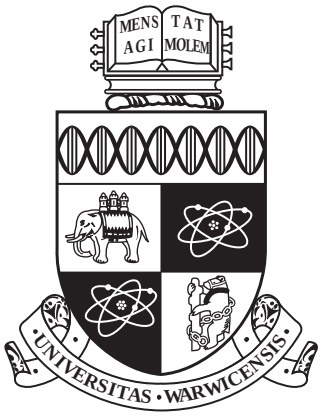


Analytical Modelling for the Performance Prediction and Optimisation of Near-Neighbour Structured Grid Hydrodynamics

by

James Alfred Davis

A thesis submitted to The University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

Doctor of Philosophy

Department of Computer Science

The University of Warwick

May 2017

Abstract

The advent of modern High Performance Computing (HPC) has facilitated the use of powerful supercomputing machines that have become the backbone of data analysis and simulation. With such a variety of software and hardware available today, understanding how well such machines can perform is key for both efficient use and future planning. With significant costs and multi-year turn-around times, procurement of a new HPC architecture can be a significant undertaking.

In this work, we introduce one such measure to capture the performance of such machines – analytical performance models. These models provide a mathematical representation of the behaviour of an application in the context of how its various components perform for an architecture. By parameterising its workload in such a way that the time taken to compute can be described in relation to one or more benchmarkable statistics, this allows for a reuseable representation of an application that can be applied to multiple architectures.

This work goes on to introduce one such benchmark of interest, Hydra. Hydra is a benchmark 3D Eulerian structured mesh hydrocode implemented in Fortran, with which the explosive compression of materials, shock waves, and the behaviour of materials at the interface between components can be investigated. We assess its scaling behaviour and use this knowledge to construct a performance model that accurately predicts the runtime to within 15% across three separate machines, each with its own distinct characteristics. Further, this work goes on to explore various optimisation techniques, some of which see a marked speedup in the overall walltime of the application. Finally, another software application of interest with similar behaviour patterns, PETSc, is examined to demonstrate how different applications can exhibit similar modellable patterns.

Acknowledgements

I owe a great appreciation of thanks to many people for their support, advice and friendship during my time as a postgraduate student at the University of Warwick.

Foremost, I would like to thank my research supervisor, Professor Stephen Jarvis, whose insight and contributions proved invaluable and without whom none of this would have been possible.

I would also like to acknowledge both past and present members of the High Performance and Scientific Computing Group in the Department of Computer Science at the University of Warwick, as well as other friends for their support In particular, I would like to thank Steven Wright, Simon Hammond, Gihan Mudalige, John Pennycook, Oliver Perks, Richard Bunt, Robert Bird, Peter Coetzee, Faiz Sayid, Matthew Leeke, Adam Chester, Stephen Roberts, Timothy Law, Andrew Mallinson, David Beckingsale and James Dickson. In addition, thanks must be given to the administrative staff of the department, including Dr Christine Leigh, Catherine Pillet, Dr Roger Packwood, Paul Williamson, Richard Cunningham, Sharon Howard and the secretarial staff whose aid behind the scenes has always provided invaluable.

Thanks must be given to those at the Atomic Weapons Establishment (AWE) for their support in not only resources but also knowledge, including Andy Herdman, Wayne Gaudin, Wadud Miah, Ash Vadgama, Dr Iain Miller and Dr Satheese Maheswaran.

Further general thanks must also go to the institutions of the Centre for Scientific Computing (CSC) at the University of Warwick, Lawrence Livermore National Laboratory (LLNL), Edinburgh Parallel Computing Centre (EPCC), STFC Daresbury and Nag Ltd for the access provided to HPC architectures used within this work, as well as the people behind the scenes facilitating access

to these resources.

Finally, I would like to give thanks to my parents and sister, whose support, presence and understanding has always been keenly felt.

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 below:

- The compilation of the Hydra benchmark and the collection of benchmark execution data on the Hera and DawnDev architectures was performed by Simon Hammond. Data processing and analysis was performed by the author.

Parts of this thesis have been published by the author:

- Davis, J. and Mudalige, G. and Hammond, S. and Herdman, J. and Miller, I. and Jarvis, S., *Predictive Analysis of a Hydrodynamics Application on Large-Scale CMP clusters*, International Supercomputing Conference (ISC11), Computer Science — Reseach and Development, 26(3–4):175-185, June 2011.

Other research works associated with the author but not used within this thesis are:

- *To upgrade or not to upgrade? Catamount vs Cray Linux Environment*, Hammond, S.D. and Mudalige, G.R. and Smith, J.A. and Davis, J.A. and Jarvis, S.A. and Holt, J. and Miller, I. and Herdman J.A. and Vadgama, A., Large Scale Parallel Processing Workshop 2010 (LSPP10) held in conjunction with IPDPS 2010

Sponsorship and Grants

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

- The Engineering and Physical Sciences Research Council (EPSRC) (2009-2012).
- Access to the Hydra Benchmark was provided by the United Kingdom Atomic Weapons Establishment (AWE) under grants CDK0660 (*The Production of Predictive Models for Future Computing Requirements*) and CDK0724 (*AWE Technical Outreach Program*)
- Use of computing resources was provided by the Centre for Scientific Research (CSC) at the University of Warwick under the Science Research Investment Fund and Joint Research Equipment Initiative under grant JR00WASTEQ.
- Further compute resources were provided by the Lawrence Livermore National Laboratory (LLNL) which is supported by the Office of Science of the United States Department of Energy (DoE), contract DE-AC52-07NA27344.
- Finally, use of the HECToR computing resources is managed by the EPSRC on behalf of the UK Research Councils.

Abbreviations

AMG	Algebraic Multi-Grid
API	Application Program Interface
AVX	Advanced Vector Instructions
AWE	Atomic Weapons Establishment
BSP	Bulk Synchronous Parallel
CAF	Co-array Fortran
CFD	Computational Fluid Dynamics
CMP	Chip Multi-Processor
CPU	Central Processing Unit
CG	Conjugate Gradient
CRCW	Concurrent Read, Concurrent Write
CREW	Concurrent Read, Exclusive Write
CSC	Center for Scientific Computing
CSR	Compressed Sparse Row
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
DPOP	Double Precision Floating Point Operation
EPCC	Edinburgh Parallel Computing Centre
ERCW	Exclusive Read, Concurrent Write

EREW	Exclusive Read, Exclusive Write
FLOP	Floating-Point Operation
FLOP/s	Floating-Point Operations per Second
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
HDL	Hardware Description Language
HPC	High Performance Computing
HYPRE	Parallel High Performance Preconditioners
HPL	High Performance LINPACK
IBM	International Business Machines
ILP	Instruction Level Parallelism
IMB	Intel MPI Benchmark
I/O	Input/Output
IPC	Instructions Per Cycle
LANL	Los Alamos National Laboratory
LLNL	Lawrence Livermore National Laboratory
MIC	Many Integrated Core
MIMD	Multiple Instruction, Multiple Data
MISD	Multiple Instruction, Single Data
ML	MultiLevel Preconditioning Package
MPI	Message Passing Interface

PAPI Performance Application Programming Interface

PCIe Peripheral Component Interconnect Express

PE Processing Element

PETSc Portable, Extensible Toolkit for Scientific Computing

PMPI MPI Profiling Interface

PMTM Performance and Modelling Timing Interface

PRACE Partnership for Advanced Computing in Europe

PRAM Parallel Random Access Machine

PPE Power Processing Element

PVM Parallel Virtual Machine

RAM Random Access Memory

RWM Read/Write/Modify

SIMD Single Instruction, Multiple Data

SISD Single Instruction, Single Data

SMP Shared Memory Parallelism

SoA Structure-of-Arrays

SPE Synergistic Processing Element

SPMD Single Program Multiple Data

SPOOLES Sparse Object Oriented Linear Equations Solver

SSE Streaming SIMD Extensions

UPC Unified Parallel C

VECOP Vector Operation

Contents

Listings

List of Figures

List of Tables

CHAPTER 1

Introduction

The advent of modern computing has unveiled a wide array of potential for modern science. Enabling fast computation on a grand scale, it facilitates the use of new techniques that enhance and compliment traditional scientific practices within multiple disciplines. Simulations, mathematical models that parameterise and capture the behaviour of real-world systems, constitute one such tool; used in tandem with more traditional empirical investigations, they have applications across a wide range of domains such as biology [47, 101, 145], chemistry/physics [87, 91] and weather prediction [26]. In doing so, they have become a driving force for the advancement of supercomputing, fueling demand for ever-more powerful machines.

As part of a scientific or industrial workflow, the power of these High Performance Computing (HPC) architectures has become intrinsically tied to the yield of both simulation and data analysis, not only in achieving faster results [146] but also in enabling more complex, refined simulations that were previously unattainable due to the time prohibitive nature of their execution. Across the course of long running executions, even a minor boost can result in significant time savings. The effective use of these machines has thus become the primary drive in HPC at all levels, from machine procurement and configuration to performance optimisation and prediction. The field of HPC has developed around these concepts, focusing upon both improving existing architectures as well as looking ahead, predicting and planning for the architectures of the future.

With an ongoing push towards the major milestone of *Exascale* computing [48], the adoption of more novel architectures such as accelerators/co-processors in conjunction with an ever-increasing core count and a heavier reliance on the

importance of the network has resulted in an increase in overall machine complexity, making understanding the behaviour of a machine more crucial than ever. The use of a significant multitude of algorithms across the domains results in a variety of different unknown demands upon these machines; ensuring a high degree of efficiency promises to only become more difficult without guidance on their usage. These HPC machines now represent a significant expense, both in their initial procurement and in ongoing maintenance costs; achieving a high throughput thus becomes neccessary to ensure a strong return from these investments.

One approach towards achieving this goal has been the use of *performance models*, constructs that aim to capture the key characteristics of a system and algorithms in order to enable the prediction of their performance without the hardware and/or time required to execute a real-time execution of the algorithm in question. The work in this thesis represents the result of research into one such approach, exploring the use of analytical modelling to capture the behaviour of near-neighbour communication, structured grid applications. In particular, this work focuses on Hydra, a key benchmark provided by the Atomic Weapons Establishment (AWE) that is representative of a real-world application, constructing the first performance model that is able to accuractely describe its behaviour. It is shown how such a model can be used alongside an understanding of the application to identify and optimise bottlenecks, exploring a multitude of potential opportunities for enhancement. Further, this thesis explores the applicability of such an approach to the Conjugate Gradient (CG) linear solver within the Portable, Extensible Toolkit for Scientific Computing (PETSc), a popular project with a different purpose/function but a demonstrably similar behaviour in its implementation, showing how such techniques can be applied on an application to application basis. In doing so, these methods can aid both science and industry in preparing for the many-core architectures of the future.

1.1 Motivation

Commenting on what he believed the future of computing hardware would achieve, in 1965 Gordon E. Moore observed a trend that would come to dominate the depiction of computing performance in the decades that followed:

> *The complexity for minimum component costs has increased at a rate of roughly a factor of two per year... Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years.* ***Gordon E. Moore*** [118]

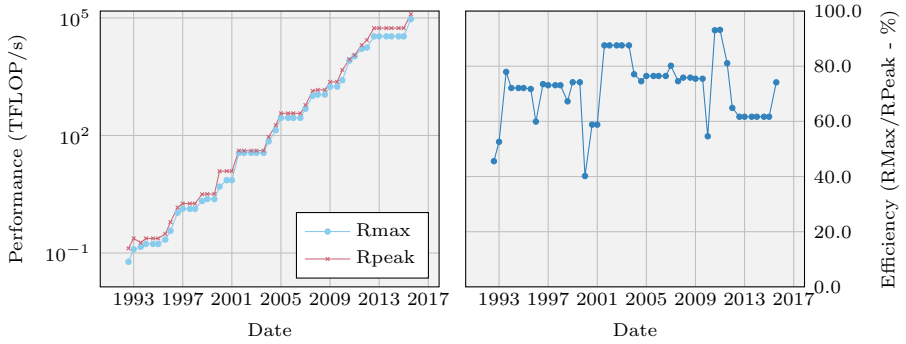
Moore's observation (known as **Moore's Law**) resulted in the prediction that the future trend of the Central Processing Unit (CPU) transistor density was to see an exponential growth, doubling approximately every two years. This trend has typically been matched by an increase in the *performance* of a chip [25, 136].

In the early period following Moore's publication, the most apparent outcome of this was an improvement in the clock speed of the CPU. In conjunction with Pollack's Rule [25], which states "performance increases (when not limited by other parts of the system) as the square root of the number of transistors or area of a processor", this has traditionally implied an overall improvement in the performance of a chip as the transistor density improves. In the past this improvement has previously offered easily accessable gains for serial applications with few to no changes required on the part of code maintainers, useful for developers dealing with large or complex codes where refactoring and optimisation of such applications would require significant developmental resources. This ultimately led to the period being described as a "free lunch" [164], yet it was inevitable that such gains were unsustainable.

The physical consequences of increasing the tranistor density on a core have proven to be an impediment to making such an approach permanently viable; notwithstanding that the size of a transistor must ultimately be bound by a

physical lower limit, voltage leakage [25] and heat generation [21, 96] threaten to become prohibitive to efforts to improve CPU technology. Previously "easy" gains in clock speed have become difficult to maintain, or are at the very least no longer cost-effective [164], culminating in a paradigm shift towards *concurrency*. Rather than making a single CPU core faster, tasks are distributed amongst multiple computing entities to allow for their execution in parallel [165]. The increase in transistor count on a chip has continued to observe Moore's Law for the present, but is no longer achieved through transistor density on a single core of a CPU; rather multiple cores on a chip are now employed instead, in some cases even resulting in an intentionally slower clock speed to provision for heat, power or space requirements. Extending this concept past a single CPU, work can be spread across multiple multi-core chips, installed in distributed machines (nodes) that are physically separated but can communicate via some form of network interconnect. This has resulted in the modern HPC field being dominated by large scale, multi-core, multi-node cluster/grid supercomputing systems that now handle the significant majority of the community's workload, typified by the concept of Beowulf clusters [159], with an additional shift towards accelerator-based computing (see Section 2.2).

Due to the increasingly demanding requirements of modern simulations, there is an ever-growing dependence upon the use of these modern supercomputers. Their use has become the focus of a significant amount of research in both industry and academia, addressing not only the development of efficient parallel algorithms but also the implementation of new architectures or hardware configurations, exploring what opportunities are available to increase the scientific yield of such machines. The Top500 [84, 161] is dedicated to documenting trends in the advancement of such machines, maintaining a bi-annually refreshed report of the LINPACK [50] benchmark on the highest rated HPC machines. Reporting both the practical (Rmax) and theoretical (Rpeak) peak number of Floating-Point Operations per Second (FLOP/s), the rankings aim to provide a relatively simple means of comparison for the maximum capabilities of these



(a) Linpack Performance (b) Machine Efficiency

Figure 1.1: Top 500 Machine Performance Trends [161]

machines that is both historical and current in nature. During the past 23 years a substantial change can be observed, from an initial 0.053 TFLOP/s in 1993 to the first reported Petascale machine, RoadRunner, in 2008 that heralded a new landmark in sustained performance and beyond. As of June 2016 the highest FLOP/s result (as reported by LINPACK) sits at a substantial 93.01 PFLOP/s (Figure 1.1(a)), with efforts now ongoing towards achieving the next major milestone — *Exascale computing* [20].

The use of the LINPACK benchmark as a simple means of comparing the computational capacity of HPC machines has proven to be a useful one. However, the scientific/industrial community at large has a wide-range of potential applications for these computing resources, with no guarantee that any two codes exhibit the same underlying performance characteristics; as a consequence, they can make different demands of the underlying hardware. A strong LINPACK performance does not mean that another application will achieve a similar efficiency, and from Figure 1.1(b) it can be seen that even LINPACK (Rmax) does not achieve 100% efficiency for numerous architectures when contrasted against a machine's theoretical peak (Rpeak). Identifying the performance characteristics of a particular workload is thus crucial when selecting an architecture; it can greatly boost the scientific throughput of a machine if it is particularly attuned to the demands of an application. Such understanding also

enables the optimisation of applications through the identification of notable bottlenecks.

While both clusters and grids are capable of executing HPC workloads, the predominant architecture used in HPC is that of cluster machines. As evidenced by how heavily they are represented in the Top500, they possess characteristics that are more suited towards the features of such workloads, specifically:

- Clusters are often more homogeneous in their node hardware, as opposed to grid systems which may use a multitude of differing compute devices. Since parallel codes can often operate at the speed of the slowest compute device (due to blocking communication behaviours), the use of similar hardware can prevent any need for extra oversight in the assignment and decomposition of data to prevent load imbalance.

- Unlike cluster machines, the nodes of grid computers are typically more geographically dispersed, resulting in the use of slower interconnects such as the internet rather than the faster network interconnects often utilised by cluster machines.

This is not to say however that there are not also complexities to the use of cluster architectures. The modern HPC cluster can possess a number of different architectural components, each of which can have complex interactions with one another that impact upon the overall performance of an application. As well as the underlying CPU performance, the data-processing throughput of many applications can put great demands upon the memory bandwidth and/or latency of a system. The difference in parity between the advancement of CPU performance and memory performance has given rise to a problem known as the "memory-wall" [178], where systems are often becoming more performance bound by the bandwidth and latency of memory rather than the maximum computational throughput a system is capable of. Further, the nature of parallel compute means that the neccessity of data communication between remotely distributed compute nodes introduces an additional overhead. The throughput,

both bandwidth and latency, of the network interconnect becomes crucial to ensuring speedy transmission of such data. A wide range of different potential network topologies can mean that the effective distribution of parallel work is crucial to minimising these overheads.

Finally, more recent advancements in the HPC domain have seen the introduction of accelerators such as the IBM Cell accelerator [85], General Purpose Graphics Processing Units (GPGPUs) using CUDA [134, 131]/OpenCL [125, 160] and Intel's Xeon Phi architecture [38, 139], add-on components that seek to enhance the parallel compute performance of individual nodes, marking a shift to a more hybrid/hetrogeneous style of HPC where multiple different compute hardware elements are available. At the time of writing many of the highest rated Top-500 machines exploit such hardware indicating that this is a trend that is unlikely to change in the immediate future. While not all codes yet use such technologies, understanding and designing for hybrid systems early in the development cycle can mitigate the cost of significant re-engineering efforts later in an application's lifecycle.

