overheads introduced by library abstractions. Removing these dependencies allows the application to be built in seconds with little or no intervention

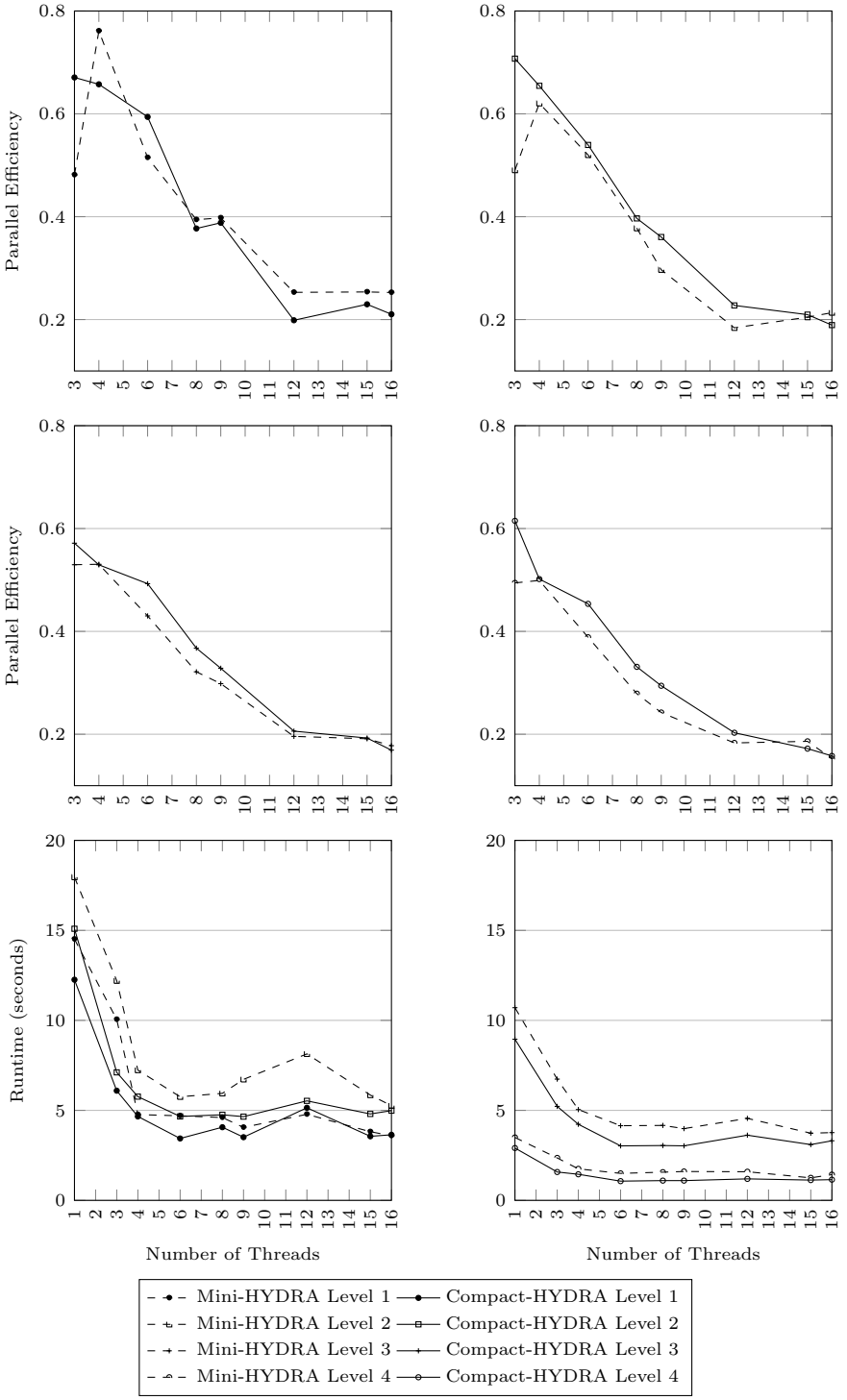


Figure 6.1: Comparison between the runtime and parallel efficiency of Compact- and Mini-HYDRA for each level of the multigrid

Nodes	Compact HYDRA				Mini-HYDRA			
	with FMA		without FMA		with FMA		without FMA	
	SIMD	no SIMD	SIMD	no SIMD	SIMD	no SIMD	SIMD	no SIMD
1	0.71	0.19	0.05	0.14	0.08	0.03	0.20	0.10
4	0.04	0.02	0.05	16.04	0.04	0.12	0.04	0.07
8	16.11	15.98	16.00	0.04	0.02	0.07	0.23	0.19
12	0.06	0.06	0.06	0.09	0.04	0.05	0.10	0.09
16	0.05	0.04	0.02	0.07	0.20	0.02	0.15	0.06

Table 6.1: Confidence intervals for Compact- and Mini-HYDRA's runtime on Tinis

past adjusting the compiler and its flags in the makefile. Another time delay in benchmarking is the creation of job submissions scripts so examples of these scripts are included for several common schedulers: SLURM, LSF and Moab.

Utilities have been included to validate the final state of the simulation after changes to the configuration (e.g. compiler flags, code optimisations, porting to accelerators) of the code. Additionally, tools are included to extract the geometries from the datasets used to prime HYDRA and transform them into a form which is understood by the mini-application. This is done to reduce the number of factors which could cause differences in runtime behaviour between HYDRA and its mini-application.

6.2 Mini-HYDRA Validation

The mini-application is validated using two existing methods. First a comparison is performed between the OpenMP parallel efficiency of both the compact- and mini-application for all levels of the multigrid [117] (see Figure 6.1). Obviously a correlation between scaling behaviour does not imply the underlying causes of the observed behaviour are the same. For this reason, the comparison is further strengthened using a second approach, VERITAS, developed by Tramm *et al*, which involves comparing the correlation of parallel efficiency loss to performance counters for both the mini-application and the parent code [156] (see Figure 6.2). These comparisons highlight differences and similarities between the two applications which will address in turn.

Figure 6.2: Comparison between Mini- and Compact-HYDRA in terms of the correlation of PAPI counters with parallel inefficiency

6.2.1 Experimental Setup

In this section validations of the mini-application against parent code behaviours are performed on a dual socket, 16 core Haswell node (see Table 3.5). Both applications were primed using the geometry from the LA Cascade dataset.

In Section 6.3 experiments are performed on both the aforementioned Haswell machine and additionally a dual socket, 24 core Ivybridge machine (see Table 3.1). All experiments are performed using both the mini-application and a derivative of the parent code using the LA Casacade dataset.

The confidence intervals for the runtime of both compact- and mini-HYDRA on both Tinis and Napier, using the LA Cascade dataset and varying compiler flags: with and without Fused Multiply-Add (FMA) (where available) and with and without auto-vectorisation can be found in Table 6.1 and Table 6.2.

6.2.2 Validation

By visual inspection of the first four charts in Figure 6.1 it can be seen that the parallel efficiency of both compact- and mini-HYDRA are similar, except when scaling between one and three processes. The parallel efficiency for mini-HYDRA is lower than that of compact-HYDRA for this transition; it is suspected that this is caused by the mini-application enjoying the increased cache bandwidth for single core workloads supplied by the Haswell platform [71] more than compact-HYDRA due to its higher number (2-3×) of level two cache reads. This suspicion is evidenced by the fact that this drop in parallel efficiency does not appear in results collected from the Ivybridge platform, which does not have increased cache bandwidth for single core workloads (see Figure 6.3). This difference in level two data cache reads manifests itself in Figure 6.2 with the correlation of the `PAPI_L2_DCR` counter not matching for both applications (and consequently `PAPI_L2_TCR`): a positive correlation with parallel inefficiency is observed for compact-HYDRA rather than a slight negative correlation. It is important to be aware of this difference when attempting to relate the mini-

HYDRA's performance back to compact-HYDRA, as the former will clearly stress the level two cache bandwidth more. Apart from this difference in level two cache reads, all other compact-HYDRA counters (with correlations above 0.8) exhibit a similar correlation with parallel inefficiency as mini-HYDRA.

Another performance feature that is visible in Figure 6.1 is that the runtime of the second level of the multigrid is comparable to that of the first, even though it contains 63% the number of edges. This is due to the nature of the multigrid solver, in that the solver is called twice as frequently on the second level as it is the first. This observation is consistent across both mini-HYDRA, compact-HYDRA and HYDRA's `iflux` kernel from version 7.3.4 (see Figure 3.7 for HYDRA runtimes).

6.3 Impact of Intel Haswell on mini-HYDRA

The introduction of the Intel Haswell architecture brought with it several new features over its predecessor, the Ivybridge platform, such as increased cache bandwidth for single core work loads, AVX2 and the addition of the fused-multiply class (FM-*) of instructions [71]. The mini-application is used to examine the impact that these features have on unstructured mesh, geometric multigrid codes.

The plots in Figure 6.3 present the scaling behaviour of compact- and mini-HYDRA in terms of both runtime, and parallel efficiency for only the Ivybridge platform as a per level breakdown of scaling behaviour on Haswell is already provided in Figure 6.1. These plots only include the results with auto-vectorisation turned-off as using auto-vectorisation has a negative impact on performance for this kernel. Figure 6.3 indicates that as the number of cores in use for a node reaches the maximum, the runtime of the mini-application on both platforms approach a similar value: a 1.16× speedup (≈1.5 seconds) when using Ivybridge over Haswell. However, the Haswell chip has a lower TDP so there is the potential to to achieve the same level performance with a lower overall

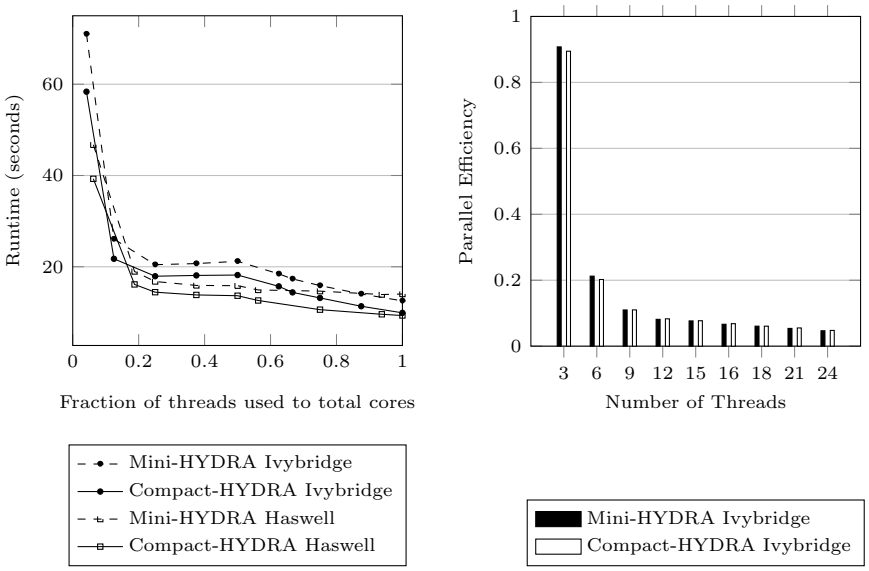


Figure 6.3: Comparison between OpenMP strong scaling behaviour between Intel Ivybridge and Intel Haswell for both Compact- and Mini-HYDRA

	Compact-HYDRA		**Mini-HYDRA**	
Nodes	**SIMD**	**no SIMD**	**SIMD**	**no SIMD**
1	0.02	0.05	0.31	0.58
3	0.11	7.56	0.08	0.11
6	10.42	0.16	0.14	0.14
9	0.12	0.14	0.21	0.20
12	0.16	0.11	0.03	0.20
15	0.07	0.13	0.20	0.24
16	0.02	0.02	0.23	0.02
18	0.01	0.05	0.24	0.25
21	0.01	0.11	6.78	0.16
24	0.02	0.08	0.01	0.06

Table 6.2: Confidence intervals for Compact- and Mini-HYDRA's runtime on Napier

power consumption. Compact-HYDRA somewhat concurs with this but the results indicate a slight performance gain in Haswell's (1.06×) favour.

Next the impact of the fused multiply set of instructions is examined, examples of which are fused multiply-add, fused multiply-subtract and their negative variants (e.g. fused negative multiply-add). These instructions were introduced because it was recognised that pairs of multiply and addition/subtraction operations are common in HPC codes (among others) and including a single instruction to handle this would reduce instruction latency [71]. In Figure 6.4 the runtime is plotted for both compact- and mini-HYDRA compiled with and with-

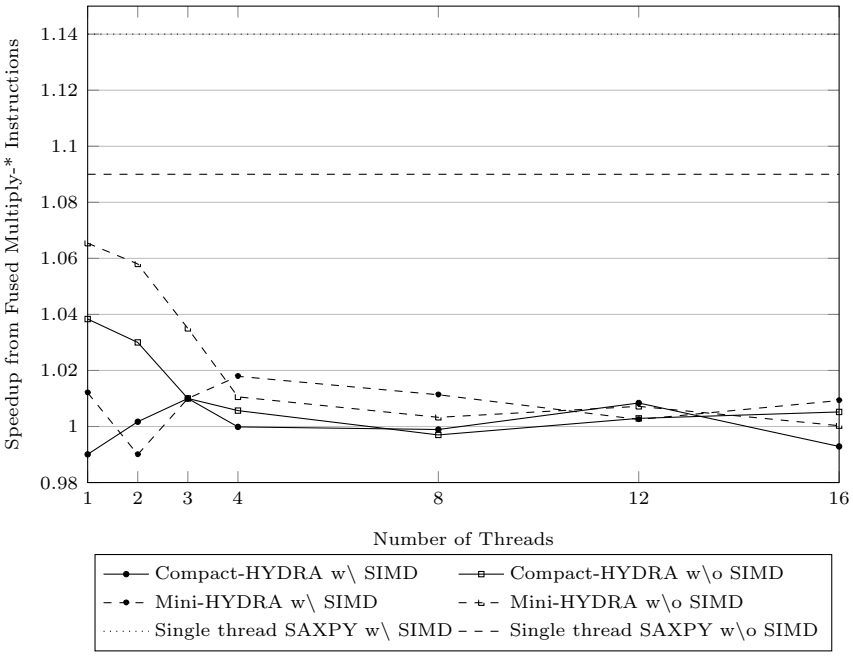


Figure 6.4: Impact of using the Intel Fused Multiply-* instructions with and without auto-vectorisation on both Compact- and Mini-HYDRA

out FM-* instructions and with and without auto-vectorisation. Additionally a plot showing the impact of these instructions on a simple benchmark code which implements Single-Precision AX Plus Y (SAXPY) (BlasBench [122]) which is used as realistic upper limit on the achievable performance improvement.

From Figure 6.4 it can be seen that the impact of FM-* instruction is positive (1.06×) on a single core, but this gain quickly drops off as the number of OpenMP threads increases. This is likely due to the FM-* instructions increasing the intensity of memory operations coupled with the decrease in memory bandwidth that is available per core as scaling occurs. When SIMD is used in conjunction with FM-* instructions the benefit of using FM-* with this kernel is negligible, presumably because SIMD instructions increase pressure on the memory bandwidth through both a reduction in instruction latency and the operation on larger registers. Both of these trends are also apparent in compact-HYDRA.

Figure 6.5 indicates that auto-vectorisation is not suitable for an unmodified

version of this kernel. The performance for both Ivybridge and Haswell is at best break even; on a single thread the speedup is 0.86 and 0.90 for mini-HYDRA running on Ivybridge and Haswell respectively. This slowdown is due to a large number of memory write stalls (data collected from the Performance Application Programming Interface (PAPI) library) at lower thread counts. An attempt to reduce the number of stalls was made by scattering the OpenMP threads across the sockets, however this had no impact on performance.

When comparing the impact of auto-vectorisation when running on either Haswell or Ivybridge, it is clear that the new features introduced for Haswell (e.g. vector gathers) have not alleviated the underlying bottleneck preventing the auto-vectorised code from out performing the non-vector code. When analysing the PAPI counters for mini-HYDRA running on Haswell, it is seen that while the total number of instructions is less for the auto-vectorised version of the code, the total number of cycles is larger, presumably due to the number of memory write stalls.

The performance trends which are represented in Figure 6.5 and their root causes are also present in compact-HYDRA. This gives further evidence for the suitability of this mini-application to represent the memory access patterns of HYDRA.

6.4 Summary

The chapter has focused on the development, validation and initial use of a mini-application representative of geometric multigrid, unstructured mesh applications.