It is in this context that understanding the software and hardware that underpins modern HPC architectures has become crucial to the effective use of resources. The complexity of modern HPC architectures increases the risk of introducing major performance bottlenecks, while the cost and time required to procure, operate, and maintain such machines makes the impact of an unsuitable/inefficient machine significant. Selecting the most appropriate machine during the procurement phase is paramount to its longevity and usefulness during its lifetime. The use of performance models provides a means by which a user can assess their workload on a HPC machine when active or potentially even prior to its purchase. Other works have demonstrated the use of performance models not only in their capacity for predicting performance runtime [32, 71, 72, 90, 107, 108, 122, 123, 124], but also in their ability to aid in the procurement and configuration of HPC machines [74, 89]. The trend of ever-increasing core counts promises to introduce new and potentially unfore-

seen complexity to maintaining high throughput, especially given the variety of workloads/applications of interest within the realm of academia and industry. Providing a means to explore performance without having the full hardware available, these models enable the user to explore not only alternate configuration on existing hardware, but also to explore the domain of future architectures.

Of the various scientific domains, hydrodynamics applications fall into one such class of codes of interest, representing a significant part of the HPC workload at organisations such as AWE in the UK and the U.S. national laboratories. For this reason, benchmark codes representative of these applications, such as SAGE from the Los Alamos National Laboratory (LANL) [90] and Hydra from AWE [44], provide a key tool for evaluating HPC systems during design, procurement, installation and maintenance. The development of such HPC codes, the evaluation of their performance on candidate systems and sustaining performant execution is a costly and time consuming exercise. To aid in these activities, much academic research has been conducted into developing accurate performance modelling tools and techniques for application analysis [75, 90, 109, 124, 163].

The subject of this thesis is the use of predictive models to capture an understanding of performance and use this knowledge to explore potential optimisation opportunities that may exist. The core focus of this thesis is based around Hydra, a high-performance hydrodynamics benchmark developed and maintained by AWE. The developed model elucidates the parallel computation of Hydra, with which it is possible to predict its run-time and scaling performance on varying large-scale Chip Multi-Processor (CMP) clusters. A key feature of the model is its granularity; the model is able to separate the contributing costs, including computation, point-to-point communications, collectives, message buffering and message synchronisation. We also explore how these techniques can be portable to other applications of interest such as PETSc, a linear solver library commonly in use among a number of scientific applications [69, 77, 86].

The aims and objectives of this work are to demonstrate how the use of performance models can aid in the development and execution of HPC applications in a fast-changing environment. The rapid development of new architectures and programming paradigms, combined with the frequent turnover of HPC architectures in favour of more modern hardware, leads to a scenario where it is crucial to not only understand the existing performance constraints of an application, but to also be aware of future hindrances that might prevent a developer from taking full advantage of new advances. Through the use of performance models, it is possible to not only develop a strong understanding of existing performance hotspots, but also to permit the adjustment of hardware parameters or to investigate the impact on overall performance when modifying a subset of the application. This can have uses in not only the development and optimisation of a code, but also during machine procurement where projections can aid in the decision making process. The development of one such performance model in this work is intended to show how such models can be constructed and applied, highlighting their potential use as part of a HPC workflow.

1.2 Thesis Contributions

This thesis makes the following contributions:

Contribution 1: An initial performance analysis of Hydra

A strong and weak scaling study is used to identify the key performance influencing characteristics of Hydra, a structured-grid hydrodynamics benchmark. Seperately distinguishing these key contributors of performance into compute, collective communications and near-neighbour data exchange communications, it is revealed how different machine characteristics can influence the application's parallel behaviour. In addition, areas of unusual behaviour are identified for further study, showing how a performance model could guide further investigations.

Contribution 2: Construction and validation of an analytical performance model of Hydra

Building upon this empirical knowledge of Hydra, a performance model is constructed that enables the prediction of run-time performance to within 15% of error at scale. As well as establishing the compute kernel behaviour, communication patterns are also captured, modelling both intra- and inter-node communications, as well as the influence of an increasing process count and synchronisation upon collective MPI operations. This granular model enables the exploration of various characteristics upon performance, changing not only the configuration parameters but also a machines performance metrics , enabling model-led investigation of alternate runtime environments. This can lead to more accurate assessment of such machines during procurement, ensuring that they meet the demands placed upon them during their operation, as well as highlighting unusual behaviour in the benchmarks performance when contrasted against model predicted outcomes.

Contribution 3: Optimisation of the Hydra benchmark

Following from the observed behaviour, there exist a number of deviations from what might be expected of model-predicted performance of some kernels. With the knowledge provided by both the model and the scaling investigations, potential optimisations are explored to both correct and improve upon the existing benchmark; this targets three machine linked characteristics of interest — compute performance, memory access patterns and compute/communication overlap. Further, demonstrating how the model can be used to predict configuration changes, the impact of modifying the number of compute cores used per node is explored. Such improvements can lead to improved performance not only for existing hardware but potentially also across future hardware, showing how the upfront cost of performance analysis and modelling can help reap benefits for future operation.

Contribution 4: Linear solver analysis

It is expected that, within the scientific domain, interest will exist for other benchmarks beyond that of a limited sample. As such, performance modelling techniques must have some degree of portability in their implementation or usage. To this end, the performance of PETSc, a linear solver library that is an integral part of many other scientific benchmarks, is investigated, focusing upon the CG linear solver algorithm. While the purpose of PETSc may differ, the underlying compute and communication behaviour of the CG algorithm exhibit many similarities with Hydra, showing how the performance modelling techniques used within this thesis could be applicable to other applications.

1.3 Thesis Overview

The remainder of this thesis is structured as follows:

Chapter 2 provides a detailed theoretical background of the basic concepts and techniques employed by the HPC community in the fields of performance analysis, engineering and modelling. In particular, it focuses upon current techniques for effective parallelisation, the theoretical laws that govern the performance of parallel algorithms and the tools used to achieve these goals.

Chapter 3 details the experiment setup of the investigations undertaken within this thesis; specifically, the machines, tools, libraries and software configurations used to obtain empirical data. A selection of benchmark results for base machine parameters such as memory or network interconnect performance are also included where available.

Chapter 4 introduces Hydra, a benchmark Hydrodynamics application provided by AWE. Used as a case study herein, it is a representative benchmark of a workload of interest, and contains characteristics that are exhibited by other scientific applications of interest in the HPC domain. We explore the performance of the current implementation and identify a number of areas for further investigation — in particular underperforming kernels and the impact of the machine's hardware metrics.

Chapter 5 expands upon the initial performance analysis work of Chapter 4. The understanding of Hydra's behaviour is used to construct a parallel performance model of Hydra, providing insight into a number of characteristics including compute performance, point-to-point communication patterns, quantity of data communicated and collective behaviour.

Chapter 6 applies our knowledge of Hydra from both performance analysis and performance modelling to investigate a broad range of optimisations, applicable to a variety of potential bottlenecks that can exist in modern HPC architectures. Techniques of interest include memory pattern optimisation, the use of vectorisation, hybrid OpenMP/Message Passing Interface (MPI) execution and message-passing overlap with compute. Machine configuration guided by modelling insights is also explored.

Chapter 7 introduces PETSc, a linear solver library, from which the performance of the CG linear solver algorithm is examined. Demonstrating the use of techniques previously applied to Hydra to capture performance characteristics of interest, similarities in the structure of its parallel implementation to Hydra are identified despite differences in their purpose. By extension, it is shown how similar modelling techniques could be applied to CG, showing an example of the portability of such techniques. Further, the performance of the CG solver is contrasted against an alternative CG variant built into PETSc designed to

reduce the number of collective operations, highlighting the performance impact of collective operations and their effect on scalability.

Chapter 8 concludes this work, providing a summary of the outcomes and outlining any potential future work of interest.

CHAPTER 2

Performance Analysis, Modelling and Optimisation

Within the fields of science and industry, the adoption of parallelism for the purposes of High Performance Computing (HPC) has led to widespread demand for a variety of tools. The complexity of parallelism is such that while the opportunities are great, the development process can be difficult, expensive and time-consuming. As a consequence, the available range of parallel hardware and techniques developed by both academia and industry has matured as the field has grown. However, despite the progress made in developing powerful tools for implementing parallelism, some degree of manual process remains. Even with automated techniques, an application must be designed in such a way that makes it amenable to executing tasks in parallel, with a range of approaches still necessitating direct implementation within the codebase itself. Identifying which parallel techniques are of interest, and understanding how they behave, is crucial during the development process to ensure both *correct* and *performant* code. Switching from a serial to parallel design can introduce a number of potential error vectors, such as race-conditions, lack of data coherency or performance degradation resulting from complex interactions between machine components and the introduction of data commmunication overheads. Addressing this, much work has occurred in the field towards classifying parallel applications, understanding their behaviours and constructing toolchains that can aid in their development.

In this chapter a background of work from the field of HPC is introduced, focusing primarily upon its importance to the performance of an application. Specifically:

- Section 2.1 introduces the core concepts of parallelism, highlighting the

different forms that it can take;

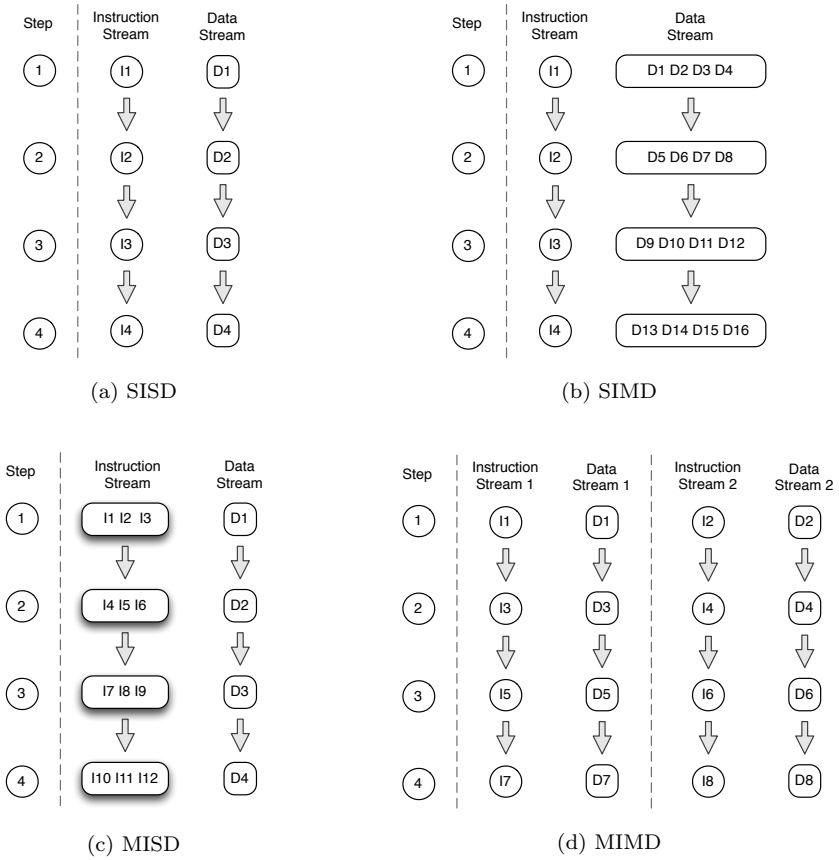
- Section 2.2 describes the critical components that can influence the parallel performance of a machine;
- Section 2.3 introduces the core concepts of performance modelling and analysis. This details the various models and laws used to describe the parallel performance of an application, as well as analytical and simulation techniques that can be used to construct a performance model of an application.

2.1 Forms of Parallelism

Unlike a serial application, which can only conduct operations sequentially, a parallel program can vary in the order of execution across multiple distinct hardware units. Capable of performing two or more operations simultaneously, such programs are potentially able to scale their performance with the introduction of additional compute components and a means to share data between them. However, these advantages often come with a number of constraints that dictate their usage; this complicates their implementation, debugging and performance optimisation. This section introduces a number of parallel concepts, including different algorithmic categories and the various hardware architectures that enable their implementation.

2.1.1 Flynn's Taxonomy — Program Classification

Flynn's Taxonomy [55] describes four different classifications in an effort to better capture the types of parallelism available. The Single Instruction, Single Data (SISD) classification (Figure 2.1(a)) describes the characteristics of a serial application, i.e., one that possesses no form of parallelism. In contrast to this, the three remaining classifications all distinguish between the parallelisation of the instruction stream and of the data stream — Single Instruction, Multiple



(a) SISD

(b) SIMD

(c) MISD

(d) MIMD

Figure 2.1: Flynn's Taxonomy

Data (SIMD); Multiple Instruction, Single Data (MISD); and, Multiple Instruction, Multiple Data (MIMD).

A SIMD program (Figure 2.1(b)) uses parallelism of the data stream but not of the instruction stream. It is capable of applying the same instruction to multiple pieces of data simultaneously, enabling a much greater throughput of data processing, albeit with the restriction that the problem must be amenable to repeating the same operation across a large dataset.

MISD (Figure 2.1(c)) is the opposite of this, parallelising the instruction stream but not the data stream, enabling the execution of multiple operations on the same piece of data. This category sees little use in modern HPC, due to scientific applications typically possessing the more common attribute of repeat-

ing a similar set of instructions across large datasets, for which this approach is not suitable.

MIMD (Figure 2.1(d)) is a combination of these two parallel approaches, enabling the parallelisation of both the instruction and the data stream. This is one of the more common forms of parallelism in HPC today, as it enables the distribution of data across multiple parallel units while each is capable of operating a different instruction stream. Darema [43] extends this classification one step further to introduce the Single Program Multiple Data (SPMD) category. This better describes parallel applications of a Shared Memory Parallelism (SMP) or distributed nature that execute the same program concurrently on various Processing Elements (PEs), resulting in the execution of similar (but distinctly separate) instruction streams asynchronously upon different datasets (or portions thereof). The critical path of such programs is usually similar between processing elements, yet cannot be classified as SIMD due to the potential for variation and there being no guarantee that the same operations on different parallel units are being conducted simultaneously as part of their asynchronous nature. It is this characteristic which adds an additional complexity to understanding the performance of SPMD applications.

2.1.2 Single-Thread Parallelism

A number of hardware designs exist to exploit parallelism within an application at the instruction level; such an approach is classified as Instruction Level Parallelism (ILP) [150]. Examples include:

Instruction Pipelining

Instructions are decomposed into a number of micro-instructions. Different micro-instructions that do not share the use of the same resources can be conducted in parallel. By employing pipelining, micro-instructions from two or more instructions are able to be executed in parallel, permitting the overlap of multiple instructions per cycle.

Superscalar Processors

A superscalar processor is designed with multiple execution resources, permitting the execution of more than one instruction per cycle. This is different from instruction pipelining where only micro-instructions are conducted in parallel. It can be the case that a processor is both superscalar and capable of pipelining.

Out-of-Order Execution

Out of order execution permits, data dependancies withstanding, the execution of instructions in an order different to that of the original program. When an instruction is waiting on data to become available for processing, an action that would normally result in a processor stall, the execution unit can use these idle cycles to execute an alternate, non-dependent instruction instead, mitigating the impact of a potential delay. The results are reordered such that they appear as if they had been executed as per the original program flow.

Vectorisation

Vector units take advantage of vector instructions to implement SIMD level parallelism within a core, enabling the processing of multiple units of data for a single instruction from an instruction stream. Examples include SSE and AVX [54, 79] (Intel/AMD), Altivec [46] (PowerPC) or NEON [9] (ARM).

2.1.3 Shared Memory Parallelism

SMP exploits multi-core architectures, where multiple processing cores exist on the same chip but must share the remaining machine resources such as its main memory. The one notable exception to this is the use of cache memory, where a multi-core processor can possess one or more cache levels unique to each core and a shared cache accessible by any core. SMP can distribute a workload

among the various cores through the use of threading Application Programming Interfaces (APIs) such as POSIX threads [99, 130] or OpenMP [42] to allocate at least one work-thread to a core. Approaches such as Intel's Hyper-Threading Technology [93, 106] support the creation of two virtual/logical threads per physical core, but still share the resources of a core among its bound threads. This enables one thread to exploit resources underused by another thread such as when stalling occurs.

The benefits of SMP are readily apparent; by distributing and executing a workload in parallel significant speedups can be obtained. However there are a number of restrictions upon the use of this form of parallelism. Use of libraries such as OpenMP permits the automation of the threading process to a degree, but a developer must still ensure that suitable regions of code are parallelisable, with no data-dependancies that impede the parallelisation process.

In addition to the mechanics of its implementation, ensuring *performant* threading has a set of additional requirements for consideration. Algorithms must have their workload decomposed and distributed in a manner that results in a reasonable workload balance, such that a single core is not required to perform a significantly greater amount of work than any other core. Further to this, the creation of a thread and allocation of work can be expensive, thus sufficient work must be made available for it to be worthwhile.

The use of shared memory also introduces the issues of concurrency, race conditions and cache-coherency, where the order of threads accessing memory can result in different output if not handled correctly. Critical regions can combat issues of concurrency, but introduce a serial bottleneck into a parallel application, as well as the overhead of locking and unlocking a region of code. Finally the use of shared resources incorporates the problems of contention, that a shared resource such as memory bandwith may reach a saturation point where an execution unit is starved of data — commonly referred to as the *memory wall* [178].

2.1.4 Distributed Memory Parallelism

Distributed memory parallelism is built upon the foundation of SMP; the principle of sharing a workload across multiple PEs remains. However SMP relies upon particular characteristics of the underlying hardware — each compute unit is able to access a shared memory space where the results of other compute units can be stored and accessed freely, for example, via main memory. Since a compute unit may require the results of computation from another compute unit, such access is a neccessity. Distributed architectures however can consist of multiple distinct machines or **nodes**, with no shared resources in common (other than the hardware that comprises the network routing and communication facilities).

The advantage of this is clear — a shared memory unit is limited by the number of available cores that can be placed into a single machine and the saturation of its shared resources. A distributed architecture however, ignoring the limitations of sufficient power, cooling and space, could continue to scale its PE count by continuously adding more and more machines to the overall structure. In doing so they offer a far greater scale of parallelism by increasing the capacity for the overall system to distribute the workload across a larger number of compute units.