First, existing research on mini-application development is distilled in to a set of questions (items 1-4 below) and extended with a new consideration (item 5 below):

1. Where does the application spend most of its execution time?

2. What performance characteristics will the mini-application capture?

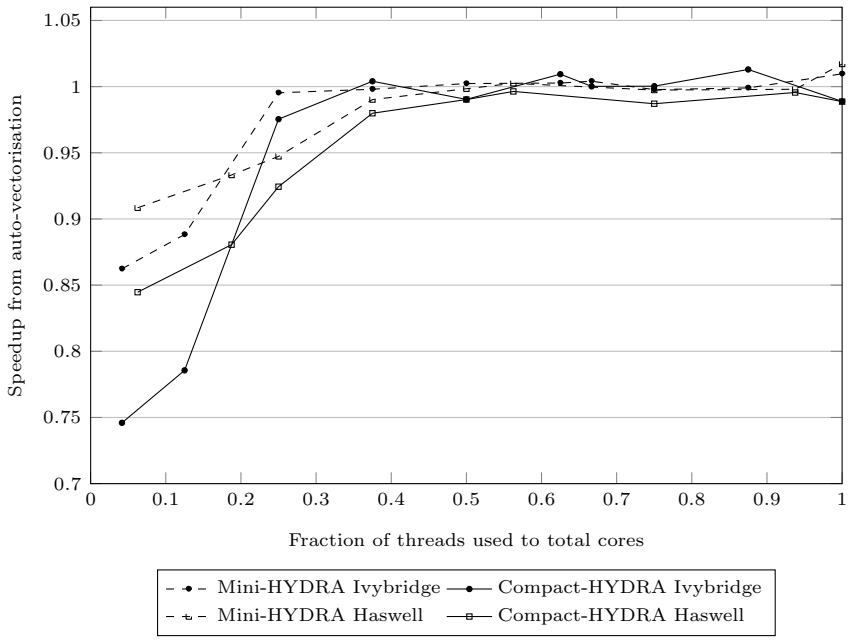


Figure 6.5: Performance impact of using auto-vectorisation on both Compact- and Mini-HYDRA on both Intel Ivybridge and Intel Haswell

3. Can any part of the development process be automated?

4. How can the build system be made as simple as possible?

5. Which aspects of the simulation contribute to the compute behaviour within the most expensive regions of code.

This final consideration is helpful for the development of a mini-application as it maps performance features to implementable code features. Following this, a survey of already existing mini-applications was carried out to assess their suitability at representing geometric multigrid, unstructured mesh applications. A mini-application which was most similar in nature in terms of simulation makeup to HYDRA's core routines was extended to support HYDRA's inputs geometries by extending the code with multigrid behaviours.

Next, two previously developed mini-application validation techniques were applied to a class of code which they have not previously been used on. This validation highlighted that the mini-application was similar to HYDRA's `iflux`

routine. This analysis highlighted that while the scaling behaviour achieved and the PAPI counters largely indicate that the hardware was being stressed in a similar way by both compact- and mini-HYDRA, the mini-application exhibited a greater number (2-3×) of level two cache reads. This difference does not necessarily need adjusting, but needs to be taken into consideration when interpreting performance results and considering them in the context of the parent code.

Finally the use of the mini-application is demonstrated by assessing the impact of the new architectural features added to Intel Haswell when compared to Intel Ivybridge on geometric multigrid, unstructured mesh applications. It is found that the `iflux` kernel is unlikely to benefit from the use of FM-* instructions or from auto-vectorisation without making significant code changes (e.g. changes to memory layout). However, this does not exclude more compute intensive kernels from benefiting from auto-vectorisation without code modifications. These results are validated successfully against the compact version of `iflux`, further indicating the mini-applications suitability for use in studies by third parties.

CHAPTER 7

Conclusions and Future Work

This thesis has detailed the development and deployment of a generalised runtime performance model for unstructured mesh, geometric multigrid applications. Further, it has outlined the use and development of two mini-applications to aid the performance engineering process.

In Chapter 3, the development of an automatic instrumentation process was detailed, that allows unprecedented flexibility in terms of implementation language, the language to be instrumented and the instrumentation to be inserted. This process was implemented and then demonstrated by applying a common set of instrumentation to three different variants of HYDRA, two being chronological releases and the third being optimised to reduce synchronisation costs. This instrumentation tool was fundamental to the performance engineering activities conducted in this thesis. It was used i) to compare historical changes in performance and to quantify the impact of the synchronisation optimisation; and, ii) to collect both static data from the source code and timing information at runtime for use with the runtime performance model.

Chapter 4 detailed the development of a runtime performance model of unstructured mesh, geometric multigrid codes, which was able to capture the expected scaling behaviour of HYDRA, and its proprietary communications library, Oxford Parallel Library for Unstructured Solvers (OPlus), when running on a grid with one level. Through the introduction of a small number of additional model terms, this model was generalised to multigrid simulations. The simple relationship between these two models significantly reduces the complexity of benchmarking a new platform, as it enables the extrapolation of complete production runtimes using data collected from the execution of small single-level

benchmarks. The power of the performance model was demonstrated by identifying a synchronisation issue which degraded performance by up to 29.22% on machine configurations with poor support for overlapping computation with communication. An optimisation was subsequently applied which decreased the cost of communication and synchronisation by 3.01× and total runtime by up to 1.41×. That it was possible to accelerate HYDRA to such a degree demonstrates both the accuracy of the model and the importance of reassessing whether an application's original design assumptions still hold on new hardware/software configurations.

In Chapter 5 the analytical runtime model for multigrid applications was further generalised to support multiple cycle types (e.g. V-Cycle, W-Cycle) and a variable number of Runge-Kutta iterations. Additional details were incorporated in to the performance model: buffer pack/unpack costs, runtime costs from all 300+ loops in the code base, and performance information for different memory access patterns. These additional details were validated on up to 1,024 cores of a Haswell-based cluster, using both a geometric partitioning algorithm and ParMETIS to partition the NASA Rotor37 input deck, with a maximum absolute error of 12.63% and 11.55% respectively. These additions to the model allow a wider range of unstructured mesh workloads to be successfully modelled.

Additionally, Chapter 5 detailed the development of Moses, an unstructured domain decomposition mini-application, which was able to convert partitioning data from multiple algorithms/libraries (ParMETIS, METIS and Scotch) up to 100,000 partitions in scale for use with the runtime performance model. This data has allowed predictions to be made at scale without first running HYDRA at equivalent process counts to collect set sizes. Runtime predictions made using this data have an under prediction in runtime of at most 7.31% over 512 processes, when compared against predictions made with empirically collected partitioning data. The use of Moses is demonstrated in conjunction with the runtime performance model to predictively compare the relative effect on HYDRA's runtime of using Scotch, ParMETIS, METIS and a geometric

partitioning algorithm on up to 30,000 cores. Following this, the predicted observation that the geometric partitioning algorithm causes reduced performance in HYDRA at 512 processes when compared with ParMETIS is validated.

Finally, this thesis documented a mini-application that performs computation over edges which operates on datasets with the following properties: i) unstructured mesh, ii) geometric multigrid, and iii) a variable number of neighbours per node. It was validated using two previously developed techniques which have not previously been applied to this class of code. These techniques provided evidence of the similarity between the mini-application and the parent code in terms of their shared memory scalability. Finally, the use of the mini-application was demonstrated by quantifying the impact of the new hardware features introduced by the Intel Haswell platform over Intel Ivybridge for geometric multigrid, unstructured mesh applications. It was found that FM-* instructions and the AVX2 features have a limited impact on performance, but there is potential for Intel Haswell to deliver application results at a much lower total energy and to impact more compute heavy kernels. It is hoped that this mini-application will be used in future as a vehicle for optimisation, porting and execution on machines where HYDRA is not cleared for use.

7.1 Research Impact

The work in this thesis has enhanced the selection of tools and techniques available for Rolls-Royce to use in their mission to continuously move their codes forward. The key impacts of these tools are highlighted below.

- Chapter 3 describes a flexible source instrumentation tool. It has eased the collection of data from common performance libraries (e.g. Performance Application Programming Interface (PAPI) and VTune) by allowing a common set of instrumentation rules to be applied to all of HYDRA's source, over three different variants of HYDRA on a mixture of platforms (Intel, BlueGene/Q and Power8). This tool is expected to be applied to

other codes, such as PRECISE, at Rolls-Royce in the near future.

- This thesis presents the first Message Passing Interface (MPI) performance model (Chapter 4) of HYDRA and OPlus; it has been used support Rolls-Royce in procurement activities by validating HYDRA's scaling behaviour on a variety of machines at leading UK High Performance Computing (HPC) installations such as The Hartree Centre, Edinburgh Parallel Computing Centre and MidPlus resource locations. Most notably, the performance model was used to i) give increased confidence of HYDRA's performant scaling behaviour on sample Intel Ivybridge-based hardware delivered to Rolls-Royce for evaluation, and ii) identify an optimisation that brought performance in line with expectations on a Haswell-based machine, leading to a decrease the cost of communication and synchronisation by 3.01×.