A distributed setup does however come with its own set of disadvantages and limitations. Data sharing between distinct nodes requires some form of communication interconnect, the bandwidth and latency of which will typically be slower than the intra-node counterpart. These data exchanges are also typically managed manually by the application due to individual processes being unaware of data stored on remote nodes, yet said data is necessary to prevent the propogation of errors that would otherwise arise. Failure to retrieve this data results in a lack of coherency across the cluster and invalid computation. This introduces a greater complexity in the development and debugging of parallel applications, as it is possible to bring about race conditions when data is incorrectly propogated, or deadlocks where a node is permenantly idle waiting

for the output of another node that never communicates its data. In addition, while the hardware may be able to scale in its quantity/capacity, this does not necessarily translate into equivalent performance, as addressed later in Sections 2.3.1 and 2.3.2 of this chapter.

The need for parallel data communication has resulted in a number of attempts to standardise the API used by parallel applications to provide a consistent and understandable description of how data is handled by the nodes. A number of different approaches have been attempted, including the Parallel Virtual Machine (PVM) [65], Message Passing Interface (MPI) [59, 67] and global address space languages such as Co-array Fortran (CAF) [133] and Unified Parallel C (UPC) [39, 51]. Of these, the MPI standard has seen the most widespread adoption, providing a standard with a wealth of descriptive API functions with well defined outcomes, but leaving the specifics of their implementation to both academia and industry. As such, a variety of different MPI libraries exist, targeted at either providing a general-purpose solution available to the community at large, such as OpenMPI [61] and MPICH [68], or to provide an optimised library attuned to the characteristics of a particular architecture, such as MPI on the BlueGene/L architecture [5].

Within the MPI standard, each unique MPI process is treated as a distinct separate processing element with no shared memory space, regardless of the underlying hardware that they are bound to. As such, data sharing between the MPI processes can only occur through the MPI API — even if two MPI processes both use a different core on the same node, they are unable to see one another's allocated block of memory; all communication occurs through the use of MPI point-to-point communication functions, such as Send or Recv, that communicate directly between two processes, or collective functions, such as MPI AllGather/AllReduce, which communicate from one-to-many, many-to-one or many-to-many.

2.1.5 Accelerators

Recent advances in computing have seen the introduction of a new addition to the field of parallel computing — accelerators. As highly parallel compute devices, they are typically installed alongside a Central Processing Unit (CPU) within a machine to form a hybrid architecture, where work is offloaded in part from the CPU to the accelerator device for processing. They can see a sizeable initial overhead in data transfer, but potentially offer a significant increase in the level of parallelism a machine is capable of. Examples of accelerators that have been introduced to the scientific community, either past or present, include:

Cell Broadband Engine Architecture

The Cell Processor [85], introduced in HPC as a component of Roadrunner [18], contains 9 processing units — a general purpose Power Processing Element (PPE) that acted as a scalar main processing unit and 8 Synergistic Processing Elements (SPEs) that functioned as vector processors to provide the bulk of the Cell's parallel processing power. However, while the chip provided the boost in power required to make Roadrunner the first Petascale architecture in the TOP500 [161], it did not see a widespread adoption among the HPC community. This was in part due to its overall complexity — the PPE acted as the primary device for orchestrating how the SPEs were used, but each SPE possessed its own reserved block of memory from which it could draw instructions and data for processing. This neccesitated the use of explicit Direct Memory Access (DMA) requests for data transfer, placing a significant degree of difficulty in modifying existing codebases for use on the architecture. In addition, the heavily restricted size of SPE memory meant that such requests had to be carefully managed, raising the degree of micro-management required to handle data for processing.

General Purpose Graphics Processing Units

While they originally had their genesis in markets such as computer graph-

ics and videogames, Graphics Processing Units (GPUs) have developed over time to become more generalised, highly parallel compute devices. It is this property which saw them becoming of interest for use in HPC, a feature which Nvidia opened up to the community at large with the introduction of the Compute Unified Device Architecture (CUDA) [134, 131]. CUDA enabled access to these General Purpose Graphics Processing Unit (GPGPU) devices for parallel computing such as that found in the field of HPC, and has since seen a significant rise in their adoption for this purpose, evidenced by their presence in the TOP500 including machines such as Titan [161] among others. As their popularity has risen, a number of other frameworks/standards that are less hardware vendor specific have also been developed, such as OpenCL [125, 160] and OpenACC [137], to facilitate efforts to develop more platform independant support for accelerators into compilers and codebases.

Intel Many Integrated Core

The Intel Many Integrated Core (MIC) architectures, sold currently under the monikor of the Intel Xeon Phi family [38], are co-processors connected over the Peripheral Component Interconnect Express (PCIe) interface containing a significant number of integrated cores, each in turn capable of supporting multiple threads. Borne out of the Intel Larrabee project [154], they are suited for highly parallel tasks due to their weaker but more numerous cores, with multiple investigations demonstrating their potential for compute performance [139]. Like GPGPUs, they have been adopted for use in the field of HPC, as demonstrated by their presence in Tianhe-2 [161].

Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are reconfigurable circuits consisting of an array of logic gates and interconnects that allows for the use of flexible or repurposed designs as demanded. Through the use of Hardware Description Languages (HDLs), they can be repurposed for HPC applica-

tions by exploiting their parallel nature.

2.1.6 High Performance Clusters

Many modern high performance supercomputers adopt a mixture of the forms of parallelism described thus far. Traditionally, the performance of a CPU was closely tied to its clock speed — the rate at which it was able to process instructions. Dennard scaling [45], historically supporting Moore's Law through the trend of improvements in the compute/heat generated per Watt, has struggled in the face of problems that have arisen from extremely small transistor sizes [52]. This is not unexpected, as by the second law of thermodynamics there must be a natural limit to the achievable efficiency of computing, as posited in the work of Landauer [96]. However, this impacts upon the future scalability of a single multi-core machine.

Based on the concept of Beowulf clusters [159], a HPC cluster can consist of multiple distributed, usually homogeneous, nodes linked by a high speed interconnect for data communication in a specific network topological arrangement. Each node can consist of proprietry hardware, but many modern clusters are comprised of commodity hardware, albeit stored in a cabinet/rack setup for maintenance, space and organisational reasons. The nodes often have multi-core CPUs that have a cache and access to a shared memory pool local to the node, permitting the combination of SMP and distributed forms of parallelism across the cluster. With this approach, the total amount of cores can theoretically be scaled infinitely (though this is not neccessarily mirrored in the scalability of an application). It distributes the problem of heat generation on a core, by scaling outward across multiple nodes rather than scaling within a single node. While heat generation still remains a significant problem across a system for such clusters (due to often shared physical location of multiple nodes), it provides a form of scalability that does not suffer from the problem of transistor leakage that accompanies miniaturisation. These setups are supported by the adoption of MPI within the HPC community, applied to a significant portion of

parallel scientific applications in use.

Alternate approaches exist in the form of Grid Computing, similar to cluster based machines in that they consist of a distributed collection of nodes interlinked by some form of communication network. However, they are often more geographically distributed and can consist of a wide range of heterogenous hardware. Neither quality is conducive to high performance computing due to the communication overheads, latencies and poor work load balance across different hardware, resulting in significant delays in synchronisation steps.

Adopting such an abstract overview to HPC machines however would do a disservice to the underlying complexity and work that goes into understanding the impact of component selection and configuration upon the performance of any arbitrary scientific application. Such applications can vary greatly in the demands they make of a system's components due to the variety of potential workloads on offer. Throughout the history of the TOP500 [161] a multitude of different combinations of hardware in a cluster format can be seen. For example, in contrast to machines based on commodity hardware, the BlueGene architectures [1] by International Business Machines (IBM) are exemplified by their selection of a low clock-speed PowerPC architecture, bolstered by their quantity and use of a high-speed interconnect to promote scaling performance over individual node performance. Recent years have also seen the rise of machines incorporating the use of accelerators/co-processors on their nodes, as addressed in Section 2.1.5.

Such a wide array of metrics and hardware alone would be an argument for the complexities of modern HPC architectures. When the additional considerations of the software stack and machine configuration are introduced, it is further apparent that the performance of applications could experience a significant amount of variance. Given the substantial cost and time involved in the procurement process, not to mention the ongoing overheads from power, maintenance, cooling and housing of said machines, it is crucial that the most suitable machine for a target workload is selected. Doing so will increase the

use and return on investment, resulting in a greater overall scientific yield. Understanding and improving upon the behaviour of the dominant workloads of interest thus becomes crucial. Achieving this requires an understanding of both the architecture and of the characteristics of any algorithms of interest.

2.2 Machine Cluster Architecture

A modern supercomputer can possess a wide number of interconnecting components, any of which can contribute towards a performance bottleneck. These include:

The CPU (Section 2.2.1)

Compute bound performance is restricted by the rate at which the CPU is capable of processing instructions.

Cache/Main Memory (Section 2.2.2)

In a memory bound code the cache or main-memory is unable to supply data at a sufficient rate to fully use the available compute resources.

The Network Interconnect (Section 2.2.3)

In distributed architectures the fulfillment of remote data-dependencies requires the transmission/retrieval of data over the network interconnect, introducing a neccesary overhead before any further compute can take place. Such interconnects are typically slower than a node's memory.

The Software Stack (Section 2.2.4)

The selection of an appropriate compiler or compiler options, alongside the MPI implementation, is capable of influencing the underlying performance of the application. This can include factors such as the use of vectorisation (a SIMD facility), the degree of floating-point accuracy (more accurate

is typically slower) or attunement of MPI to use architecture specific features/characteristics.

Understanding the impact of these various components upon a system's performance is crucial to the process of profiling, modelling and optimisation.

2.2.1 The Central Processing Unit

The modern CPU supports a number of features that lend themselves towards modern parallel HPC applications. As well as those techniques identified in Section 2.1.2, provided here is an introduction to some commonly supported CPU characteristics.

Vectorisation

Vectorisation is a single-core parallelisation technique, an implementation of SIMD from Flynn's Taxonomy (Section 2.1.1) that introduces the capacity for multiple pieces of data elements to have the same operation applied to them simultaneously. It is dependent upon a number of criteria, but support is common among modern CPUs since the introduction of Intel's Streaming SIMD Extensions (SSE) instruction set. The repetitive nature of many scientific simulations that process large datasets means that such an approach is often a viable strategy.

Multi-core

Multi-core CPUs are the most common outcome of the adoption of the parallelism paradigm. By incorporating multiple compute cores onto a single chip, each is able to use its own compute resources/cache while accessing a shared memory for communication. A simple example is where each core is assigned one thread with a distinct instruction stream, independant of other threads. Some hardware supports multi-socket as well as multi-core architectures, where a machine can have more than one CPU on the same

motherboard sharing the same memory resource.

Hyper-Threading

Intel's Hyper-Threading technology [93, 106] takes the concept of multiple cores one step further, allowing a single physical core to appear as two logical cores. This enables a single core to support two distinct threads that may, under ideal conditions, use different system resources for an overall speedup. However, since the underlying hardware still consists of one core, hyperthreads are capable of contending for resources and thus care must be taken with their use.

CPU Scaling

The use of CPU scaling recognises that not all algorithms are perfectly parallel. Via the application of dynamic clock speeds, a core that is operating under its power/thermal limits can be placed under greater load by raising its clock frequency higher than one for which it operates by default. This can be used to exploit scenarios where a chip is being underutilised, such as when not all cores are in use [37].

While the opportunities for on-chip parallelism are varied, their use often puts a strain upon a shared memory resource where it becomes a requirement that all compute units are efficiently served to prevent them being starved of data. Thus it is vital that the implementation of any memory subsystem is able to support this constraint. The most common form this takes in many cluster architectures is that of the *Memory Hierarchy*.

2.2.2 The Memory Hierarchy

In modern systems there exist multiple different forms of data storage, each possessing its own set of advantages and disadvantages. These include the use of CPU registers, cache, Random Access Memory (RAM), and hard disks. The

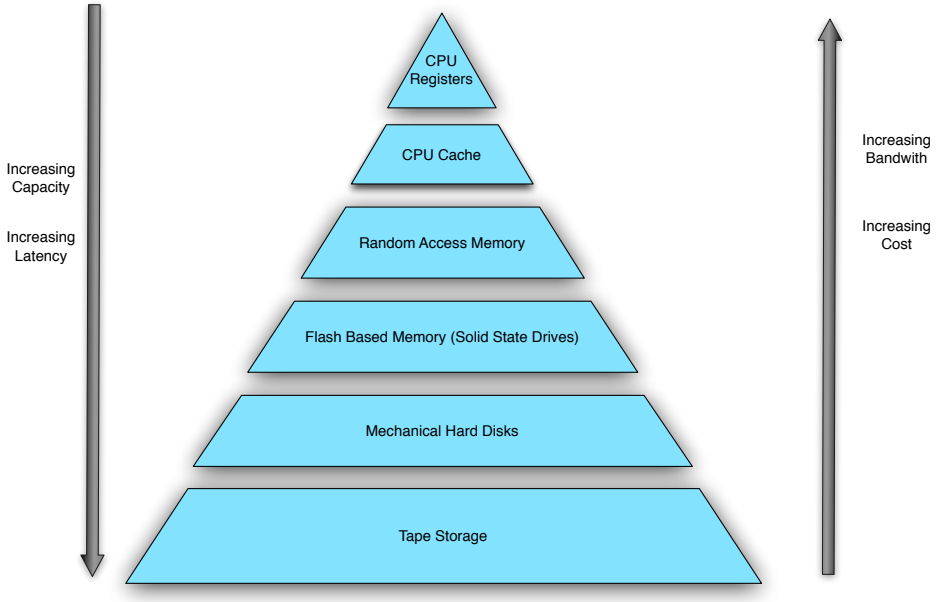


Figure 2.2: The Memory Hierarchy Pyramid

use of each is afforded different levels of importance depending upon the desired task, known as the memory hierarchy.

Prioritising the use of smaller but faster memory where possible is the core concept of the memory hierarchy, with multiple layers of varying size and performance. It is typically most common for higher bandwidth/lower latency memory to be small in size and more expensive (in both cost and/or physical space), while more sizeable storage capacity is reserved for slower forms of memory such as RAM. Hard disks, while slowest, offer a form of permenant storage and thus their use is typically reserved for long term data storage rather than intermediate compute, with the exception of checkpointing — a process that stores a snapshot for the resumption of processing in the event of an unforeseen error or interruption [36]. This balance between performance and size is represented as the **Memory Hierarchy**, captured in Figure 2.2.

The use of a memory hierarchy results in multiple layers of high-performance memory, with modern systems relying heavily upon the use of an on-chip cache of very small size (typically between 32 kB and 2 MB), moving data out of

slower memory into faster memory as necessary. A system is designed around promoting the use of this small amount of memory where possible to offset the performance cost of using larger, slower memory. This concept is not a binary either/or however, it is a sliding scale of balancing the cost of memory, its size and its performance. In particular, locating memory closer to the CPU is desirable due to the reduction in latency overheads that would occur out of transmitting data over smaller distances. However, such space upon a machine is at a premium and thus the prioritisation of faster memory is key to its effective use.

Prefetching is a technique to predict (or be informed prior) of upcoming memory accesses, and retrieve data in advance of it being required so that it already exists in cache for subsequent operations. These techniques can be either software or hardware based. Since this requires accesses from slower memory in the memory hierarchy, it is done in advance to take advantage of quiet periods on the memory bus (such as during intense compute), masking the cost of data movement. This requires that a number of conditions be met in order to be ultimately useful — the memory bus must not already be saturated, for no such prefetching can then occur, and for the prefetcher to be effective it must retrieve the *correct* data.

Obtaining the correct data requires the prefetcher to make an informed guess, based on the behaviour of the program, to predict what data may be required in advance of any requests. However, if it fails to achieve this, then a subsequent request for the actual data from cache will fail — known as a *cache-miss*. This incurs a significant performance penalty as the CPU must wait for the data to be retrieved from lower in the memory hierarchy and moved into the cache/CPU registers. In addition, if the cache was full, the prefetched data will have caused the ejection of other data from cache in order to make room for it. This can lead to complications of futher potential cache-misses if said ejected data is then requested. To improve the likelihood of an accurate prediction, there are two behaviours that a prefetcher will typically assume to be true. These behaviours

are known as **spatial locality** and **temporal locality**. Implementing memory access patterns that observe these behaviours will greatly improve the prefetcher success rate.

Under the principle of spatial locality, it is presumed that data stored close to previously accessed memory locations is more likely to be imminently requested. When a cache miss occurs, multiple data elements are transfered into cache at the same time as a whole, categorised as a **cache line**. The data elements retrieved are located sequentially in memory — the number retrieved dependant upon the size of a machine's cache-line (on modern architectures typically in the region of 32/64 bytes). By retrieving a cache-line, accessing sequential locations after the initial address results in a cache-hit due to the data already existing in cache. Memory patterns that traverse sequential blocks of memory are thus able to exploit the use of full cache-lines before they are evicted.

By the principle of temporal locality, a second assumption about data is made to improve the handling of data eviction from the cache. It is presumed that data accessed recently has a greater chance of being re-used than less recently accessed data, so data that has been less recently accessed in cache is given a greater priority in cache for eviction over more recently accessed data. By using memory access patterns that minimise the time between subsequent accesses to the same location, the chance of data still remaining in the cache is greater, as is the corresponding chance of a cache-hit.

Combined, both of these principles can guide the design of memory access patterns that improve the likelihood of reusing data within cache. By doing so this can have a significant reduction on the potential for the memory bus to become a bottleneck.

2.2.3 Network Interconnects

The key component that facilitates the use of distributed components to achieve massive scale parallelism is that of the network interconnect. Providing high-speed data communication mechanisms is crucial to ensuring minimal delay in

providing fresh data for processing to the various PEs within an HPC system.

In any distributed cluster there must exists some means by which data can be communicated between remote nodes in the system. This communication system is typically referred to as the network interconnect, and consists of three key features:

The Network Interface

Hardware located in nodes, dedicated to interacting with the network.

The Network

Hardware dedicated to the task of transporting communications, such as switches. Examples include high-speed ethernet [116] and multiple, differently performing variations of Infiniband [11, 29] such as DDR, QDR or FDR.

The Network Topology

The arrangement of distributed nodes within the network, dictating the routes available between two nodes through the network. Examples include Fat Tree [98], Torus [23] and Hypercube [22, 152]. The dominant characteristic this influences is the speed that a packet can traverse the network and can be down to many factors, with the positioning and availability of network switches and interconnects determining the available routes for selection. The routing and data management algorithms (such as data buffers) determine the speed with which packets can traverse these available routes, by processing data in a timely manner and identifying the routes with the shortest traversal time. Further, while the underlying hardware determines the theoretical peak throughput, existing network contention can limit the actual throughput while also causing said routing algorithms to select longer routes. Managing these factors to select suitable routes for the packets across the entire machine is crucial to ensuring good parallel performance.

The selection of these three characteristics influences both the network bandwidth and the network latency:

Network Bandwidth

The network bandwidth dictates the *throughput* of a network — i.e. the amount of data that can be transported via a network connection at the same time.

Network Latency

The network latency dictates the *travel time* of data on a network — i.e. the time taken for data to travel from a source A to a destination B. This combines with the bandwidth to dictate the overall time taken for data transport. With a latency α (seconds) and a bandwidth β (bytes/second) the time taken for transport is roughly equivalent to $\alpha + \beta \times datasize(bytes)$. Assuming a uniform travel time for a stream of network packets (e.g. they traverse the same network route, all packets are the same size etc.), the latency cost is only measureably incurred on the first packet communicated — the travel time of the second packet is mostly masked by the travel time of the first, the third by the second etc.

2.2.4 The Software Stack

Implementing and executing a parallel application is dependent not only upon the underlying hardware, but also the software that interfaces with it. The selection of a suitable software stack is crucial to ensuring a machine efficiently employs its hardware to achieve the best performance, with some vendors providing software attuned and optimised for their own architectures. Software that can prove to be influential to the performance in an HPC environment can include:

The Compiler

Different compilers may incorporate alternate optimisations or exploit specific hardware features. This can include the use of CPU features such as vectorisation or memory access pattern optimisations.

The Parallel Programming Interface

A software API can provide communication functionality to a developer, enabling the exchange of data between distinct PEs. Examples include MPI or UPC. Their selection can depend upon the inclusion of machine specific interface optimisations or support for select network interfaces.

The Scheduler

In shared HPC environments with multiple users, the contention for resources means that frequently a queue is required to manage access. A scheduler handles the prioritisation of jobs, balancing the resources (both hardware and time) requested against the time spent in the queue to ensure that a high machine utilisation is achieved without causing overly long queue times for any particular user. A good scheduler will ensure that the number of idle CPU hours is kept to a minimum.

The software stack on a HPC machine is normally application independent due to such devices often managing a wide range of user requests. However, typically a selection is available to the end-user, from which the most appropriate can be selected via configuration settings.

2.3 Performance Analysis and Modelling

To understand the performance of an application it is neccesary to identify what is expected behaviour and what is unexpected behaviour. If an application falls short of an optimal performance outcome then there is a potential opportunity for improvement, either algorithmically or in the configuration. In turn, identi-

fying the ideal outcome is dependent upon a combination of the capabilities of the machine, assuming ideal conditions, and a theoretical limit on the algorithm in question. Much work from the academic and industrial community has gone into the development of algorithms, descriptive laws and tools that allow the capture and analysis of an application's performance, both theoretically and in real-world scenarios.

2.3.1 Amdahl's Law

Amdahl conceptualised a definition of the maximum achieveable speedup of a parallel application, termed **Amdahl's Law** [6]. This law, depicted in Equation 2.1, was intended to show the limits of performance scaling for a fixed problem-size as the number of parallel processes in a system increased — an experimental setup known as **strong-scaling**.

$$Speedup = \frac{(s+p)}{(s+\frac{p}{N})} \tag{2.1}$$

The terms s and p represent the proportion of application runtime that consist of serial components and parallel components respectively. The speedup of an application is a factor of the runtime at N processes, $s + p/N$, against its performance when N is one, $s+p$. Serial components show no change regardless of the size of N, whereas the contribution of the parallel components is inversely proportional to the number of processes.

This introduces a ceiling to the obtainable performance from a parallel code. As N tends to infinity, p/N tends to zero, resulting in the maximum achieveable speedup described in Equation 2.2.

$$MaxSpeedup = \frac{(s+p)}{s} \tag{2.2}$$

When the runtime becomes dominated by the serial component, any improvements in performance are restricted since $s > p/N$, and no increase in N can reduce the cost of s. From this, two conclusions can be drawn about parallel

performance:

- As the serial component comes to dominate the overall walltime, investing in more parallel processes has diminishing returns;
- No matter the level of optimisation within the parallel component, the maximum achievable speedup is bound by s.

Thus while achieving a greater scale offers the potential for improved performance, the potential gains are constrained by the serial component of the application. Unless the elimination of such serial components is addressed, performance gains at scale will be restricted.

2.3.2 Gustafson's Law

Gustafson proposed that there was a flawed assumption in using Amdahl's law when considering parallelisation efficiency [70] — namely that Amdahl's law assumes a strong-scaling setup where the overall problem size is fixed, with the per-process problem size decreasing as the process count, N, increases. Contrary to this, Gustafson instead argued that there is a second scenario in which parallel hardware may be used. For an arbitrary problem size, rather than using the increased parallelisation to improve the runtime, a larger problem size could be executed in the same time as the original, smaller, problem size — an experimental setup known as **weak-scaling**. This experimental setup exploits the notion that larger HPC machines are capable of executing experiments that were previously infeasible due to time, memory and/or storage constraints.

In this approach, both the serial component, s, and the parallel component, p, per process would be fixed as the process count N increases due to the scaling of the global problem size (but fixed size per process). Since the parallel component per process is fixed, then the time on a single process is $s + p$. If a problem scaled for N processes were to be run on a single process, the runtime taken would be $s + p \times N$. Theoretically, in a weak scaling scenario, the scaled up problem size on N processes would run in the same time as a single process,

thus the *scaled speedup* can be derived by Equation 2.3 to provide an alternate means of analysing efficiency compared to Amdahl's law, termed **Gustafson's Law**.

$$Scaled\ Speedup = \frac{(s + p \times N)}{(s + p)} \tag{2.3}$$

The crucial difference arises due to the ratio of serial to parallel components per process. Under Amdahl's law, in a strong scaling setup the runtime comes to be dominated by the serial component as the process count increases, due to the ratio of serial to parallel skewing towards the serial cost. As such, the addition of more processes results in diminishing returns, and a declining parallel efficiency. However in a weak-scaling setup there is less of a skew towards the serial component due to the amount of parallel work scaling with the process count to remain fixed per process. This is a setup representative of many experimental workloads in modern HPC, due to the construction of larger machines enabling experiments previously unavailable due to time, memory or storage constraints, hence Gustafson's law provides a more suitable means for comparing the speedup of these tasks by reframing the parallel efficiencyiy in this context. The diminishing returns of Amdahl's law are avoided not by completing the same amount of work in less time, but by completing more work in the same time. In so doing, the performance is not bound by the minimum time required for the serial component, and the potential speedup is represented by Gustafson's law.

2.3.3 Benchmarking

The modern HPC supercomputing machine consists of a wide-array of components, and has a variety of architecture selections available. To anyone looking to procure or use an HPC machine, some form of comparison is neccesary in order to identify the optimal selection. One such approach is the use of the LINPACK/High Performance LINPACK (HPL) benchmark [50] used for the

purposes of the TOP500 [161] supercomputer rankings.

Benchmarks such as LINPACK are intended to obtain suitable metrics which provide quick and easy comparisons between machines, using real-world performance as opposed to theroretical hardware peak performance. It summarises these results in the form of the number of Floating-Point Operations per Second (FLOP/s). Due to the significant use of floating-point operations in many scientific codes, these are considered a reasonable metric of "useful work" performed. However, LINPACK is most representative of applications that are computationally bound, not those that may exhibit alternative patterns such as unpredictable memory access patterns or possess a low ratio of FLOP/s to memory operations. As such, it can be flawed to use LINPACK results as the sole means for comparing supercomputing performance [49].

While a single number as a comparison metric sounds appealing, it can inhibit the comparison of different architectures in a number of ways. Due to the TOP500, a not insignificant amount of importance is accredited to having a high LINPACK ranking, especially when significant costs are involved in procuring such machines. Care must be taken such that achieving a high performance on LINPACK does not become the sole focus of design efforts, as opposed to identifying and targeting bottlenecks that may impact a machine's intended workload. A wide range of scientific applications exist, and as such different algorithms can exhibit different behaviours and, by extension, make different demands of the underlying architecture and its components. The HPC Challenge Benchmark [105] is one such alternate approach, incorporating not only HPL but an assortment of kernels designed to assess both local and global compute for kernels with both high and low spatial and temporal locality. Its use allows for the capture of a selection of different kernel metrics to be reported in a manner akin to the TOP500 [104], demonstrating the relationships between kernel behaviour and the architectures upon which they are run. It is apparent that while LINPACK has proven to a be a useful benchmark for understanding the FLOP/s potential of a machine, it is not representative enough to act as the

sole means of comparison when considered in the wider context of a machine's intended purpose.

The selection of any HPC machine should, first and foremost, be conducted with the target workload in mind. To aid in this goal, a number of more specialised benchmarks have been constructed over the years that are intended to capture different metrics of interest pertaining to specific scientific algorithmic behaviours, such that a greater understanding of both the machine and how well a code will perform can be formed. These benchmarks fall into two forms of classification — micro-benchmarks and macro-benchmarks.

Micro/component benchmarks are targeted at a particular characteristic of a machine, be it compute, memory, I/O performance etc. Their intention is typically to capture the raw performance of a particular component under ideal conditions, devoid of other compromising bottlenecks using a small piece of code that focuses predominantly on the behaviour of interest. Examples include CacheBench [121] and STREAM [111] (memory); the Intel MPI Benchmark [81], and SKaMPI [167, 151] (network); and IOR [95, 156] (I/O). While such benchmarks cannot describe the behaviour of an algorithm, if a component is known to be the limiting bottleneck such benchmarks can help to reveal the extent of the machine's limitations.

Macro-benchmarks are intended to be representative of real-world applications, using multiple aspects of a system. They typically constitute cut-down versions of such applications, intended to capture the core behaviours of the application and how the various subsystems of a machine interact with one another. Examples of such benchmarks include the NAS Parallel Benchmarks [13, 14], and Sweep3D [75, 140].

The goals of these two classes of benchmarks are to provide different forms of insight into a machine's performance. Micro-benchmarks are ideally suited to capturing the behaviour of one particular aspect of a machine, be it the compute, network or memory. Macro-benchmarks however are better suited to capturing the interactions between these systems, identifying the unexpected

bottlenecks that may arise based on an application's demands of a machine. In an ideal scenario, an algorithm would use 100% of all necessary subsystems simultaneously, with each subsystem handling data as fast as it arrives. In reality however, systems are not in such perfect balance.

Advances in machine components, developed independently of one another, have progressed at different rates. The most highlighted example of this is the influence of memory and compute [110], the memory wall [178] being identified as a significant factor in the performance of some HPC applications [8, 113]. As HPC machines have progressed to ever larger scale, the onus has fallen to the software developer to ensure that algorithms are sufficiently parallel to fully exploit the hardware. Given the proliferation of legacy codes, it can be the case that codes are not optimised for a specific architecture or fail to take advantage of a machine's full potential. As a result a code may become significantly bottlenecked by a subsystem and leave portions of the available resources idle.

With the unlikely scenario that a machine is in "perfect balance", it is to be expected that there will always be some form of bottleneck within a system. Identifying the current bottleneck and eliminating it is a key part of the process of optimisation — a task that requires both knowledge of the machine and the application. To support this, a wide range of profiling tools and techniques have been developed by the HPC community. Their effective use now forms a key part of any HPC code development process.

2.3.4 Profiling

Understanding the internal behaviour of an application is crucial to identifying and improving bottlenecks within an application. A significant amount of research has gone into tools that can obtain performance metrics about internal application behaviour, breaking down what would otherwise be a black-box scenario. These techniques are collectively known under the moniker of *profiling*. Of the various profiling approaches available, each possess different advantages and disadvantages. These techniques include:

- The use of hardware counters to track CPU metrics of interest;
- Instrumentation timers that capture the runtime of code blocks;
- Statistical sampling that approximates the time spent in different stack locations;
- Application traces that capture runtime via instrumentation, while preserving temporal data — distinct timings for the same code block called at different times are preserved.

Hardware Counters

Hardware performance counters are specific to the hardware, using special registers to track the execution of events of interest defined by the runtime environment. Such counters provide a flexible means of measuring metrics that typically would be difficult to extract from the application behaviour due to their low-level nature, such as the number of floating-point operations or cache misses, and would otherwise require other approaches to assess such as simulation.

Instrumentation

Instrumentation is the most direct of profiling approaches. Timers are introduced to capture the walltime (or cycle count) between two arbitrary positions within an application. Profilers that adopt this approach typically instrument every function call; for example, GProf [66] will provide a stack trace of the critical path, accompanied by a runtime breakdown of each function. This provides insight into the functions that dominate overall walltime.

Such approaches however are not necessarily refined enough for parallel applications. At the function level no distinction is obtained between critically different behaviours such as compute and communication operations. Some profilers, such as Tau [157] or Vampir [92, 128], provide additional

functionality to capture event data via an instrumentation API through dynamic or manual means. Further, MPI libraries provide specialised function names as part of MPI Profiling Interface (PMPI). This allows for the capture of MPI function calls via dynamic library loading, where timers can be wrapped around MPI calls before calling the true function.

The use of instrumentation can provide a detailed breakdown of performance. However manual instrumentation can rely upon possessing an already in-depth knowledge of the areas of interest within a code. In addition, care must be taken to avoid the introduction of a significant overhead, which could potentially skew any results.

Statistical Sampling

An alternate approach to that of manual instrumentation is *sampling*. Sampling differs from instrumentation in that, rather than wrapping dedicated timers around a block of code, predefined time intervals are specified to query the current location within the stack and derive the duration spent within a specific location statistically based on its frequency. Sampling is overall less disruptive, since it prevents the overhead issues of high frequency function calls (unless defining a very small interval period), but at the cost of less accurate timing.

Tracing

The approach that provides the most extensive output is the production of application runtime traces. Instrumentation as described above provides runtime data, i.e., how *long* a particular portion of code took, but is usually aggregated across the course of the run. No temporal data is stored, preventing any distinction between separate executions of the same call path. Temporal data can be particularly useful, especially in understanding the synchronisation behaviours of applications. By using a common reference point across all processes, a trace can reveal the duration of time taken

to reach a shared common point within an application, such as a blocking communication procedure where one or more processes are dependent on remote processes to continue. However the use of traces produces more data than that provided by aggregration metrics, resulting in significant storage requirements. In addition, these storage requirements increase with process count and, as is typical of any technique that produces sizeable datasets, can be difficult to parse for behaviours of interest. Both Tau and Vampir include functionality for such an approach, alongside alternate tools such as Scalasca [64].

As well as any tools described above, an internal instrumentation library was developed for use in Atomic Weapons Establishment (AWE) applications, called the Performance and Modelling Timing Interface (PMTM). Unless otherwise mentioned, all investigations conducted within this work make use of this interface to take timing measurements. Further description of its functionality can be found in Section 3.1.

2.3.5 PRAM Model

The Parallel Random Access Machine (PRAM) model is an abstract description of a parallel architecture of p processes on a shared memory machine proposed by Fortune and Wyllie [58]. It assumes a number of characteristics to reduce complexity, including:

- An unspecified (potentially unlimited) number of parallel processes;
- An unspecified (potentially unlimited) amount of shared memory;
- A uniform unit time for memory access from any process, ignoring issues such as latency, memory locality, cache behaviour and resource contention (such as the memory bus).

The different classifications of parallelism are:

- Exclusive Read, Exclusive Write (EREW) — No simultaneous access by two processes to the same memory location is permitted; This can inhibit performance if there is any overlap between processes in the accessed memory locations;
- Concurrent Read, Exclusive Write (CREW) — Memory can be read in parallel, but two processes cannot write to the same memory in the same step. This can significantly improve performance in any algorithm that shares a data input set but maintains separate blocks of memory per process for data storage;
- Exclusive Read, Concurrent Write (ERCW) — Typically ignored due to concurrent write but no concurrent read being an unusual behaviour;
- Concurrent Read, Concurrent Write (CRCW) — Memory can either be read or written to by multiple processes simultaneously. This has the greatest performance potential as there is no restriction on two or more processes accessing the same location.

CRCW poses the fewest restrictions on memory access, and in turn the greatest potential for good performance. However unlike concurrent reads (which do not interfere with the correctness of the data), concurrent writes possess a non-deterministic nature due to the result being tied to the order of operations. This leads to a further refinement of the behaviour of concurrent writes:

- Common — All processes must write the same value, else an undefined state is achieved;
- Arbitrary — Any single random processor gets a write attempt, all other remaining pending writes are unused;
- Priority — A priority algorithm determines which process out of all pending writes is chosen.

The abstract nature of PRAM makes it especially amenable to the construction of parallel algorithms in the absence of a formal implementation. The use

of an idealised nature of the underlying hardware makes it especially useful for comparison from an algorithmic complexity perspective. This encourages the design of parallel algorithms that are theoretically faster. Unfortunately, this same characteristic inhibits the usefulness of PRAM for performance modelling. The level of abstraction prevents any comparison of the same algorithm on two different hardware environments, as issues such as contention and machine performance metrics are abstracted away in the assumption of uniform time, factors which have a significant impact upon the real-world performance.

2.3.6 The Bulk Synchronous Parallel Model

The Bulk Synchronous Parallel (BSP) model [168] is another model that sets out to capture the behaviour of distributed parallel algorithms similar to the PRAM model. A BSP algorithm consists of a series of *supersteps*. Each superstep consists of a three-stage process:

- A concurrent compute stage where p processes each perform a block of local compute — the degree of compute does not have to be equal between processes;
- A bulk communication stage — communication of h messages occurs between the processes, where g is the time taken to communicate a single message;
- A synchronisation stage — a global synchronisation ensures the completion of the bulk communication stage.