- This research has provided Rolls-Royce with a representative geometric multigrid, unstructured mesh application (Chapter 6) which has been used to provide feedback on the running of and the potential impact of new hardware on HYDRA's `iflux` kernel. Additionally, this mini-application paves the way for comprehensive studies into the suitability of accelerator cards and different parallel computing frameworks for this class of code.

7.2 Limitations

The limitations inherent to the auto-instrumentation tool are a direct result of the trade-offs made when selecting the level at which the tool should instrument the application (i.e. source, abstract syntax tree, or binary). As discussed in Chapter 3 it was chosen to implement source based instrumentation due to its flexibility and its potential lack of reliance on third-party libraries. This choice, despite its advantages brings with it several limitations. For each new programming language which needs to be instrumented, the auto-instrumentation tool will need to be updated. This flaw is mitigated in part by the fact that

support for a language can be added by defining a few key properties, such as the comment and line continuation characters. Furthermore, HPC applications tend to be written in a small subset of the languages available today, primarily FORTRAN-like and C-like languages; good application coverage could be achieved by handling just these. Another inherent limitation of the auto-instrumentation tool is the requirement to have access to the source code (either direct or indirect), which is problematic for some applications such as LS-DYNA where the source code is not readily available.

The primary limitation of the runtime performance model, which was developed in Chapters 4 and 5, is the scale at which it has been validated for (up to 1024 processors). This severely limits the validity of the conclusions which can be drawn from the performance models predictions at large scale. However, the vast majority of HYDRA's capacity runs fall within the range which the model has been validated for, so it was justified for the work to begin by focusing on this scale. However, the need to support capability runs of HYDRA is recognised and as part of this, two challenges were identified: acquiring representative partitioning data, and then running the code at scale. Towards overcoming these, Moses was developed in Chapter 5 to provide representative partitioning data.

However, Moses is not without limitation: i) the inaccuracy of the partitioning data generated for the coarser levels of the multigrid and ii) the current state of its generality. The impact of this first limitation is discussed in Chapter 5 and despite this inaccuracy, Moses is still able to generate usable partitioning data. However, this has only been tested on a single dataset (Rotor37) and this error may cause issues with datasets or code changes where the higher levels of multigrid contribute to an increased percentage of the runtime. This could happen if higher order methods are applied to the coarser levels than to the finer levels.

The second limitation of Moses relates to its generality with respect to other unstructured mesh multigrid codes and their datasets. This mini-application was developed for HYDRA, however HYDRA is an instance of the Access-

Execute abstraction and therefore Moses will be applicable to other codes which instance this abstraction. However, codes which are of the same class but implement different communication strategies such as those which do not overlap computation and communication will need new partitioning rules. Likewise, new rules will need to be implemented for datasets which use mesh features other than nodes and edges, such as cells.

In Chapter 6, a mini-application is developed which is representative of HYDRA's edge-based computations with the intention for it to be used to aid procurement, optimisation and porting activities. The main limitation of this mini-application relates to its code coverage. As while it replicates the unstructured memory access patterns that more than 70% of the code uses, it only considers the arithmetic intensity exhibited by `iflux`. This means that while optimisations to the memory access pattern are likely to relate directly to the parent code, more specific compute optimisations are only likely to translate back to `iflux`. In addition to the above limitation, the validation method requires a compact version of the code to validate against which is a time consuming endeavour.

7.3 Future Work

It is planned to apply the auto-instrumentation tool developed in Chapter 3 to additional codes in use at Rolls-Royce, to aid in the application of performance engineering techniques such as those applied to HYDRA in this thesis. The likely target for this study is PRECISE which is, like HYDRA, a Computational Fluid Dynamics (CFD) code. PRECISE is written in C++ therefore, the auto-instrumentation tool will need to be primed with the line continuation and comment tokens in use by this language. Preliminary work on this saw C++ code being instrumented after only a few hours of development.

Also, the aim is to use the tool to apply instrumentation exposed by performance engineering frameworks such as TAU and Intel VTune. The product of

this work would be several rule sets which could be applied to numerous codes at Rolls-Royce to collect performance information for use in common analyses. Preliminary work towards this includes a rule set to insert VTune instrumentation which is able to start and stop the collection of data around any subset of OPlus loops.

To address some of the limitations of the runtime performance model developed across Chapters 4 and 5, the intention is to augment the model to cover additional areas of the code (such as I/O and initial setup routines); accounting for the effects of background noise (due to other users); and considering cache effects (due to decreasing local problem size at scale).

The biggest limitation of the runtime performance model is the lack of validation results at scale. To rectify this, validations will be performed at larger scale and the increasing error due to synchronisation cost will be addressed either through optimisations to the code or modelling additional performance factors.

As part of Chapter 5 a mini-application (Moses) representative of the partitioning process internal to HYDRA was developed. Several improvements are planned: i) decreasing the mini-application's runtime error (7.31%) by increasing the accuracy of predicting the set sizes on the highest levels of the multigrid; ii) increasing the scale at which the mini-application is able to generate set and halo data (beyond 100,000 and towards 1,000,000 partitions); and, iii) extend Moses to support other unstructured mesh applications.

Moses, in conjunction with the runtime performance model of HYDRA, delivered several predictions which need validating or acting upon. First, HYDRA's partitioning process will be extended such that it is able to read in the mesh partitioning data from serial algorithms and second, a performance analysis must be run to determine if the effect of these partitions on HYDRA's runtime matches the predicted effect. Furthermore the intent is to predictively and empirically analyse the effect of different partitioning algorithms on HYDRA's runtime when using a variety of datasets, as the plan is to use the runtime per-

formance model to examine the continued effectiveness of these algorithms as new datasets are brought into use.

Expansions to the mini-application developed in Chapter 6 are planned to cover more kernels so that a higher percentage of HYDRA's performance can be represented in code bases which are more readily available for use by third-parties. This larger representation of the code base will increase the relevance of performance engineering activities with the mini-applications. Preliminary work has begun on developing compact-`vflux` but it has not been fully integrated into the mini-application in the same way that `iflux` has. Work has not yet started on a mini-application version of `vflux`; for this to become a reality a potential developer must either already have or acquire a strong grounding in CFD techniques, such as turbulence modelling.

7.4 Final Word

The push towards exascale has seen the end of Dennard scaling and caused a fundamental shift in the degree of parallelism available in supercomputers. This move towards multi-core, many-core and accelerator architectures has prompted the development of a plethora of performance engineering techniques to support them and the codes which they execute. The outlook is promising that through these techniques, industry and the research community will overcome the known and unknown challenges on the journey to realising the first exascale machine.

Bibliography

[1] M. F. Adams, J. Brown, J. Shalf, B. Van Straalen, E. Strohmaier, and S. Williams. HPGMG 1.0: A Benchmark for Ranking High Performance Computing Systems. Technical report, Lawrence Berkeley National Laboratory, 2014.

[2] V. S. Adve. *Analyzing the Behavior and Performance of Parallel Programs*. PhD thesis, University of Wisconsin-Madison, 1993.

[3] V. S. Adve and M. K. Vernon. Performance Analysis of Mesh Interconnection Networks With Deterministic Routing. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):225–246, March 1994.

[4] B. Agathangelou and M. Gascoyne. Aerodynamic Design Considerations of a Formula 1 Racing Car. Technical report, SAE Technical Paper, Warrendale, PA, February 1998.

[5] 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):71–79, July 1997.

[6] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the Joint Computer Conference 1967 (SJCC'67)*, pages 483–485, Atlantic City, NJ, April 1967. ACM, New York, NY.

[7] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, et al. The Landscape of Parallel Computing Research: A View From Berkeley. Technical report, University of California, EECS Department, University of California, Berkeley, CA, December 2006.

[8] M. K. Bane and G. Riley. Automatic Overheads Profiler for OpenMP Codes. In *Proceedings of the 2nd European Workshop on OpenMP 2000 (EWOMP'00)*, pages 162–166, Edinburgh, Scotland, September 2000. Springer, Berlin, Germany.

[9] 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 Supercomputing 2008 (SC'08)*, pages 453–469, Austin, TX, November 2008. IEEE Computer Society, Los Alamitos, CA.

[10] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. A Performance Evaluation of the Nehalem Quad-Core Processor for Scientific Computing. *Parallel Processing Letters*, 18(04): 453–469, December 2008.

[11] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. Sancho. Using Performance Modeling to Design Large-Scale Systems. *Computer*, 42(10):0042–49, November 2009.

[12] D. A. Beckingsale. *Towards Scalable Adaptive Mesh Refinement on Future Parallel Architectures*. PhD thesis, The University of Warwick, February 2015.