The advantage of an approach such as BSP over PRAM is that it does not assume a uniform cost for any action — it aims to incorporate the cost of communication and synchronisation for data communication, thus providing a more representative overview of the cost of parallel overheads. It is dependent upon the operations operating in lockstep, with synchronisation points enforcing the end of communication, but provides a more realistic assessment of performance

when considering modern parallel environments that consist of a wide variety of communication costs depending upon the hardware.

2.3.7 LogP/LogGP

The LogP model [40, 41] was developed as an analytical, mathematical means of representing the cost of a network communication. It parameterises the communication such that a model of the communication cost can be constructed from the machine metrics. These metrics include:

- L — The maximum latency of a communication between two processes.
- o — The compute overhead of sending a small message.
- g — The minimum gap in between communicating small messages. The reciprocal of this is the bandwidth of the network for small messages.
- P — The number of processes.

This model was further improved upon by Alexandrov [4], who expanded upon the initial LogP model to include an additional parameter G — the minimum gap between communicating large messages. The reciprocal of this is the bandwidth for large messages. This important contribution made the LogP model better able to capture the network performance, due to potential dissimilarities in behaviour between small and large messages on different interconnects. A number of works have since sought to expand upon the work of LogP/LogGP, introducing additional characteristics such as network contention [60, 119].

2.3.8 Statistical and Analytical Modelling

One of the simplest approaches to modelling is the exploration of historical data in conjunction with statistical techniques to identify trends in an application's performance. Methods such as simple linear regression enable the quick identification of basic trends, yet without capturing and separating more com-

plex attributes such as algorithmic behaviour or machine attributes, accurate conclusions about an application's behaviour cannot be drawn.

The use of analytical modelling is intended to provide a more accurate means of applying statistical approachs in conjunction with parameterised, refined mathematical models that can adequately capture detailed behaviours of HPC applications, typically focusing upon the critical path (normally the execution path taken by an application for a specific problem of interest). This is historically based on work from the field of queueing theory, with a stochastic modelling approach to performance [27, 102, 117, 148], and a deterministic approach advocated by the thesis of Adve [2], who argued that for the purposes of performance prediction the potential variance introduced into a system by communication and contention for many general cases has minimal impact on the execution time.

By employing this deterministic top-down approach a general model of a critical path can be validated by the use of existing historical data (via an understanding of the critical path and profiling/instrumentation), ensuring all contributors to a component are captured, and then further refined by the introduction of sub-models that are capable of using relevant parameters for the purposes of prediction rather than requiring pre-existing historical data, which does not exist for unexplored scenarios of interest. Examples of such include PRAM, BSP and LogGP (as already introduced in this chapter), which can make use of measurements such as network micro-benchmarks or application instrumentation to obtain relevant metrics.

The advantages of an analytical model is that it enables the relatively rapid and flexible prediction of different machine configurations or hardware via the substitution of new values without a lengthy computation phase due to the mathematical nature of the approach. However, there are also some factors that can inhibit the usefulness of this approach. The use of statistical techniques prevents a significant degree of depth — e.g. capturing the time of single instructions can be prohibitive given the complexities of measuring such values

(such as accounting for parallel pipelines and other processor optimisation techniques). Instead, an analytical model will focus upon the critical path *blocks*, collections of instructions that can be grouped into a single sub-model that can be parameterised to provide sufficient predictive capacity without the need for further refinement. For example, the compute component of any model may be broken down into compute *kernels*, blocks of code that apply a repetitive set of instructions with an influencing parameter that dictates the number of times it is executed, such as a loop. In this scenario, a value representing the time taken to conduct a minimal block of compute such as a single loop iteration, the Wg value (a term appropriated from the work of Mudalige [124]), can be used in conjunction with a parameter that determines the number of loops to provide an overall prediction for the block of compute without the need to assess the performance at an instruction level. By repeating this process for all kernels, deriving the Wg values by statistical means such as regression analysis from prior historical data, a set of sub-models can be built to form an overall compute model. The use of such analytical techniques does however come with a set of restrictions.

First, given the generalised nature of sub-models, more complex mannerisms such as conditional execution paths can pollute the historical data set, leading to inaccurate sub-models if not accounted for in a parameterised way. This can be resolved through the use of focused experimental setups — where a problem of interest focuses upon a particular execution path, or through the use of sufficient parameterisation. However, even with this possible restriction, there are numerous examples of the use of analytical models that not only accurately predict the runtime [75, 76, 90, 107, 123, 124, 163], but are also used to aid in the machine procurement/investigation [74, 88] and machine configuration [88, 89, 141, 158] when executing HPC applications, implying there are sufficient use-cases to make this a viable approach.

Second, the construction of such models neccessitates a sufficient level of application/domain knowledge, especially given the complexity of parallel HPC

applications, and as a consequence their construction can often be a lengthy procedure due to the time investment required in their creation. Attempts have been made to redress this with either the use of reusable models that can be applied across applications such as in the work of Mudalige [124], or the use of automation to reduce the application knowledge required to develop such models [166].

2.3.9 Simulation

Simulation of a system typically takes one of two forms — continuous or discrete. Continuous simulations model a system that is a constant state of flux, via means such as mathematical equations. Discrete-event simulation represents a system's state as that of a stream of individual events, the processing of which takes a system from one state into the next. In a performance model, an application's instruction stream can be thought of as a set of events, with a performance model predicting the time taken to conclude each event by the simulation of hardware from the intended target system. By processing the full set of events, this results in an overall model for the final runtime. Such systems typically make a tradeoff between simulation time and accuracy through the definition of what constitutes an "event" — the more refined the event, the more events that must be processed, resulting in a more accurate but longer simulation time. As such, there exist a few different approaches to simulation-based performance modelling.

Instruction-driven simulation attempts to simulate an application's execution on an instruction-by-instruction basis such as with PACE [35, 132]. This has a few benefits, namely that it provides a high degree of accuracy without the need for someone to be familiar with the underlying algorithms and codebase — the extraction of an instruction list can be automated. However this also comes with significant compute costs leading to lengthy simulation times that, while still useful for exploring unavailable hardware, might be little faster (or possibly even slower) than executing an application on the native hardware.

Trace-based simulators such as DIMEMAS [94, 142] use traces to store the outcomes of executed instructions, typically from an application execution on smaller configurations, and use these to extrapolate simulations on alternate scenarios such as large-scale configurations. However, the use of traces can prove to be prohibitive given the parallel environments of HPC, with the traces requiring large amounts of storage space, and the generation of new simulations being dependent upon these pre-existing traces.

The WARPP [72, 71] toolkit takes a more coarse-grained approach to that of instruction-driven simulation, defining an event to be an appropriate collection of instructions such as a loop block of compute. Used in conjunction with application instrumentation, the toolkit is able to associate a time-cost with these event blocks, enabling the prediction of runtimes through a simulation of the event-list. This is in turn supported by additional models, such as network profiles for MPI communication, or even possibly via the substitution of models from alternate techniques such as an analytical approach. Some systems such as POEMS [3] enable a hybrid approach, incorporating both analytical and simulation-based modelling using systems such as LogGP [4] or MPI-SIM [144].

2.4 Summary

In this chapter the core topics and literature of HPC and performance modelling/analysis were introduced, specifically:

- The multiple forms of parallelism used in modern HPC;
- The critical components that form part of a modern HPC platform;
- Existing work from the field, including existing laws and performance models used to capture the behaviour of parallel algorithms;
- Existing approaches to performance analysis and model construction that influence the approaches taken within this work.

In the next chapter the tools and machines used throughout the remainder of this work to define a well understood experimental environment for this work are established.

CHAPTER 3

Software and Hardware Overview

To investigate the scaling properties of applications and algorithms, a set of experimental tools and architectures of interest must first be established; this enables investigations to be conducted in a well-defined, reproducable manner. This chapter introduces the libraries, benchmarks and machines that are used throughout the course of this thesis, as well as presenting micro-benchmark results for architectures of interest. The following topics are addressed:

- Throughout the course of this work use is made of multiple existing libraries, developed either for the purpose of this work or by third-parties for general use in academia and industry. These libraries either facilitate the implementation of a parallel scientific program, or are used to measure performance metrics of interest, and are detailed here in Section 3.1 in the interests of potential future experimental replication;
- Section 3.2 introduces the benchmarks used within this work for capturing the performance metrics, both micro and macro in nature, that are used to assess both the machine architectures and the potential real-world performance of applications. Such metrics are useful in identifying the potential bottlenecks that inhibit performance, and inform our analysis of under-performing applications;
- The architectures/machines used within this work are detailed in Section 3.3 for the purposes of both experimental replication as well as for comparative purposes. To this end, a selection of the micro-benchmarks introduced in Section 3.2 have their results presented here alongside the machine specifications. Macro-benchmark results are also obtained, but

the outcomes of these experiments are not presented within this chapter; rather, these are analysed in greater depth in Chapters 4 and 7.

3.1 Libraries

The work in this thesis makes use of a small number of notable libraries provided by the Atomic Weapons Establishment (AWE) or other third-parties, either as part of benchmarking efforts or application implementation. A brief description of these libraries is provided here for reference.

The Performance and Modelling Timing Interface

The Performance and Modelling Timing Interface (PMTM) is a small instrumentation library developed by AWE and the University of Warwick that provides facilities for defining, using and aggregating timing results between two arbitrary fixed points in an application. In this work it measures the walltime of blocks of code, enabling us to identify hot-spots, validate critical-paths and provides historical data useful for the construction of performance models. This time is retrieved via the difference between two timer calls, made using the C function *gettimeofday*. Different blocks of code are assigned different identifiers to distinguish between one another. These times are aggregated across multiple calls to the same timer block to reduce storage requirements, reporting a mean time taken per Message Passing Interface (MPI) process.

The Portable, Extensible Toolkit for Scientific Computing

The Portable, Extensible Toolkit for Scientific Computing (PETSc) [16, 17] is a library developed for the purposes of efficient and scalable parallel solving of systems of linear equations, solving for x in the matrix equation $Ax = b$. Containing multiple implementations of a variety of solvers and preconditioners, it also provides interfaces for alternate third-party libraries while at

the same time acting as a consistent general framework which application developers can use to explore the use of different linear solver approaches. It provides support for both serial and parallel applications, using MPI for the communication of distributed data.

PAPI

The Performance Application Programming Interface (PAPI) [31] provides a unified interface for interacting with performance counter hardware that is often provided by a wide variety of modern microprocessors. Due to potential differences between manufacturers in hardware counter implementation, the use of such an interface allows for a "write-once, reuse-forever" approach to capturing useful performance information such as cycle counts, cache hits/misses and number of floating-point operations. These details provide useful insight into understanding where or why a code may be underperforming on an architecture, an important part of the performance engineering process.

3.2 Benchmarks

During the course of this work, a select number of benchmarks were used to acquire information about various machine characteristics.

3.2.1 Network Interconnect Micro-Benchmarks

The use of network interconnect benchmarks enables us to explore the underlying latency and bandwidth performance of various hardware without the complicating factor of a scientific application's additional interactions, such as contention, synchronisation or load-balancing that may misrepresent the potential peak performance. The two interconnect benchmarks used in this thesis are:

Intel MPI Benchmark

The Intel MPI Benchmark (IMB) [82] captures the performance of MPI network operations, including point-to-point, collective and I/O operations [83].

SKaMPI

The SKaMPI benchmark [167, 12] is an alternate network benchmark that is similarly capable of capturing both MPI point-to-point and collective operations, useful for validation of the IMB benchmark output. It is also extensible, enabling the provision of custom tests for exploring alternate network scenarios other than the tests provided by default.

3.2.2 Memory Micro-Benchmarks

Given the significant data processing elements of many scientific applications, ensuring sufficient throughput of data for compute is key to maintaining a good degree of performance. Memory benchmarks provide an insight into the underlying capabilities of a target machine or architecture, highlighting potential bottlenecks that could prove to be an inhibitor of high performance.

STREAM

STREAM [111] is a memory benchmark that by default uses a large block of memory to measure the bandwidth performance of a machine's Random Access Memory (RAM). It has both C and Fortran implementations, as well as being capable of executing in either a single-thread, OpenMP or MPI setup. Using a large fixed-size block of memory, the benchmark captures the bandwidth of a number of different operations (with differing byte and Floating-Point Operation (FLOP) counts), summarised in Table 3.1.

The STREAM benchmark is particularly useful for capturing the behaviour of multi-core contention. When the memory bandwidth is substantially restricted, Central Processing Units (CPUs) can become memory starved due to the inability of the memory subsystem to sustain sufficient

Operation	Kernel	Bytes	FLOPs
Copy	$a(i) = b(i)$	16	0
Scale	$a(i) = q * b(i)$	16	1
Sum	$a(i) = b(i) + c(i)$	24	1
Triad	$a(i) = b(i) + q * c(i)$	24	2

Table 3.1: STREAM Benchmark Operations [111]

throughput. By scaling up the number of cores used per node, the degree to which performance can suffer as a result of an increased load on the memory bus can be captured. Any such memory-starving should manifest itself as poor scaling when the number of cores is increased. It is crucial to identify such behaviour as any applications that process a substantial amount of data (common in scientific simulations) can exhibit memory-bound performance if the bandwidth is insufficient.

CacheBench

CacheBench [121] is a tool designed to capture the bandwidth performance of the multiple levels of cache that a machine may possess. As part of LLCbench [120], it provides useful insights into the underlying performance of cache-level memory accesses, a key component of many scientific applications that can process a substantial amount of data. In particular, it can reveal the potential cost of a cache-miss for different levels of cache, crucial given the variety of possible memory access patterns that can arise from different data processing requirements. Tests include read performance, write performance and Read/Write/Modify (RWM) performance.

3.2.3 Macro-Benchmarks

Unlike the previous micro-benchmarks designed to capture a single aspect of a system, these macro-benchmarks are intended to be more representative of real-world applications, stressing multiple characteristics of a machine at once. This work focuses on two macro-benchmarks of interest, explored within the

context of performance analysis and modelling.

Hydra

Hydra is a benchmark 3D Eulerian structured mesh hydrocode implemented in Fortran, with which the explosive compression of materials, shock waves, and the behaviour of materials at the interface between components can be investigated. The Hydra benchmark code simulates a cube of mixed materials under stress by discretising the data onto a 3D grid of cells given by $N_x \times N_y \times N_z$ and using message passing for parallelisation. Thus, in a typical Single Program Multiple Data (SPMD) fashion, the 3D cube of data is decomposed onto a number of processing elements (PEs) during execution. During the course of this work Hydra is used as part of a case-study, demonstrating both the performance and optimisation prediction capabilities of our performance analysis and modelling efforts. Further information on the Hydra benchmark can be found in Chapter 4.

Orthrus

While simple in description, solving for x in a linear system $Ax = B$ proves to be an expensive and common problem across a range of high-performance scientific domains [77, 86, 103]. Orthrus is a benchmark 2D/3D radiation solver developed at AWE, intended to explore the use of different linear solver solutions such as Conjugate Gradient (CG) or Algebraic Multi-Grid (AMG). It captures the behaviour of a structured, 7-point stencil, sparse linear system that passes linear solver capabilities to external third-party libraries such as PETSc.

This work modifies the Orthrus benchmark to use one of PETSc's newer Application Programming Interfaces (APIs), the Distributed Array structured interface introduced in version 3.2. This allows the exploration of performance when ghost cells, underlying matrix memory allocation and grid decomposition are all handled exclusively by PETSc, and the correspond-

ing performance of a linear solver approach such as CG within a parallel environment. Further details can be found in Chapter 7.

3.3 Machines

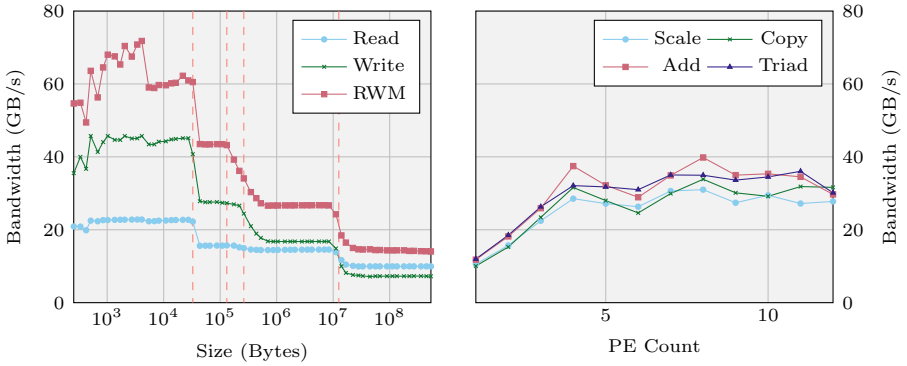
A number of different architectures are used during our investigations within this work. This section details the hardware and software components that make up these architectures, as well as provide the results of select micro-benchmarks for comparison and analytical purposes. Specifically, the four machines of interest within this work are Minerva (Section 3.3.1), Hector (Section 3.3.2), DawnDev (Section 3.3.3) and Hera (Section 3.3.4).

3.3.1 Minerva — Warwick Commodity Cluster

Processor	Intel Xeon X5650 (2.67 GHz)
Sockets Per Node	2
Cores Per Node	12
Nodes	396
Total Cores	4752
Memory Per Node	24 GB
Interconnect	QLogic Truescale 4X QDR InfiniBand
OS	SUSE Linux Enterprise Server 11
Compiler Toolkit	Intel v12.0
MPI Toolkit	OpenMPI v1.4.3/OpenMPI v1.4.4

Table 3.2: Machine Specification — Minerva

Minerva is a distributed computing platform located at the University of Warwick's Center for Scientific Computing (CSC). Provided by IBM and constructed from commodity components, it is a capacity cluster providing computing resources to scientific departments internal to the university and as part of Midplus, a collaborative computing effort between the University of Warwick, Queen Mary University, University London and the University of Nottingham. A full set of the machine specifications is provided in Figure 3.2, but most notable is its use of dual socket, hex core, 2.67 GHz Intel Xeon X5650s, 24 GB of memory per node and a QLogic Truescale 4X QDR InfiniBand interconnect.



(a) CacheBench (Single Core)

(b) STREAM (Multi-Core)

Figure 3.1: Memory Benchmarks — Minerva (Intel v12.0)

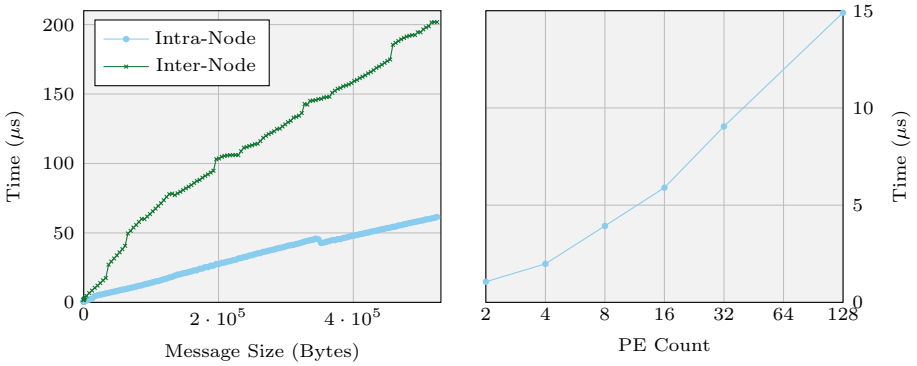
Memory Benchmark Results

Figure 3.1(a) captures the overall bandwidth throughput for various problem sizes of the basic CacheBench experiments — a double read test, a double write test and a double read/write/modify test. It is apparent that there exist four appreciable levels, consistent with the existence of three levels of cache and RAM. The most pronounced drops occur on all three tests at 32 KB and 12 MB, likely corresponding to the L1 and L3 cache sizes respectively, given a maximum cache size of 12 MB reported by Intel [80]. Another further drop is pronounced on both the write and RWM tests (and to a lesser degree on the read tests), and occurs between 128–256KB, the behaviour of which is likely corresponding to a transition from L2 to L3. This highlights the dramatic performance difference between the different cache levels and main memory, with a factor of 5.10× between the RWM benchmark's best and worst bandwidths.

However, while extremely useful at capturing cache performance, CacheBench does not capture the contention of main memory in a parallel shared memory environment. Figure 3.1(b) demonstrates this issue, presenting the results of the STREAM benchmark using multiple OpenMP threads to capture the behaviour of a parallel shared-memory environment. When the thread count reaches 4 or higher, it can be seen that the available bandwidth is capped between 3-3.5

GB/s depending upon the operation, suggesting that a significant degree of contention is occuring at higher thread counts due to saturation of the memory bus. The STREAM performance at one thread is also on par with the worst performing CacheBench results when hitting main memory after experiencing cache misses at large byte counts.

Interconnect Benchmark Results



(a) Point-to-Point PingPong

(b) AllReduce (8 Bytes)

Figure 3.2: Intel MPI Benchmark (OpenMPI v1.4.3) — Minerva

Using Intel's MPI Benchmark, it is possible to investigate the point-to-point and collective networking performance for both intra-node and inter-node behaviour. From the ping-pong results in Figure 3.2(a), it is apparent there exists a noticable disparity between the performance of inter-node and intra-node performance, as might be expected when contrasting a network connection with the performance of a node's memory. As the message size increases, the scaling performance is better for intra-node than inter-node communications, implying that any parallel application should place an emphasis on intra-node communications where possible.

At the smaller process counts the AllReduce performance for a single value in Figure 3.2(b) demonstrates a scaling behaviour exists in relation to the process count, as might be expected. However, unfortunately an opportunity for

an investigation at larger process counts was not possible due to the heavy use of the machine and machine restrictions. Nevertheless, these results provide a useful insight into the behaviour of machine load on the collective performance seen later within this thesis, where disparities between benchmarked times and instrumented calls within scientific applications can be identified. Since the IMB benchmark is conducted in isolation from any extraneous compute work, it would exhibit minimal synchronisation costs in the absence of any imbalance in the work-load across processing elements. As such it provides a means to distinguish between communication costs and synchronisation costs in later work.

3.3.2 HECToR

Processor	AMD Opteron Interlagos (2.3 GHz)
Sockets Per Node	2
Cores Per Node	32
Nodes	2816
Total Cores	90112
Memory Per Node	32 GB
Interconnect	Cray Gemini 3D Torus
Compiler Toolkit	PGI v12.10
MPI Toolkit	Cray MPT v5.6.1 (MPICH2)

Table 3.3: Machine Specification — HECToR

HECToR was a Cray XE6 supercomputer, funded by the UK research councils and operated by Edinburgh Parallel Computing Centre (EPCC), STFC Daresbury and NAG Ltd [135]. Running from 2007 to early 2014, it was a shared resource available for numerous scientific projects with access provided via an application process as part of the Partnership for Advanced Computing in Europe (PRACE) [57], before being superceded by a new machine, ARCHER. As a large scale machine with up to 90,112 cores available, using the more unusual AMD interlagos architecture, HECToR provided the means for exploring scalability at high process counts within this work, with some investigations reaching up to 16,384 cores. Other features of note include the use of a Cray Gemini 3D Torus network interconnect, as well as dual socket boards with 4 NUMA regions per node. An expanded specification can be found in Table 3.3

Memory Benchmark Results

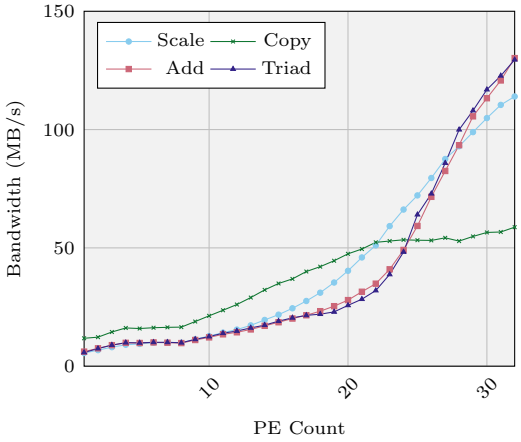


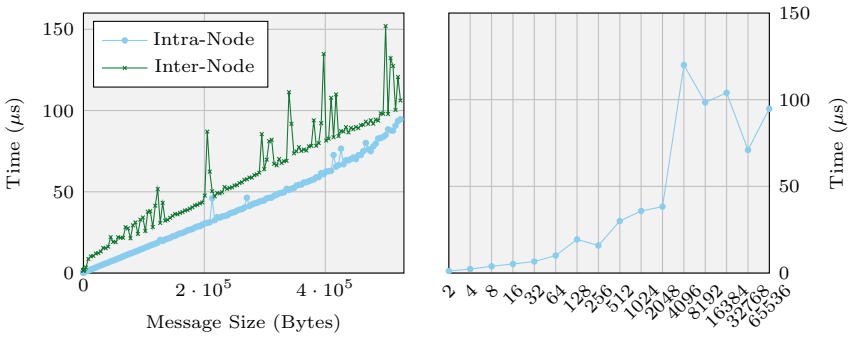
Figure 3.3: HECToR STREAM Benchmark (PGI 12.10)

As with Minerva, the nodes on HECToR possess dual socket setups, enabling two distinct CPUs that are able to access a shared unified memory. However, unlike Minerva, the memory architecture of HECToR consists of 4 distinct **N**on-**U**niform **M**emory **A**ccess (NUMA) regions, each tied to 8 individual cores. Within a NUMA region, any of these 8 cores have uniform access to this block of memory. However, accessing NUMA regions owned by any of the other 24 cores results in a bandwidth and latency penalty, with apparent implications for the performance of parallel processing.

Figure 3.3 examines the STREAM performance, scaling up to 32 threads within a single node. It can be seen that, as with Minerva, a contention point is reached at a low thread count of between 4-8 threads where the bandwidth performance does not improve. However, at 8 threads and above there is an increase in bandwidth that can likely be attributed to the use of additional NUMA regions (although note that the copy operation does appear to once again hit a maximum threshold and does not scale much past 22 processes). This implies that reasonable scaling performance can be achieved if the NUMA regions are used appropriately, though it still falls short of perfect theoretical scaling and such a conclusion cannot be assumed to hold in the event of cross-

NUMA region accesses, due to the performance hit that may be incurred.

Interconnect Benchmark Results



(a) Point-to-Point Ping-Pong

(b) AllReduce (8 Bytes)

Figure 3.4: HECToR IMB Benchmark Measurements (MPICH2)

On HECToR it is possible to investigate a much greater core-count than Minerva. The Ping-Pong experiments capture a reasonably linear relationship between the message-size and the time taken for communication. It exhibits less stepping than for Minerva, but once again there exists a measureable difference in performance between inter-node and intra-node performance. Some minor spikes in performance exist, but are uncommon enough that they may be attributed to network contention given the significant number of shared workloads that run simultaniously on machines such as HECToR.

The AllReduce experiments see an unusual fluctation in the maximum time taken at some higher process counts. The causes of these fluctuations between 2,048 and 32,768 cores is unknown, but could potentially be attributed to contention on the machines network at the time of investigation. Even so, there is an apparent pattern of gradually increasing performance costs when using the AllReduce function at higher core counts, as would be expected. When restricted to the process range of Minerva this cost is less apparent, exhibiting similarities between the two architectures in that there is relatively little

increase in time between 2 and 256 processes. This would suggest that the collective function will become a significant contribution towards the overall runtime of the application when using a significant number of cores, as might be expected, but have minimal impact at smaller process counts. With future architectures (especially those pushing for Exascale) preparing to use an ever greater number of processing elements, this makes the usage of AllReduce in any algorithm a factor worthy of further investigation.

3.3.3 DawnDev

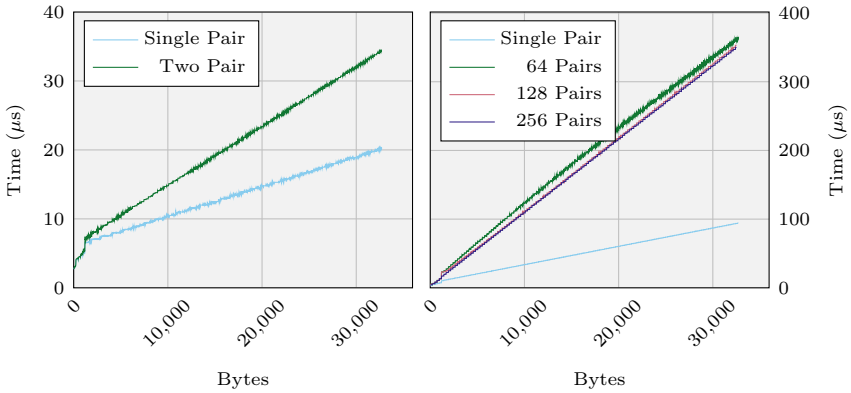
Processor	PowerPC 450(d) (850 MHz)
Cores Per Node	4
Nodes	1024
Total Cores	4096
MemoryPerNode	4 GB
Interconnect	BlueGene Torus and Tree
OS	IBM CNK
Theoretical Peak	13.0 TFLOPs
Compiler Toolkit	IBM XL 11.0 Fortran, 9.0 C
MPI Toolkit	IBM BlueGene MPI

Table 3.4: Machine Specification — DawnDev

DawnDev is a now decommisioned IBM BlueGene/P previously in use at the Lawrence Livermore National Laboratory (LLNL), acting as a development system for Dawn that was itself an initial delivery system for Sequoia, a BlueGene/Q deployed in 2012 with a LINPACK performance of 16 PFLOP/s. It exhibits the traditional properties of a BlueGene system, consisting of slower (850MHz) but more numerous processors than many typical commodity HPC systems, a small amount of memory per node (1 GB per core) and a proprietary BlueGene Torus interconnect. The architecture targets scalability with a relatively low power footprint over individual node performance. As such, the focus for applications on such a system is on parallelisation, rather than raw compute power, to achieve effective machine utilisation. This places a heavier emphasis upon the cost of communication if the use of parallelisation is intended to offset the use of poorer compute nodes. An extended list of the machine's

specification is provided in Table 3.4.

Interconnect Benchmarks



(a) Intra-Node Ping-Pong Timings, Intel MPI Benchmark

(b) Inter-Node Ping-Pong Timings, Intel MPI Benchmark

Figure 3.5: Network Benchmark — DawnDev

Figure 3.5 presents the outcome of the IMB benchmarks for select PingPong configurations. A selection of multi-communications benchmarks are also included, where messages are exchanged concurrently between fixed nodes. This is done in a fashion where all communications from a node, A, are received by another node, B, with two processes being paired together and multiple pairs of processes communicate in tandem with one another. The pairing configurations are as follows:

- 64 Pairs, seperated by a process id gap of at least 16 (i.e. 0 ⇆ 16, 1 ⇆ 17, 2 ⇆ 18, 3 ⇆ 19, 4 ⇆ 20 etc);
- 128 Pairs, seperated by a process id gap of at least 8 (i.e. 0 ⇆ 8, 1 ⇆ 9, 2 ⇆ 10, 3 ⇆ 11, 4 ⇆ 12 etc);
- 256 Pairs, seperated by a process id gap of at least 4 (i.e. 0 ⇆ 4, 1 ⇆ 5, 2 ⇆ 6, 3 ⇆ 7, 8 ⇆ 12 etc).

In doing so, the number of pairs within the overall system is varied, while

stressing the same communication link between two nodes. As might be expected, since this stresses the same communication link, this results in higher communication times, but the overall number of pairs in the system leads to little difference in this increase, suggesting it is predominantly the number of communications between a pair of nodes that is the dominant influencing factor.

3.3.4 Hera

Processor	AMD Opteron (2.3 GHz)
Sockets	4
Cores Per Socket	4
Cores Per Node	16
Nodes	847
Total Cores	13552
Memory Per Node	32 GB
Interconnect	4X DDR InfiniBand Switch
OS	CHAOS 4.3
Theoretical Peak	127.2 TFLOPs
Compiler Toolkit	PGI 8.0
MPI Toolkit	OpenMPI 1.3.2

Table 3.5: Machine Specification — Hera

Hera is a now decommisioned AMD/InfiniBand system that was based at LLNL, using an InfiniBand DDR high-speed interconnect. It exemplifies a more typical large capacity resource, with densely packed nodes of four quad-core CPUs across 847 nodes for a total of 13,552 cores, as well as 2 GB per core of memory. An extended machine specification is provided in Table 3.5.

Interconnect Benchmarks

Figure 3.6 presents the outcome of ping-pong timings for both the Intel MPI Benchmark and SKaMPI benchmarks; the SKaMPI benchmark reports the full round-trip time of a ping-pong benchmark, while the Intel MPI benchmark only returns half the round-trip time. Once halved, the measured SKaMPI results correlate with the IMB results in Figure 3.6(a), albeit with the Intel MPI Benchmark exhibiting more noise variation past approximately 12,000 bytes. The drop at ≈12,000 bytes could potentially be attributed to a switch in MPI behaviour, such as that experienced by a shift between an eager protocol, where the acknowledgement of a waiting receieve is not required (possibly neccesitating

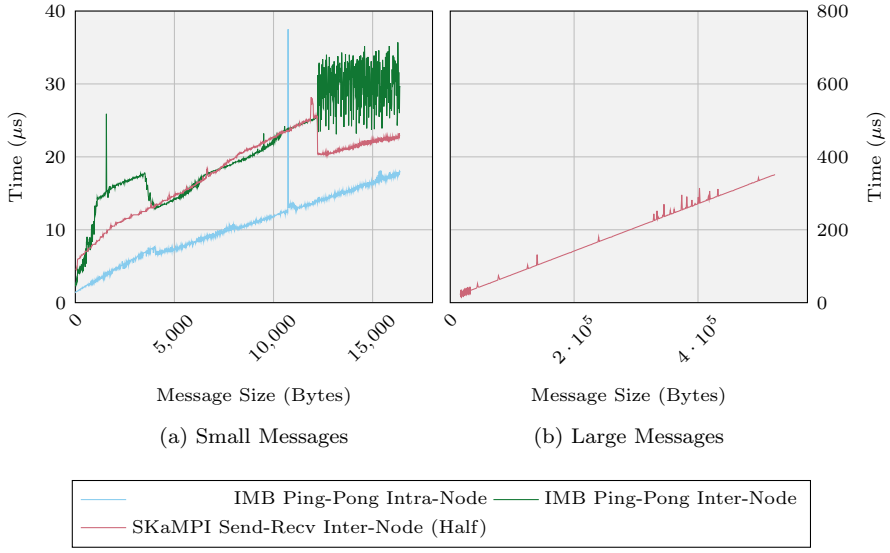


Figure 3.6: Point-to-Point Timings, Intel MPI Benchmark/SKaMPI

the use of a buffer and extra copy), to a rendevzous protocol where such an acknowledgement is neccessary. As with many of the other benchmarks, at large message sizes the relationship between the time taken and message size is largely linear.

3.3.5 Intel X3430 workstation

This machine, a 2.4GHz Intel X3430 workstation, is not used in the vast-majority of benchmarks, nor for any timings runs. However, due to the lack of select PAPI counters, primarily the L1 Data Cache Hit or Access measurements, it is used to obtain these values to provide an approximation of an application's behaviour for these metrics. It is presumed measurements such as hit rates are not directly translatable between machines due to differences in cache sizes and/or behaviours. However metrics such as total accesses may be tied more closely to an application rather than a machine since accesses represents the sum of the hit and miss rate and thus, as long as a cache miss does not also register as an additional hit, the ratio of these two characteristics does not

matter. The access count from one machine could potentially then be used to compute the potential cache hits on another by subtracting the measured miss rate of the second machine.

However, while these derived numbers can potentially be useful for identifying trends/behaviours, the translatable nature of the L1 Data Cache Access count is an assumption and cannot neccesarily be considered accurate for the target machine. While all other counters are measured directly from a target machines (such as L1 Data Miss Rate, L2 Hit Rate etc.), the L1 Data Accesses/Hits are always provided by the X3430 Workstation. Without the existence of such counters on different architectures such as Sandy Bridge or Haswell, it is difficult to verify how well such numbers translate. Appendix B.3 attempts to address some of the issues surrounding the use of PAPI counters amongst different architectures, including the accuracy/validity of Floating-Point Operations per Second (FLOP/s) and the use of L1 Data Cache Access rates. Despite this, the method is adopted since no hardware based alternative are available for such machines if the relevant counters are not available for a chipset, and such counters are still sufficient to identify trends of interest when contrasting between kernels (since such readings are all from the same machine).