[13] D. A. Beckingsale, O. Perks, W. Gaudin, J. Herdman, and S. Jarvis. Optimisation of Patch Distribution Strategies for AMR Applications. In *Computer Performance Engineering*, volume 7587 of *Lecture Notes in Computer Science*, pages 210–223. Springer, Berlin, Germany, 2013.

[14] R. F. Bird, S. A. Wright, D. A. Beckingsale, and S. A. Jarvis. Performance Modelling of Magnetohydrodynamics Codes. In *Computer Performance Engineering*, volume 7587 of *Lecture Notes in Computer Science*, pages 197–209. Springer, Berlin, Germany, 2013.

[15] T. Brecht. On the Importance of Parallel Application Placement in NUMA Multiprocessors. In *Proceedings of USENIX Systems on USENIX Experiences With Distributed and Multiprocessor Systems 1993 (SEDMS'93)*, pages 1–18, San Diego, CA, September 1993. USENIX Association, Berkeley, CA.

[16] W. L. Briggs. *Multigrid Tutorial*. SIAM, Philadelphia, PA, 1987. ISBN 0898712211.

[17] R. Brightwell, K. Underwood, and R. Riesen. An Initial Analysis of the Impact of Overlap and Independent Progress for MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *Lecture Notes in Computer Science*, pages 370–377. Springer, Berlin, Germany, September 2004.

[18] S. Browne, J. 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, August 2001.

[19] B. Buck and J. K. Hollingsworth. An API for Runtime Code Patching. *International Journal of High Performance Computing Applications*, 14 (4):317–329, November 2000.

[20] T. N. Bui and C. Jones. Finding Good Approximate Vertex and Edge Partitions Is NP-hard. *Information Processing Letters*, 42(3):153–159, May 1992.

[21] J. M. Bull. Measuring Synchronisation and Scheduling Overheads in OpenMP. In *Proceedings of First European Workshop on OpenMP 1999 (EWOMP'99)*, pages 99–105, Lund, Sweden, September 1999.

[22] J. M. Bull and D. O'Neill. A Microbenchmark Suite for OpenMP 2.0. *ACM SIGARCH Computer Architecture News*, 29(5):41–48, December 2001.

[23] R. A. Bunt, S. J. Pennycook, S. A. Jarvis, L. Lapworth, and Y. K. Ho. Model-Led Optimisation of a Geometric Multigrid Application. In *Proceedings of the 15th High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing 2013 (HPCC&EUC'13)*, pages 742–753, Zhang Jia Jie, China, November 2013. IEEE Computer Society, Los Alamitos, CA.

[24] R. A. Bunt, S. A. Wright, S. A. Jarvis, M. Street, and Y. K. Ho. Predictive Evaluation of Partitioning Algorithms Through Runtime Modelling. In *In Proceedings of High Performance Computing, Data, and Analytics (HiPC'16)*, pages 1–11, Hyderabad, India, December 2016. IEEE Computer Society, Los Alamitos, CA.

[25] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13–25, June 1997.

[26] D. A. Burgess, P. I. Crumpton, and M. B. Giles. A Parallel Framework for Unstructured Grid Solvers. In *Proceedings of the 2nd European Computational Fluid Dynamics Conference 1994*, pages 391–396, Stuttgart, Germany, September 1994. Wiley, Hoboken, NJ.

[27] J. C. Butcher. A History of Runge-Kutta Methods. *Applied Numerical Mathematics*, 20(3):247–260, March 1996.

[28] K. W. Cameron and X. Sun. Quantifying Locality Effect in Data Access Delay: Memory logP. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium 2003 (IPDPS'03)*, pages 8–pp, Nice, France, April 2003. IEEE Computer Society, Los Alamitos, CA.

[29] M. S. Campobasso and M. B. Giles. Stabilization of a Linearized Navier-Stokes Solver for Turbomachinery Aeroelasticity. In *Proceedings of the 2nd International Conference on Computational Fluid Dynamics 2002 (ICCFD'02)*, pages 343–348, Sydney, Australia, July 2002. Springer, Berlin, Germany.

[30] M. S. Campobasso and M. B. Giles. Effects of Flow Instabilities on the Linear Analysis of Turbomachinery Aeroelasticity. *Journal of Propulsion and Power*, 19(2):250–259, March 2003.

[31] P. E. Ceruzzi. When Computers Were Human. *Annals of the History of Computing*, 13(3):237–244, July 1991.

[32] J. W. Chew and N. J. Hills. Computational Fluid Dynamics for Turbomachinery Internal Air Systems. *Philosophical Transactions. Series A, Mathematical, Physical, and Engineering Sciences*, 365(1859):2587–611, October 2007.

[33] P. K. Chittimalli and V. Shah. GEMS: A Generic Model Based Source Code Instrumentation Framework. In *Proceedings of the International Conference on Software Testing, Verification and Validation 2012 (ICST'12)*, pages 909–914, Montreal, Canada, April 2012. IEEE Computer Society, Los Alamitos, CA.

[34] A. Corrigan, F. F. Camelli, R. Löhner, and J. Wallin. Running Unstructured Grid-Based CFD Solvers on Modern Graphics Hardware. *International Journal for Numerical Methods in Fluids*, 66(2):221–229, May 2011.

[35] P. I. Crumpton and M. B. Giles. Aircraft Computations Using Multigrid and an Unstructured Parallel Library. In *Proceedings of the 33rd Aerospace Sciences Meeting and Exhibit 1994*, Reno, NV, January 1994. American Institute of Aeronautics and Astronautics, Reston, VA.

[36] P. I. Crumpton and M. B. Giles. Mutigrid Aircraft Computations Using the OPlus Parallel Library. In *Proceedings of Parallel Computational Fluid Dynamics: Implementation and Results Using Parallel Computers 1995*, pages 339–346, Pasadena, CA, June 1995. Elsevier, Amsterdam, The Netherlands.

[37] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A Practical Model of Parallel Computation. *Communications of the ACM*, 39(11):78–85, November 1996.

[38] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, 1998. ISBN 1558603433.

[39] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. RAPL: Memory Power Estimation and Capping. In *Proceedings of the International Symposium on Low-Power Electronics and Design 2010 (ISLPED'10)*, pages 189–194, Austin, TX, August 2010. IEEE Computer Society, Los Alamitos, CA.

[40] 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.

[41] K. Davis, K. J. Barker, and D. J. Kerbyson. Performance Prediction via Modeling: A Case Study of the ORNL Cray XT4 Upgrade. *Parallel Processing Letters*, 19(04):619–639, December 2009.

[42] T. Defraeye, B. Blocken, E. Koninckx, P. Hespel, and J. Carmeliet. Computational Fluid Dynamics Analysis of Cyclist Aerodynamics: Performance of Different Turbulence-Modelling and Boundary-Layer Modelling Approaches. *Journal of Biomechanics*, 43(12):2281–2287, May 2010.

[43] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[44] L. DeRose, T. Hoover, and J. K. Hollingsworth. The Dynamic Probe Class Library – An Infrastructure for Developing Instrumentation for Performance Tools. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium 2001 (IPDPS'01)*, pages 10066.2–, San Francisco, CA, April 2001. IEEE Computer Society, Los Alamitos, CA.

[45] J. J. Dongarra. The LINPACK Benchmark: An Explanation. In *Proceedings of Supercomputing 1987 (ISC'87)*, pages 456–474, Athens, Greece, June 1987. Springer, Berlin, Germany.

[46] J. J. Dongarra and M. A. Heroux. Toward a New Metric for Ranking High Performance Computing Systems. Technical report, Sandia National Laboratory, Albuquerque, NM, June 2013.

[47] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK Benchmark: Past, Present and Future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, July 2003.

[48] M. C. Duta, M. B. Giles, and M. S. Campobasso. The Harmonic Adjoint Approach to Unsteady Turbomachinery Design. *International Journal for Numerical Methods in Fluids*, 40(3–4):323–332, September 2002.

[49] M. C. Duta, M. B. Giles, and M. S. Campobasso. The Harmonic Adjoint Approach to Unsteady Turbomachinery Design. *International Journal for Numerical Methods in Fluids*, 40(3-4):323–332, September 2002.

[50] J. Eisenbiegler, W. Lowe, and A. Wehrenpfennig. On the Optimization by Redundancy Using an Extended LogP Model. In *Proceedings of Advances in Parallel and Distributed Computing 1997 (APDC'97)*, pages 149–155, Shanghai, China, March 1997. IEEE Computer Society, Los Alamitos, CA.

[51] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Power Challenges May End the Multicore Era. *Communications of the ACM*, 56(2):93–102, Februrary 2013.