3.4 Summary

This section has introduced a variety of tools and machines used within this work for the purposes of empirical investigation. It highlights some of the core benchmark characteristics, and tools necessary to reveal insightful performance details about the behaviour of a code. In the remainder of this work, these tools are applied to the task of performance analysis and optimisation, exploring how the applications introduced in this chapter behave in parallel environments and how this knowledge can be applied in a predictive capacity.

CHAPTER 4

Performance Scaling of a Near-Neighbour Hydrodynamics Application

Hydrodynamics is a domain of science belonging to the field of Computational Fluid Dynamics (CFD), specifically addressing the behaviour of fluid or fluid-like substances in motion across a passage of time within a spatial domain. These behaviours can be modelled computationally through the use of physical laws/equations that represent fluid behaviour.

Predicting the dynamic behaviour of materials as they flow under the influence of high pressure and stress is of considerable importance to understanding weapons. Without recourse to underground testing, access to experimental hydrodynamics facilities and supporting high-performance simulations has an important role in providing data to assess weapon safety and performance. Hydra is a benchmark 3D Eulerian structured mesh hydrocode implemented in Fortran, with which the explosive compression of materials, shock waves, and the behaviour of materials at the interface between components can be investigated.

The ultimate goal of any High Performance Computing (HPC) application is to provide accurate results, yet it is implicitly acknowledged that it is desirable for these results to be obtained as *quickly* as possible. Given the possible variance in machine configuration, both software and hardware, understanding the behaviour of the applications in question is crucial to both quick execution of said application and knowing how its performance might be impacted by modifications in the future. This can be further enhanced by the use of *performance models*, mathematical or simulation-based systems that are capable of capturing an application's core behaviours and predicting its runtime.

This work sets out to construct an analytical performance model of Hydra,

an application of interest. However, in order to do so, a greater understanding is required of the application itself. This chapter introduces Hydra, investigating its current strong and weak-scaling performance with respect to both its code structure and its use of the differing machine components such as compute resources, point-to-point communications, collectives etc. It also highlights any unusual behaviours that may be of interest in the model construction or optimisation process.

Specifically, this chapter sets out to achieve the following goals:

- Introduce Hydra, a hydrodynamics benchmark provided by the Atomic Weapons Establishment (AWE), describing its structure, critical path and communication patterns;
- Investigate the parallel scaling performance of Hydra, including serial compute, strong-scaling and weak-scaling performance on a large scale machine — part of this work is published prior in 2011 [44];
- Identify performance influencing factors that can guide modelling and optimisation efforts, including any unusual discrepancies that warrent futher investigation.

4.1 Hydra

The Hydra benchmark code simulates a cube of mixed materials under stress by discretising the data onto a 3D grid of cells given by $N_x \times N_y \times N_z$ and using message passing for parallelisation. The 3D cube of data is decomposed onto a number of processing elements (PEs) in a typical Single Program Multiple Data (SPMD) fashion during execution. By representing the spatial volume as a collection of cells, the physical properties of materials at different cartesian locations within the grid can be quantified. The benchmark can then reflect delta changes in the value of these properties as the time progresses throughout the course of a simulation.

To achieve this goal the simulation executes a series of functions that are each responsible for updating different simulated properties. The rate of progress is delineated by Δt, the amount of simulated time that has passed since the last update. A single pass of this collection of functions is known as an *iteration*. Repeated iterations of this series of functions progresses the simulated time, with the benchmark terminating once the sum of Δt values across all iterations reaches a preconfigured amount. Large Δt values progress the simulation faster but lead to a loss of detail, potentially becoming too course-grained to be an accurate simulation. Small Δt values avoid this loss of detail, but increase the overall runtime and may offer little benefit to accuracy if the grid is not sufficiently refined/discretised to a point where any differences are appreciable. To mitigate this, Δt can change from iteration to iteration and is determined by the current state of the simulation; a suitable value is computed at the beginning of every iteration. The total number of iterations executed is determined by the amount required for the sum of Δt values to reach a preconfigured total.

From this it can be determined that two properties dictate the overall runtime of the benchmark — the time taken to run a single iteration, and the number of iterations to run to completion. Given its repetitive nature, identifying the critical path across the course of an iteration becomes key to understanding the performance of Hydra. As a parallel program, during the course of its execution the functional components of Hydra can be summarised as falling into one of a number of different categories tied to the use of various machine components (e.g. memory or network interconnect), therefore a constructive breakdown of the various sub-functions called during the course of an iteration is required. The five categories identified within this work are as follows:

- **Memory Management** — Functions responsible for the dynamic allocation of large temporary arrays (Section 4.2.3).
- **Compute** — Kernels that perform computational operations (Sections 4.2.4 and 4.2.5).

- **Update Boundary** — Specialised kernels used to update the problem boundary cells of the grid (Section 4.2.6).
- **Point-to-Point Communications (Exchange)** – MPI Point-to-Point communications, such as Send and Recv, used to communicate data between two MPI processes (Section 4.3.2).
- **Collective Communications** — MPI functions, such as MPI Allgather or MPI Allreduce, that provide gather/scatter operations to communicate data across a set (potentially all) of the available MPI processes (Section 4.3.3)

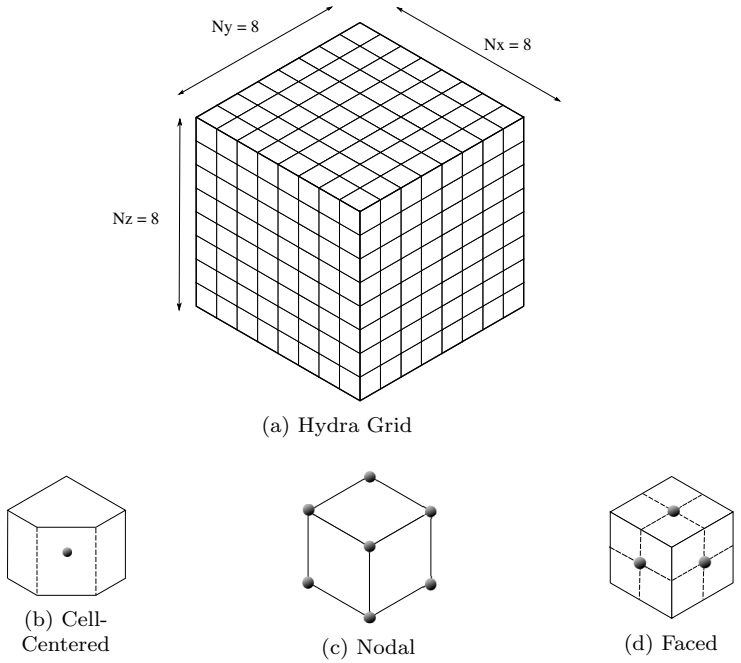
The significant details of each operation is provided in more depth in Sections 4.2 and 4.3. It is necessary to provide a distinction between them for the purposes of separating each of the functions into their different components, important when distinguishing between different performance behaviours, especially in a parallel environment.

4.2 Serial Behaviour

This section introduces the serial behaviour of Hydra, focusing on the application behaviours that influence performance in the absence of parallel considerations. Doing so will reveal how Hydra's walltime can be tied to the problem configuration and the structure of its compute kernels.

4.2.1 Structured Mesh

To simulate a hydrodynamic system, the problem space is discretised into cells. The segmentation of the problem space influences both the accuracy and speed to solution; the greater the number of cells the more refined the solution becomes. Since computation must occur for each cell position within the grid, accuracy is increased but requires a more significant amount of computation.



(a) Hydra Grid

(b) Cell-Centered

(c) Nodal

(d) Faced

Figure 4.1: An $8 \times 8 \times 8$ Cell Structured Mesh

These cell decompositions are known as meshes and can be structured, unstructured or hybrid in nature.

Structured meshes consist of a regular pattern, with a well-defined neighbour relationship between cells. The cells are typically quadrilateral (2D) or cuboid (3D) in shape. Such meshes implicitly store information regarding cell neighbours as part of their data structure, with the indexing of a 2D or 3D data array acting as a cartesian co-ordinate system from which lookups can be performed, making them relatively memory efficient.

Unstructured meshes are irregular in nature, with variable cell shapes, resulting in a more ill-defined neighbour lookup for an arbitrary cell. As such, they must also store neighbour relationship data, making them more memory inefficient.

Hybrid meshes incorporate both structured and unstructured components, possessing regions that can be one of either approach resulting in an overall decomposition that consists of both variants.

Hydra's regular, spatially discretised grid is one such example of a structured mesh. Its problem size is determined by two components, the spatial size and the cell size. The spatial property defines the simulated physical size of the problem. The cell count however determines the number of cells this physical space is decomposed into — e.g. with 100 cells each cell represents $1/100th$ of the simulated physical space. The majority of compute kernels within Hydra are tasked with operating upon every cell within the grid; consequently the greater the number of cells, the more significant the impact upon compute/memory performance.

During the course of the simulation an iterative solve refreshes a collection of simulated physical properties, termed *quantities*, each of which has a distinct value stored per cell. These quantities fall into one of three different classifications — cell-centered, nodal or faced:

- Cell-Centered (Figure 4.1(b)) — Oriented at the centre of a cell.
- Nodal (Figure 4.1(c)) — Oriented at the vertex of a cell.
- Faced (Figure 4.1(d)) — Oriented at the centre of a cell face. This is the equivalent of a nodal quantity in one dimension, and of a cell centered quantity in the remaining two dimensions.

These classifications influence the storage requirements and the amount of work required to process them. Each quantity has its own grid of data, and multiple quantities are updated per cell at different stages during the course of a Hydra iteration. The data for each quantity is stored in a 3D Structure-of-Arrays (SoA) format.

4.2.2 Mixed Cells

As a mixed material simulation, each material is associated with their own distinct values per relevant quantity. Those quantities that are mutually exclusive from individual materials are known as *pure quantities*, for which only one value

is required per cell, while those consisting of a property of a material are known as *mixed quantities*, and must have a separate value per material, per cell. Cells that possess only a single material at a given simulation time are known as *pure cells*, while those with more than one are known as *mixed cells*. Whether a cell is pure or mixed has implications for both storage and compute overheads. Mixed cells require additional processing and storage per cell, with its time taken tied to not only the number of cells but the number of mixed materials within the system. This can complicate the analysis and prediction of performance, as whether a cell is pure or mixed is dependent directly on the state of the simulation, a nebulous state for any given series of inputs that cannot be determined without executing the simulation itself. While it is possible to track the state of mixed and pure cells through the use of tracing/simulation history, this work focuses primarily on the execution of pure cell runs only to simplify the initial modelling process. Doing so allows us to establish a baseline performance that identifies links between the grid size, the application behaviour and the machine hardware in the absence of complicating factors such as state dependant mixed cell handling. This remains an avenue for future exploration, where the load-balancing of mixed cells across processes remains a factor of interest, but is not considered within the scope of this work.

4.2.3 Memory Management

Each quantity in Hydra requires storage space to preserve data across iterations — this is typically stored in a 3D array, indexed by grid coordinates. The majority of such quantities are permanent — allocated once and only deallocated at the completion of a run. However, a number of functions require the use of intermediate values — computation that is reused within a function or across a sub-set of functions, but does not need to be maintained between iterations. While in some cases this can be a small 1D array or even a temporary variable, in some cases the use of an additional 3D array is required. As such, a small portion of the runtime is allocated to the creation and destruction of these tem-

Classification	Grid Size
Cell-Centered	$N_x \times N_y \times N_z$
Nodal	$(N_x + 1) \times (N_y + 1) \times (N_z + 1)$
Faced in X Dimension	$(N_x + 1) \times N_y \times N_z$
Faced in Y Dimension	$N_x \times (N_y + 1) \times N_z$
Faced in Z Dimension	$N_x \times N_y \times (N_z + 1)$

Table 4.1: Quantity grid sizes for a $N_x \times N_y \times N_z$ problem

porary arrays. By allocating and deallocating only when needed, the maximum watermark of memory used is reduced at the cost of a hit to performance.

4.2.4 Grid Kernels

The amount of data stored in memory (and thus processed during computation) is related to the overall size of the grid, with slight variation depending on the type of the quantity in question. A grid kernel is a kernel that iterates over every cell point within the grid, and thus its performance is directly tied to the number of data-points that must be processed. The size of the grid for each of the different quantity types is summarised in Table 4.1. From this it can be seen that any variation in grid size between the quantities is at its most significant for very small problem sizes, but as a whole the general problem size is dominated by the size of the grid ($N_x \times N_y \times N_z$).

In addition to the number of cells in any dimension, the grid also has a spatial size associated with each dimension, such that each cell represents a portion of the overall spatial volume. The parallel scaling investigations in this work typically keep such values fixed per cell within a set of experiments for consistency, though such values influence the simulation state, and indirectly Δt, rather than a kernel's compute time.

4.2.5 Stencil Kernels

Stencil kernels possess similarities to grid kernels, in that they also typically iterate over every cell point within the grid. However, they possess unique characteristics that can complicate any computation. As well as using data associated with the cell of interest, stencil kernels also require data from one or

more neighbouring cells for computation. This can introduce additional data-dependencies with corresponding restrictions when performing compute. For example:

- In-order array updates cannot occur immediately if there exists an uncompleted stencil operation in a neighbouring cell that requires the original data. This necessitates the use of out-of-order updates and additional temporary arrays;
- In some circumstances the computation depends on data not local to the current process (Section 4.3.2);
- The use of stencil kernels can introduce more irregular memory access patterns with regards to both spatial and temporal locality. This can become more prominent as the size of the arrays in all three dimensions increases.

Hydra possesses a number of these stencil kernels, though ultimately their performance is still tied to the grid size. Unless otherwise noted, in this work they are treated in a similar manner to grid kernels for the purposes of analysis and modelling.

4.2.6 Update Boundary Kernels

The use of stencil kernels can require neighbouring cells in one or more dimensions in order for them to be computed successfully. However, in the case of cells on the boundary of the local grid, no such neighbour cell exists within the context of the problem space, introducing a potential unhandled scenario.

To tackle this, the use of *ghost* cells, cells that extend past the boundary of the local grid, is necessary. The number of cells that extend past the boundary of the grid is known as a *halo* — e.g. a halo of two adds two ghost cells past the boundary in a given dimension. These ghost cells can be populated with appropriate data to ensure that the stencil computation resolves to an accurate resolution without introducing errors into the simulation.

The data stored in ghost cells for internal boundaries is retrieved from remote processes as part of the point-to-point data exchange described in Section 4.3.2. The task of populating the ghost cells for external boundaries however falls to a small collection of kernels, described within this work as *update boundary* kernels. Such kernels do not iterate across the entire grid of cells; rather, they focus primarily upon only those cells that form the outer faces of the grid (the external boundary of the problem space) and the neighbouring ghost cells. Every point-to-point exchange stage is followed by an update boundary stage that must refresh these external boundary ghost cells.

It is noted here that the definition of an external boundary cell is not restricted to just the very outermost cells of the grid in any dimension, for example, if a stencil operation requires a neighbouring cell of up to two cells away, a halo of two is required. In this context any cell within two cells of the edge of the grid would have a dependency upon a ghost cell past the external boundary, and the kernels would have to process two faces per boundary rather than one.

4.3 Parallel Behaviour

With the additional considerations of communication overheads and fixed synchronisation points, a critical path of Hydra's performance hotspots can be constructed, informing future performance analysis and predictive modelling efforts.

4.3.1 Decomposition

The decomposition of the dataset attempts to distribute the problem as evenly as possible between the available Processing Elements (PEs). Given P processing elements, the problem will be decomposed on to a processor grid of $P_x \times P_y \times P_z$ such that a local cell grid of size $N_x/P_x \times N_y/P_y \times N_z/P_z$ will be stored by a single PE. The decomposition is achieved by finding the factors of P where the grid is partitioned successively, favouring decomposition in the

Cores	Case	PE Decomp.			Cells			Local Decomp.
		Px	Py	Pz	Nx	Ny	Nz	
2048	Power of 2	16	8	16	160	160	160	$10 \times 20 \times 10$
1000	Integer cube root	10	10	10	100	100	100	$10 \times 10 \times 10$
1650	Three factors	10	11	15	165	165	165	$16/17 \times 15 \times 11$
817	Two factors	1	19	43	817	817	817	$817 \times 43 \times 19$
2003	Prime number	1	1	2003	2003	2003	2003	$2003 \times 2003 \times 1$

Table 4.2: Sample P_x, P_y and P_z values at scale [44]

dimension with the highest cell count to produce as "cubic" a local grid as possible. In the case of a cubic grid of equal length in all dimensions, the application favours the order z, y, x with the exception of powers of 2, where the order is adjusted to y, z, x. Table 4.2 illustrates example decompositions for various cases of P, assuming a global grid with dimensions of equal length.

In the event that the division has a remainder, the remaining cells are spread evenly across the processes in that dimension, for example, let $N_x = 53$, $P_x = 3$. The base number of cells per process is $53/3 = 17$, with a remainder of 2. The two remaining cells are spread across the first two processes in an X row of processes, resulting in a total cell decomposition in the X dimension of $18, 18, 17$ for each PE respectively.

4.3.2 Point-to-Point Communications

The use of stencil kernels within Hydra necessitates that there will exist cases where requested data is stored on a remote process. The purpose of point-to-point communications is to directly obtain any required data from these processes, while simultaneously also providing any data they require from the local process. Due to the nature of these stencil kernels, the set of processes that possess this data is limited to only neighbouring processes in a cartesian layout of the process allocation, making it a subset of the overall total set of processes with a cap on the maximum number of processes that can be within this set.