[52] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard. Cray Cascade: A Scalable HPC System Based on a Dragonfly Network. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis 2012*, pages 103:1–103:9, Salt Lake City, UT, November 2012. IEEE Computer Society, Los Alamitos, CA.

[53] W. Feng and K. W. Cameron. The Green500 List: Encouraging Sustainable Supercomputing. *Computer*, 40(12):50–55, December 2007.

[54] J. T. Feo. An Analysis of the Computational and Parallel Complexity of the Livermore Loops. *Parallel Computing*, 7(2):163–185, June 1988.

[55] E. J. Fluhr, J. Friedrich, D. Dreps, V. Zyuban, G. Still, C. Gonzalez, A. Hall, D. Hogenmiller, F. Malgioglio, R. Nett, J. Paredes, J. Pille, D. Plass, R. Puri, P. Restle, D. Shan, K. Stawiasz, Z. T. Deniz, D. Wendel, and M. Ziegler. POWER8TM: A 12-core Server-Class Processor in 22nm SOI with 7.6Tb/s Off-Chip Bandwidth. In *Proceedings of the IEEE International Solid-State Circuits Conference Digest of Technical Papers 20014 (ISSCC'14)*, pages 96–97, San Francisco, CA, February 2014. IEEE Computer Society, Los Alamitos, CA.

[56] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing 1978 (STOC'78)*, pages 114–118, San Diego, CA, May 1978. ACM, New York, NY.

[57] M. I. Frank, A. Agarwal, and M. K. Vernon. LoPC: Modeling Contention in Parallel Algorithms. *SIGPLAN Not.*, 32(7):276–287, June 1997.

[58] K. Frlinger and M. Gerndt. ompP: A Profiling Tool for OpenMP. In *OpenMP Shared Memory Parallel Programming*, pages 15–23. Springer, Berlin, Germany, 2008.

[59] H. Gahvari, A. H. Baker, M. Schulz, U. M. Yang, K. E. Jordan, and W. Gropp. Modeling the Performance of an Algebraic Multigrid Cycle on HPC Platforms. In *Proceedings of the International Conference on Supercomputing 2011 (ISC'11)*, pages 172–181, Tucson, AZ, June 2011. ACM, New York, NY.

[60] H. Gahvari, W. Gropp, K. E. Jordan, M. Schulz, and U. M. Yang. Performance Modeling of Algebraic Multigrid on Blue Gene/Q: Lessons Learned. In *Proceedings of the Workshop on High Performance Computing, Networking, Storage and Analysis 2012 (SCC'12)*, pages 377–385, Salt Lake City, Utah, November 2012. IEEE Computer Society, Los Alamitos, CA.

[61] H. Gahvari, W. Gropp, K. E. Jordan, M. Schulz, and U. M. Yang. Modeling the Performance of an Algebraic Multigrid Cycle Using Hybrid MPI/OpenMP. In *Proceedings of the 41st International Conference on Parallel Processing 2012 (ICPP'12)*, pages 128–137, Pittsburgh, PA, September 2012. IEEE Computer Society, Los Alamitos, CA.

[62] P. Gepner and M. F. Kowalik. Multi-Core Processors: New Way to Achieve High System Performance. In *Proceedings of the International Symposium on Parallel Computing in Electrical Engineering 2006 (PAR ELEC'06)*, pages 9–13, Bialstok, Poland, September 2006. IEEE Computer Society, Los Alamitos, CA.

[63] J. R. Gilbert, G. L. Miller, and S.-H. Teng. Geometric Mesh Partitioning: Implementation and Experiments. *SIAM Journal on Scientific Computing*, 19(6):2091–2110, 1998.

[64] M. B. Giles, M. C. Duta, J. Mller, and N. A. Pierce. Algorithm Developments for Discrete Adjoint Methods. *AIAA Journal*, 41(2):198–205, February 2003.

[65] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H. Kelly. Per-

formance Analysis of the OP2 Framework on Many-Core Architectures. *SIGMETRICS Performance Evaluation Review*, 38(4):9–15, March 2011.

[66] M. B. Giles, G. R. Mudalige, C. Bertolli, P. H. Kelly, E. Laszlo, and I. Z. Reguly. An Analytical Study of Loop Tiling for a Large-Scale Unstructured Mesh Application. In *Proceedings of High Performance Computing, Networking, Storage and Analysis 2012 (SC'12)*, pages 477–482, Salt Lake City, Utah, November 2012. IEEE Computer Society, Los Alamitos, CA.

[67] M. B. Giles, G. R. Mudalige, B. Spencer, C. Bertolli, and I. Z. Reguly. Designing OP2 for GPU Architectures. *Journal of Parallel and Distributed Computing*, 73(11):1451–1460, November 2013.

[68] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.*, 17(6):120–126, June 1982.

[69] J. L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, May 1988.

[70] D. Hackenberg, R. Oldenburg, D. Molka, and R. Schone. Introducing FIRESTARTER: A Processor Stress Test Utility. In *Proceedings of the International Green Computing Conference 2013 (IGCC'13)*, pages 1–9, Arlington, VA, June 2013. IEEE Computer Society, Los Alamitos, CA.

[71] P. Hammarlund, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. Haswell: The Fourth-Generation Intel Core Processor. *IEEE Micro*, 34(2):6–20, March 2014.

[72] S. D. Hammond, G. R. Mudalige, J. A. Smith, S. A. Jarvis, A. J. Herdman, and A. Vadgama. WARPP: A Toolkit for Simulating High-Performance Parallel Scientific Codes. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques 2009 (ICSTT'09)*, pages 1–10, Rome, Italy, March 2009. ICST, Gent, Belgium.

[73] S. D. Hammond, G. R. Mudalige, J. A. Smith, A. B. Mills, S. A. Jarvis, J. Holt, I. Miller, J. A. Herdman, and A. Vadgama. Performance Prediction and Procurement in Practice: Assessing the Suitability of Commodity Cluster Components for Wavefront Codes. *IET Software*, 3(6):509–521, December 2009.

[74] R. Hanna. Can CFD Make a Performance Difference in Sport? *Sports Engineering*, 5(4):17–30, November 2002.

[75] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, A. G. Shet, G. Chrysos, and P. Dubey. Design and Implementation of the Linpack Benchmark for Single and Multi-Node Systems Based on Intel® Xeon Phi Coprocessor. In *Proceedings of the 27th International Parallel and Distributed Processing Symposium 2013 (IPDPS'13)*, pages 126–137, Boston, MA, May 2013. IEEE Computer Society, Los Alamitos, CA.

[76] J. Hemminger, G. Fleming, and M. Ratner. *Directing Matter and Energy: Five Challenges for Science and the Imagination.* Department of Energy's Office of Science, 2007.

[77] M. Heroux and R. Barrett. Mantevo Project. `https://mantevo.org/` (accessed March 3, 2016), March 2016.

[78] J. M. Hill, P. I. Crumpton, and D. A. Burgess. Theory, Practice, and a Tool for BSP Performance Prediction. In *Proceedings of the 2nd European Conference on Parallel Processing 1996 (Euro-Par'96)*, pages 697–705, Lyon, France, August 1996. Springer, Berlin, Germany.

[79] R. W. Hockney. The Communication Challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20(3):389–398, March 1994.

[80] H. P. Hodson and R. G. Dominy. Three-Dimensional Flow in a Low-Pressure Turbine Cascade at Its Design Condition. *Journal of Turbomachinery*, 109(2):177–185, April 1987.

[81] H. P. Hodson and R. G. Dominy. The Off-Design Performance of a Low-Pressure Turbine Cascade. *Journal of Turbomachinery*, 109(2):201–209, April 1987.

[82] T. Hoefler, L. Cerquetti, T. Mehlan, F. Mietke, and W. Rehm. A Practical Approach to the Rating of Barrier Algorithms Using the LogP Model and Open MPI. In *Proceedings of the International Conference on Parallel Processing Workshops 2005 (ICPPW'05)*, pages 562–569, Oslo, Norway, June 2005. IEEE Computer Society, Los Alamitos, CA.

[83] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm. LogfP - A Model for Small Messages in InfiniBand. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium 2006 (IPDPS'06)*, pages 319–319, Washington, DC, USA, April 2006. IEEE Computer Society, Los Alamitos, CA.

[84] T. Hoefler, T. Schneider, and A. Lumsdaine. LogGOPSim: Simulating Large-Scale Applications in the LogGOPS Model. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing 2010 (HPDC'10)*, pages 597–604, Chicago, Illinois, June 2010. ACM, New York, NY.

[85] A. Hoisie, G. Johnson, D. J. Kerbyson, M. Lang, and S. Pakin. A Performance Comparison Through Benchmarking and Modeling of Three Leading Supercomputers: Blue Gene/L, Red Storm, and Purple. In *Proceedings of Supercomputing 2006 (SC'06)*, pages 3–3, Tampa, FL, November 2006. IEEE Computer Society, Los Alamitos, CA.

[86] S. Homer and M. Peinado. Design and Performance of Parallel and Distributed Approximation Algorithms for Maxcut. *Journal of Parallel and Distributed Computing*, 46(1):48–61, October 1997.

[87] W. Hwu and Y. N. Patt. HPSm, a High Performance Restricted Data

Flow Architecture Having Minimal Functionality. *SIGARCH Comput. Archit. News*, 14(2):297–306, May 1986.

[88] Intel Corporation. Intel MPI Benchmark User Guide and Methodology Description 3.2.3. Technical report, Intel Corporation, Santa Clara, CA, 2011.

[89] Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual. Technical report, Intel Corporation, Santa Clara, CA, January 2016.

[90] Intel Corporation. And IA-32 Architectures Software Developer's Manual Instruction Set Reference. Technical report, Intel Corporation, Santa Clara, CA, April 2016.

[91] M. Itzkowitz and Y. Maruyama. HPC Profiling With the Sun Studio Performance Tools. In *Tools for High Performance Computing 2009*, pages 67–93. Springer, Berlin, Germany, May 2010.

[92] H. Jasak, A. Jemcov, and Z. Tukovic. OpenFOAM: A C++ Library for Complex Physics Simulations. In *Proceedings of the International Workshop on Coupled Methods in Numerical Dynamics 2007 (CMND'07)*, pages 1–20, Dubrovnik, Croatia, September 2007. University of Zagreb.

[93] F. T. Johnson, E. N. Tinoco, and N. J. Yu. Thirty Years of Development and Application of CFD at Boeing Commercial Airplanes. *Computers & Fluids*, 34(10):1115–1151, December 2005.

[94] G. Johnson, D. J. Kerbyson, and M. Lang. Optimization of Infiniband for Scientific Applications. In *Proceedings of the 22nd International Parallel and Distributed Processing Symposium 2008 (IPDPS'08)*, pages 1–8, Miami, FL, April 2008. IEEE Computer Society, Los Alamitos, CA.

[95] T. Kalinowski, I. Kort, and D. Trystram. List Scheduling of General Task Graphs Under LogP. *Parallel Computing*, 26(9):1109–1128, July 2000.

[96] I. Karlin, J. Keasler, and R. Neely. Lulesh 2.0 Updates and Changes. Technical report, Lawrence Livermore National Laboratory, Livermore, CA, August 2013.

[97] G. Karypis and V. Kumar. A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95, January 1998.

[98] W. Kellar, A. Savill, and W. Dawes. Integrated CAD/CFD Visualisation of a Generic Formula 1 Car Front Wheel Flowfield. In *Proceedings of the 7th International Conference on High-Performance Computing and Networking 1999 (HPCN'99)*, pages 90–98, Amsterdam, The Netherlands, April 1999. Springer, Berlin, Germany.

[99] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive Performance and Scalability Modeling of a Large-Scale Application. In *Proceedings of Supercomputing 2001 (SC'01)*, pages 37–37, Denver, CO, November 2001. ACM, New York, NY.

[100] D. J. Kerbyson, A. Hoisie, and H. J. Wasserman. A Comparison Between the Earth Simulator and AlphaServer Systems Using Predictive Application Performance Models. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium 2003 (IPDPS'03)*, page 64, Nice, France, April 2003. IEEE Computer Society, Los Alamitos, CA.

[101] D. J. Kerbyson, A. Hoisie, and H. J. Wasserman. Use of Predictive Performance Modeling During Large-Scale System Installation. *Parallel Processing Letters*, 15(4):387–395, December 2005.

[102] W. Kieffer, S. Moujaes, and N. Armbya. CFD Study of Section Characteristics of Formula Mazda Race Car Wings. *Mathematical and Computer Modelling*, 43(11):1275–1287, June 2006.

[103] N. Kurd, M. Chowdhury, E. Burton, T. P. Thomas, C. Mozak, B. Boswell, P. Mosalikanti, M. Neidengard, A. Deval, A. Khanna, et al. Haswell: A

Family of IA 22 Nm Processors. *IEEE Journal of Solid-State Circuits*, 50 (1):49–58, January 2015.

[104] L. Lapworth. HYDRA-CFD: A Framework for Collaborative CFD Development. In *Proceedings of the International Conference on Scientific and Engineering Computation 2004 (IC-SEC'04)*, Singapore, June 2004.

[105] J. Levesque and G. Wagenbreth. *High Performance Computing: Programming and Applications*. Chapman and Hall, London, UK, 2010. ISBN 978-1420077056.

[106] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools With Dynamic Instrumentation. *SIGPLAN Not.*, 40(6): 190–200, June 2005.

[107] R. Lukes, S. Chin, and S. Haake. The Understanding and Development of Cycling Aerodynamics. *Sports Engineering*, 8(2):59–74, December 2005.

[108] R. Lukes, J. Hart, S. Chin, and S. Haake. The Role and Validation of CFD Applied to Cycling. In *Proceedings of the Fluent User Group Meeting*, pages 65–75, Warwick, UK, 2005.

[109] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi. The HPC Challenge (HPCC) Benchmark Suite. In *Proceedings of Supercomputing 2006 (SC'06)*, Tampa, Florida, November 2006. ACM, New York, NY.

[110] A. C. Mallinson, S. A. Jarvis, W. P. Gaudin, and A. J. Herdman. Experiences at Scale With PGAS Versions of a Hydrodynamics Application. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models 2014 (PGAS'14)*, pages 9–20, Eugene, Oregon, October 2014. ACM, New York, NY.

[111] A. D. Malony, S. Shende, and A. Morris. Phase-Based Parallel Performance Profiling. In *Proceedings of Parallel Computing 2005 (ParCo'05)*, pages 203–210, Malaga, Spain, September 2005.

[112] M. Martonosi, A. Gupta, and T. Anderson. MemSpy: Analyzing Memory System Bottlenecks in Programs. *SIGMETRICS Perform. Eval. Rev.*, 20 (1):1–12, June 1992.

[113] M. M. Mathis and D. J. Kerbyson. A General Performance Model of Structured and Unstructured Mesh Particle Transport Computations. *The Journal of Supercomputing*, 34(2):181–199, November 2005.

[114] D. J. Mavriplis. Unstructured-Mesh Discretizations and Solvers for Computational Aerodynamics. *American Institute of Aeronautics and Astronautics Journal*, 46(6):1281–1298, June 2008.

[115] 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, December 1995.

[116] Message Passing Interface Forum. MPI: A Message Passing Interface Standard Version 3.0. `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf` (accessed May 2016), September 2012.

[117] O. E. B. Messer, E. D'Azevedo, J. Hill, W. Joubert, S. Laosooksathit, and A. Tharrington. Developing MiniApps on Modern Platforms Using Multiple Programming Models. In *Proceedings of Cluster Computing 2015 (CLUSTER'15)*, pages 753–759, Chicago, IL, September 2015. IEEE Computer Society, Los Alamitos, CA.

[118] H. Meuer, E. Strohmaier, J. J. Dongarra, and H. Simon. Top 500 Supercomputer Sites. `http://top500.org/` (accessed June, 2nd 2016), June 2016.

[119] P. Moinier, J. Müller, and M. B. Giles. Edge-Based Multigrid and Preconditioning for Hybrid Grids. *AIAA Journal*, 40(10):1945–1953, October 2002.

[120] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics Magazine*, 38(8):114–117, April 1965.

[121] P. J. Mucci, K. London, and J. Thurman. The Cachebench Report. Technical report, Innovative Computing Laboratory University of Tennessee, Knoxville, TN, March 1998.

[122] P. J. Mucci, K. London, and J. Thurman. The BLASBench Report. Technical report, Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, November 1999.

[123] G. R. Mudalige, S. A. Jarvis, D. P. Spooner, and G. R. Nudd. Predictive Performance Analysis of a Parallel Pipelined Synchronous Wavefront Application for Commodity Processor Cluster Systems 2006. In *Proceedings of the IEEE International Conference on Cluster Computing 2006 (CLUSTER'06)*, pages 1–12, Barcelona, Spain, September 2006. IEEE Computer Society, Los Alamitos, CA.

[124] G. R. Mudalige, M. K. Vernon, and S. A. Jarvis. A Plug-And-Play Model for Evaluating Wavefront Computations on Parallel Architectures. In *Proceedings of the 22nd International Parallel and Distributed Processing Symposium 2008 (IPDPS'08)*, pages 1–14, Miami, Florida, April 2008. IEEE Computer Society, Los Alamitos, CA.

[125] G. R. Mudalige, S. D. Hammond, J. A. Smith, and S. A. Jarvis. Predictive Analysis and Optimisation of Pipelined Wavefront Computations. In *Proceedings of the Workshop on Advances in Parallel and Distributed Computational Models 2009 (APDCM'09)*, pages 1–8, Rome, Italy, May 2009. IEEE Computer Society, Los Alamitos, CA.

[126] G. R. Mudalige, M. B. Giles, C. Bertolli, and P. H. Kelly. Predictive Modeling and Analysis of OP2 on Distributed Memory GPU Clusters. *SIGMETRICS Performance Evaluation Review*, 40(2):61–67, October 2012.

[127] National Aeronautics and Space Administration. Euler Equations. `https://www.grc.nasa.gov/www/k-12/airplane/eulereqs.html` (accessed May 2016), September 2016.

[128] U. Naumann and O. Schenk. *Combinatorial Scientific Computing*. Chapman and Hall, London, UK, 1st edition, 2012. ISBN 1439827354, 9781439827352.

[129] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[130] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox. PACEA Toolset for the Performance Prediction of Parallel and Distributed Systems. *International Journal of High Performance Computing Applications*, 14(3):228–251, August 2000.

[131] P. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011. ISBN 978-0123742605.

[132] S. Pakin. Byfl: Analysis of Low-Level Application Characteristics. Technical report, Los Alamos National Laboratory, Los Alamos, NM, February 2011.

[133] S. J. Pennycook, S. D. Hammond, G. R. Mudalige, S. A. Wright, and S. A. Jarvis. On the Acceleration of Wavefront Applications Using Distributed Many-Core Architectures. *The Computer Journal*, 55(2):138–153, February 2012.

[134] 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 Parallel and Distributed Processing Symposium 2013 (IPDPS'13)*, pages 1085–1097, Boston, MA, May 2013. IEEE Computer Society, Los Alamitos, CA.

[135] O. F. J. Perks, D. A. Beckingsale, S. D. Hammond, I. Miller, J. 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, February 2013.

[136] F. Petrini, D. J. Kerbyson, and S. Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Proceedings of Supercomputing 2003 (SC'03)*, pages 55–55, Phoenix, AZ, November 2003. IEEE Computer Society, Los Alamitos, CA.

[137] S. Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of Computational Physics*, 117(1):1–19, March 1995.

[138] D. Quinlan. ROSE: Compiler Support for Object-Oriented Frameworks. *Parallel Processing Letters*, 10(02n03):215–226, June 2000.

[139] B. R. Rau and J. A. Fisher. Instruction-Level Parallel Processing: History, Overview, and Perspective. *The Journal of Supercomputing*, 7(1-2):9–50, May 1993.

[140] I. Z. Reguly, G. R. Mudalige, and M. B. Giles. Design and Development of Domain Specific Active Libraries With Proxy Applications. In *Proceedings of Cluster Computing 2015 (CLUSTER'15)*, pages 738–745, Chicago, IL, September 2015. IEEE Computer Society, Los Alamitos, CA.

[141] L. Reid and R. D. Moore. Design and Overall Performance of Four Highly Loaded, High Speed Inlet Stages for an Advanced High-Pressure-Ratio Core Compressor. Technical Report NASA TP 1337, NASA Lewis Research Center, Cleveland, OH, October 1987.

[142] R. Reussner, P. Sanders, and J. L. Träff. SKaMPI: A Comprehensive Benchmark for Public Benchmarking of MPI. *Scientific Programming*, 10 (1):55–65, April 2002.

[143] R. Reussner, P. Sanders, L. Prechelt, and M. Müller. SKaMPI: A Detailed, Accurate MPI Benchmark. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1497 of *Lecture Notes in Computer Science*, pages 52–59. Springer, Berlin, Germany, June 2006.

[144] A. D. Robison and R. E. Johnson. Three Layer Cake for Shared-Memory Programming. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns 2010 (ParaPLoP'10)*, page 5, Carefree, AZ, March 2010. ACM, New York, NY.

[145] R. M. Russell. The CRAY-1 Computer System. *Communications of the ACM*, 21(1):63–72, January 1978.

[146] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications? *SIGARCH Computer Architecture News*, 40(3):440–451, June 2012.

[147] D. M. Schuster. NASA Perspective on Requirements for Development of Advanced Methods Predicting Unsteady Aerodynamics and Aeroelasticity. Technical Report 20080018644, NASA, Langley Research Center, May 2008.

[148] J. P. Shen and M. H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. Waveland Press, Long Grove, IL, 2013. ISBN 1478607831.

[149] S. S. Shende and A. D. Malony. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications*, 20 (2):287–311, May 2006.

[150] D. B. Skillicorn, J. Hill, and W. F. McColl. Questions and Answers About BSP. *Scientific Programming*, 6(3):249–274, March 1997.

[151] D. P. Spooner, S. A. Jarvis, J. Cao, S. Saini, and G. R. Nudd. Local Grid Scheduling Techniques Using Performance Prediction. *IEEE Proceedings – Computers and Digital Techniques*, 2(150):87–96, April 2003.

[152] V. E. Taylor and B. Nour-Omid. A Study of the Factorization Fill-In for a Parallel Implementation of the Finite Element Method. *International Journal for Numerical Methods in Engineering*, 37(22):3809–3823, November 1994.

[153] X. Tian, A. Bik, M. Girkar, P. Grey, H. Saito, and E. Su. Intel® OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance. *Intel Technology Journal*, 6(1):36–46, February 2002.

[154] M. M. Tikir, M. A. Laurenzano, L. Carrington, and A. Snavely. PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications. In *Proceedings of the European Conference on Parallel Processing 2009 (Euro-Par'09)*, pages 135–148, Delft, The Netherlands, August 2009. Springer, Berlin, Germany.

[155] A. Tiskin. The Bulk-Synchronous Parallel Random Access Machine. *Theoretical Computer Science*, 196(1):109–130, April 1998.

[156] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz. XSBench-the Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *Proceedings of the Role of Reactor Physics Toward a Sustainable Future 2014 (PHYSOR'14)*, pages 1–12, Kyoto, Japan, September 2014. Taylor and Francis, Oxford, UK.

[157] U. Trottenberg, C. W. Oosterlee, and A. Schuller. *Multigrid*. Elsevier, Amsterdam, The Netherlands, 2001. ISBN 978-0127010700.

[158] UK Mini-App Consortium. UK Mini-App Consortium. `http://uk-mac.github.io/papers.html` (accessed March 6, 2016), 2016.

[159] K. D. Underwood, M. Levenhagen, and A. Rodrigues. Simulating Red Storm: Challenges and Successes in Building a System Simulation. In *Proceedings of the 21st International Parallel and Distributed Processing Symposium 2007 (IPDPS'07)*, pages 1–10, Miami, FL, March 2007. IEEE Computer Society, Los Alamitos, CA.

[160] University of Maryland, University of Wisconsin Madison. Dyninst: An Application Program Interface (API) for Runtime Code Generation. `http://www.dyninst.org` (accessed June, 15th 2016), 2016.

[161] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[162] H. Versteeg and W. Malalasekera. *An Introduction to Computational Fluid Dynamics: The Finite Volume Method (2nd Edition)*. Pearson, London, UK, 2007. ISBN 0131274988.

[163] J. F. Wendt. *Computational Fluid Dynamics: An Introduction*. Springer, Berlin, Germany, New York City, New York, March 2013. ISBN 978-3-540-85056-4.

[164] R. D. Williams. Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations. *Concurrency: Practice and Experience*, 3(5):457–481, October 1991.

[165] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76, April 2009.

[166] M. Wu, J. L. Bentz, F. Peng, M. Sosonkina, M. S. Gordon, and R. Kendall. Integrating Performance Tools With Large-Scale Scientific Software. In *Proceedings of the 21st International Parallel and Distributed Processing*

Symposium 2007 (IPDPS'07), pages 1–8, Miami, FL, March 2007. IEEE Computer Society, Los Alamitos, CA.

[167] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, February 1995.

[168] R. Yang, J. Antony, P. P. Janes, and A. P. Rendell. Memory and Thread Placement Effects as a Function of Cache Usage: A Study of the Gaussian Chemistry Code on the SunFire X4600 M2. In *Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks 2008 (I-Span'08)*, pages 31–36, Miami, FL, April 2008. IEEE Computer Society, Los Alamitos, CA.