Within Hydra the communication pattern is restricted to the six immediate process neighbours that share a grid face with the local process. While data

Listing 4.1: MPI Point-to-Point Data Exchange – Psudeocode

```
for d in {X,Y,Z} {
  for f in faces(d) {
    for dt in datatypes {
      PackMessage(stage,d,dt)
    }
  }
  for f in faces(d) {
    for dt in datatypes {
      MPI ISend(f,d,dt)
      MPI IRecv(f,d,dt)
    }
  }
  MPI Waitall
  for f in faces(d) {
    for dt in datatypes {
      UnpackMessage(stage,d,dt)
    }
  }
}
```

is potentially required from processes that share a corner, i.e, diagonal neighbours, this data is obtained via proxy from one of the immediate face-sharing neighbours due to the order of communication in a Hydra point-to-point communication process, referred to within this work as an *Exchange* stage, where for each neighbour all relevant data is packed into a single message, communciated via the use of MPI ISend/IRecv functions, and unpacked by the receiving process to be placed into the appropriate ghost cells.

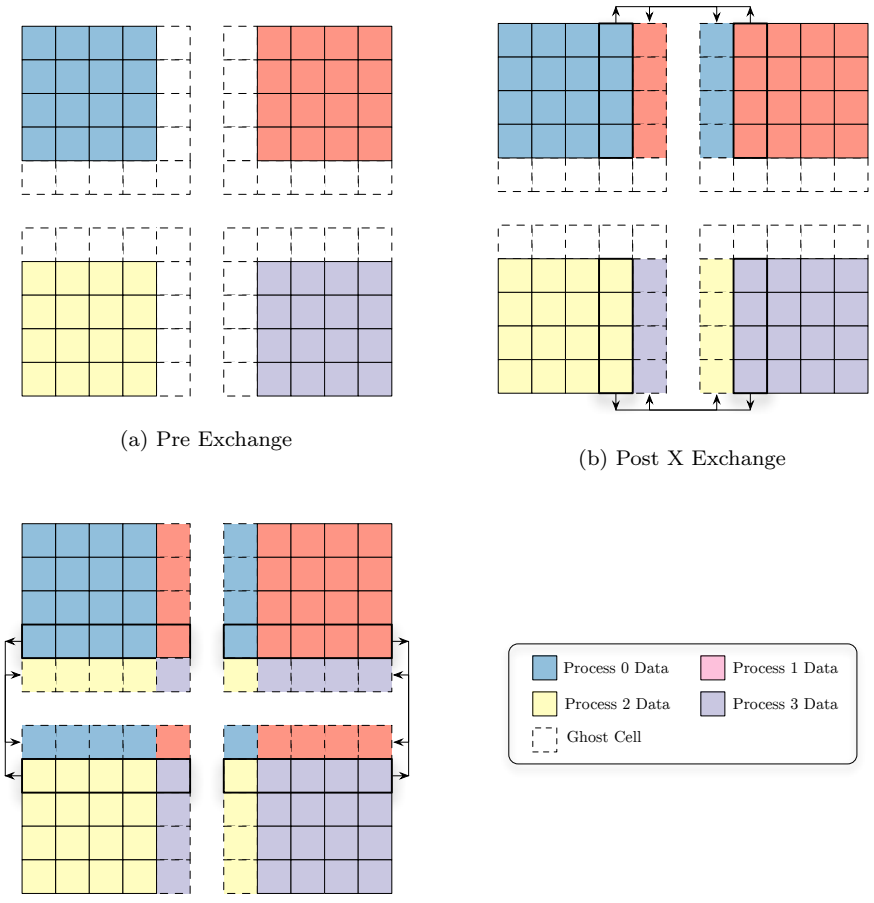
This workflow of packing, communicating and unpacking is performed for each dimension of communication in a strict order, with the X dimension running to completion before processing the Y and then the Z dimension messages, as summed up in Listing 4.1. For each dimension, the workflow consists of:

- Construct a set of messages for communication — one per valid face (i.e. those with a valid process neighbour up to a maximum of two for each direction), per datatype (e.g. integer, double). Each of these messages contains the data for all relevant quantities packed into a single buffer;
- Initialise the sending/receiving of these messages via the use of MPI non-blocking primitives;

- Halt further progress until all sends and receives for the local process are complete;
- Unpack the received data in a reverse manner to that of the packing stage, populating ghost cells on the local MPI process.
- Repeat for each remaining dimension.

The size of these point-to-point messages is determined by which exchange stage is being performed; different exchange stages require data from different arrays depending on the position within the overall iteration. There exist 5 distinct communication stages in Hydra, which are distinguished in this work by the unique labels *Lartvis*, *Mlagh(1)*, *Mlagh(2)*, *Madv*, and *Madvm*. The location of their respective function calls is detailed in Section 4.4. Further to this, the size of each message is also influenced by the number of ghost cells required. A stencil kernel that requires data from a neighbouring cell up to two cells away requires the communication of up to two faces worth of data rather than one to populate the local processes' ghost cells (a halo of two). No communication with diagonal neighbours is performed directly — obtaining data from these processes is achieved indirectly via the inclusion of retrieved ghost cells from prior communications. Figure 4.2 is a 2D example of such an exchange.

Figure 4.2(a) presents the initial process layout prior to any communication — a simplified example is provided here that only consists of a single data array for consideration. The ghost cell data is not coherent with the remote processes, and must be refreshed from the neighbours before any further compute can continue. Each process has a single neighbour in both the X and Y dimensions. Figure 4.2(b) is the state of the communication stage after the X dimension communications are complete, but before any Y dimension communications have been conducted. No ghost cells have been included in the sent message, as prior to the exchange no relevant data was stored in them, meaning only a single face of data was transmitted. This changes however for the Y dimension communications. The face data transmitted in the Y dimension also extends



(a) Pre Exchange

(b) Post X Exchange

(c) Post Y Exchange

Figure 4.2: Hydra 2D Message Exchange — 2×2 Decomposition

into the ghost cells, including a small amount of data that was received from the X dimension communications. This results in the outcome portrayed in Figure 4.2(c). This is notable since, as is apparent in the diagram, it can be seen that data has been transmitted from a diagonal neighbour indirectly via an immediate neighbour. While not included here, it is noted that the same behaviour is observed for Z dimension communications, where ghost data from both the X and Y dimensions is incorporated into its messages.

From this, the most notable features of the data exchange process are as follows:

- There exists an explicit data dependancy within the exchange workflow — Y dimension communications cannot begin until all X dimension communications have completed for a process, with the same relationship applicable between the Z and Y dimensions;
- Different stages have different data sizes, and thus influence the performance of Hydra differently. However, they all observe the same underlying communication pattern;
- The number of messages for an individual process is capped by the number of face sharing neighbours.

4.3.3 Collective Communications

As well as point-to-point communications, there are also a number of collective operations in use — MPI operations that communicate with all processes at once rather than just a subset. A consequence of this attribute is that they can threaten to scale in cost as the total number of processes increases. Therefore, capturing them is necessary for any work which might involve a significant number of processes, including the modelling of future large-scale architectures.

The predominant form of collectives in Hydra is the use of AllReduce, a many-to-many operation where each process contributes a portion of data, an operation such as sum/min/max is applied and the result is distributed to all processes.

4.4 Function Breakdown

This section introduces the core functions of Hydra that make up the critical path. These functions form the dominant performance hotspots, and make use of multiple different machine components during their execution.

Hydra Iteration

Listing 4.2: Single Hydra Iteration — Pseudocode

```
 1 Allocate Memory(MDT)
 2 Call MDT
 3 Deallocate Memory(MDT)
 4 Call ShortPrint
 5 Allocate Memory(Mlagh)
 6 Call Mlagh
 7 Deallocate Memory(Mlagh)
 8 Allocate Memory(Madv)
 9 Call Madv
10 Deallocate Memory(Madv)
11 Call ShortPrint
```

Listing 4.2 introduces the critical path as identified through profiling and source-code analysis. In addition to these major functions, there also exist communication *stages*. There are repeated instances during the course of an iteration where compute cannot begin until the completion of both a communication step and an update boundary step is reached. The quantities communicated and processed during these steps depends upon what immediate data-dependancies must be resolved for computation to continue, varying depending on what point has been reached within the iteration. These different stages are distinguished by attributing unique identifiers to each as part of the following function descriptions:

Dynamic Memory Handling (**Listing 4.2: Lines 1, 3, 5, 7, 8, 10**)

The MDT, Mlagh and Madv primary functions all use temporary data arrays (typically 3D) to store the intermediate results of computation, necessary for the execution of that function but containing no data that must be preserved across functions. Rather than preserve all of these arrays in memory simul-

taneously, they are allocated and deallocated before and after each function call to conserve space.

MDT **(Listing 4.2: Line 2)**

Listing 4.3: MDT Function — Pseudocode

```
1 Compute: Kernel 1 (Grid)
2 Compute: Kernel 2 (Grid)
3 Call: Leosdrv
4 Call: Lartvis
5 Communications: AllGather × 23
```

This function is responsible for calculating the Δt value for the current iteration. Multiple Δt values are computed from the current simulation state, selecting the global minimum as the Δt value for the current iteration to determine the maximum amount of change permissible within the simulation iteration. If outside predetermined minimum/maximum bounds, one of these bounds is selected as appropriate.

Mlagh (**Listing 4.2: Line 6**)

Listing 4.4: Mlagh Function — Pseudocode

```
1 Compute: Kernel 1 (Grid)
2 Communications: Data Exchange(Mlagh(1))
3 Compute: Update Bounds(Mlagh(1))
4   for i in 0 → mlag  {
5     Compute: Kernel 2 (Grid)
6     Compute: Kernel 3 (Grid)
7     Communications: AllGather × 1 (1 int)
8     Compute: Kernel 4 (Grid)
9     Compute: Kernel 5 (Grid)
10    Compute: Kernel 6 (Grid)
11    Communications: Exchange(Mlagh(2))
12    Compute: Update Bounds(Mlagh(2))
13    Compute: Kernel 7 (Stencil)
14    Compute: Kernel 8 (Grid)
15    Compute: Kernel 9 (Boundary× 6 Faces)
16    Compute: Kernel 10 (Grid)
17    Compute: Kernel 11 (Grid)
18    Communication: AllGather × 1 (1 int)
19    Compute: Kernel 12 (Grid)
20    if(i != 0) {
21      Compute: Kernel 13 (Grid)
22      Call: Lartvis
23    }
24    Compute: Kernel 14 (Grid)
25    Compute: Kernel 15 (Grid)
26    Compute: Kernel 16 (Grid)
27    Compute: Kernel 17 (Grid)
28    Compute: Kernel 18 (Grid)
29 }
30 Compute: Kernel 19 (Grid)
31 Compute: Kernel 20 (Grid)
32 Compute: Kernel 21 (Grid)
33 Call: Mvolflx (Grid)
```

Within the Mlagh function exists an internal loop governed by a loop bound term, *mlag*. The value of *mlag* can vary from iteration to iteration, depending on the state of the simulation, with a minimum of 1 and a maximum defined by preset constraints. As such, the Mlagh function can exhibit some variability in walltime between iterations as, for example, an iteration where *mlag* equals one takes less time to execute than an iteration where *mlag* equals two.

Madv (**Listing 4.2: Line 9**)

Listing 4.5: Madv Function — Psuedocode

```
1 Compute: Kernel 1 (Stencil)
2 if(iteration step is even) {
3   Communications: Exchange(Madv)
4   Compute: Update Bounds(Madv)
5   Call: Madvx
6   Communications: Exchange(Madv)
7   Compute: Update Bounds(Madv)
8   Call: Madvy
9   Communications: Exchange(Madv)
10  Compute: Update Bounds(Madv)
11  Call: Madvz
12 }
13 else {
14  Communications: Exchange(Madv)
15  Compute: Update Bounds(Madv)
16  Call: Madvz
17  Communications: Exchange(Madv)
18  Compute: Update Bounds(Madv)
19  Call: Madvy
20  Communications: Exchange(Madv)
21  Compute: Update Bounds(Madv)
22  Call: Madvx
23 }
24 Compute Kernel 2 (Grid)
25 if(κ) {
26  Call: Lartvis
27  Compute Kernel 3 (Grid)
28 }
29 Compute Kernel 4 (Grid)
```

This function governs the advection updates of multiple quantities. This is conducted one dimension at a time via a number of sizeable compute kernels and communication exchange phases; this typically constitutes the bulk of the runtime. It calls a number of smaller advection functions, *Madvx*, *Madvy* and *Madvz* that each operate on the X, Y and Z dimensions respectively. κ represents a pre-defined parameter than can be enabled or disabled in the simulation input.

Madvx/Madvy/Madvz (**Listing 4.5: Lines 5, 8, 11, 16, 19, 22**)

Listing 4.6: Madv{x/y/z} Function — Psuedocode

```
1 Compute: Kernel 1 (Grid)
2 Compute: Kernel 2 (Stencil)
3 Call: Madvm(x/y/z)
```

Madvmx/Madvmy/Madvmz (**Listing 4.6: Lines 3**)

Listing 4.7: Madvm{x/y/z} Function — Psuedocode

```
1 Communications: Exchange(Madvm(x/y/z))
2 Compute: Update Bounds(Madvm(x/y/z))
3 Compute: Kernel 1 (Grid)
```

The *Madv(x/y/z)* and *Madvm(x/y/z)* collection of functions provide the majority of the overall advection functionality of the Madv function.

Lartvis (**Listing 4.3: Line 4, Figure 4.4: Line 22, Figure 4.5: Line 26**)

Listing 4.8: Lartvis Function — Psuedocode

```
1 Communications: Exchange(Lartvis)
2 Compute: Update Bounds(Lartvis)
3 Compute: Kernel 1 (Stencil)
```

Called during the operation of a selection of other functions, the Lartvis function is notable for it potentially requiring a refresh more than once per iteration.

ShortPrint (**Listing 4.2: Lines 11**)

Listing 4.9: ShortPrint Function — Psuedocode

```
1 Compute: Kernel 1 (Grid)
2 Communications: MPI AllGather    ×7 (1 int)
3 Communications: MPI AllGather    ×42(1 double)
4 Communications: Vector AllGather ×9:
5                   MPI AllGather  ×9 (1 int) +
6                   MPI AllGatherv ×9 (1 double)
7 Communications: MPI AllGather    ×1 (32 chars)
```

The ShortPrint function is responsible for collecting and displaying summary data every iteration.

4.5 Scaling Behaviour

This section introduces the outcome of several investigations into the scaling behaviour for both problem size and process count, revealing a number of behaviours of interest.

Number of Cells X	Y	Z	Total Iterations	Mlag Iterations 1	2	3	4	Walltime (s)	Standard Error
30	30	30	209	209	0	0	0	11.18	0.03
50	50	50	209	193	16	0	0	50.94	0.14
80	80	80	210	169	17	10	14	205.09	0.27
100	100	100	217	157	18	10	32	418.70	0.31
120	120	120	229	148	17	10	54	809.37	0.81
150	150	150	258	136	18	10	94	1941.77	0.63

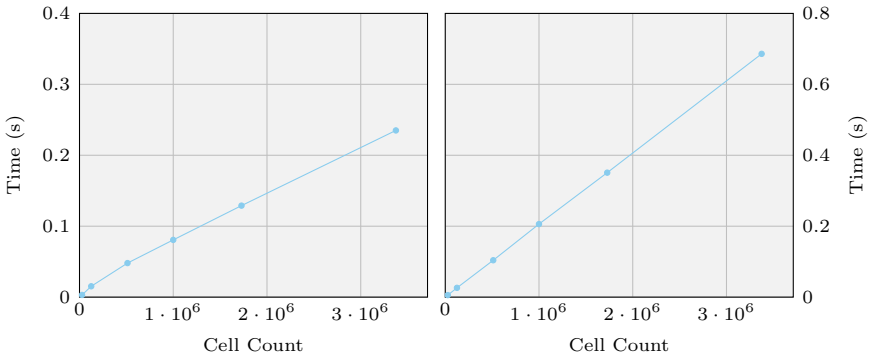
Table 4.3: Minerva, Hydra Serial Walltimes

4.5.1 Serial Results

This chapter has established a variety of potentially influential input parameters such as the problem mesh size, number of processing elements, decomposition, etc. In order to distinguish between the influence of these characteristics, only serial executions are initially examined — this eliminates those factors that are affected by communication overheads such as MPI point-to-point messages or collectives, focusing upon the impact of the grid size on compute performance. It is expected that in a code dominated by grid kernels a roughly $1:1$ scaling performance with the cell count would be exhibited, assuming that the work per cell is consistent. To this end, the serial performance on Minerva is explored, covering a range of different global problem sizes.

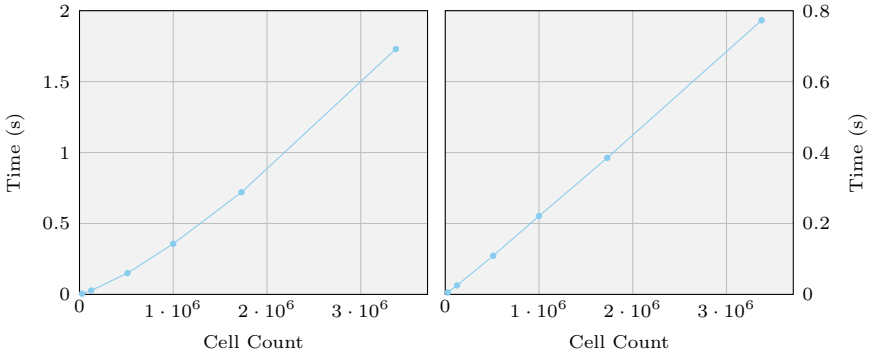
The outcome of these experiments is presented in Table 4.3 — the number of cells in the X, Y and Z dimensions are scaled equally to maintain a cubic shape. As well as the overall walltime, the times of individual kernels and library calls are also captured. The instrumentation captures the entirety of the critical path within 0.5% of the measured walltime (a breakdown by function is provided in Table B.2, Appendix B.2), enabling the identification of Hydra's hotspots with confidence and confirming the inital assessment of Hydra's critical path (at least for serial runs) presented in Section 4.1.

Before the scaling performance of the serial experiments can be fairly compared, a variable behaviour that is an outcome of changing the problem size must first be accounted for. The number of iterations to reach an experiment's conclusion is inherently tied to the state of the experiment — if the Δt values for an experiment are smaller, the number of iterations taken to simulate the same



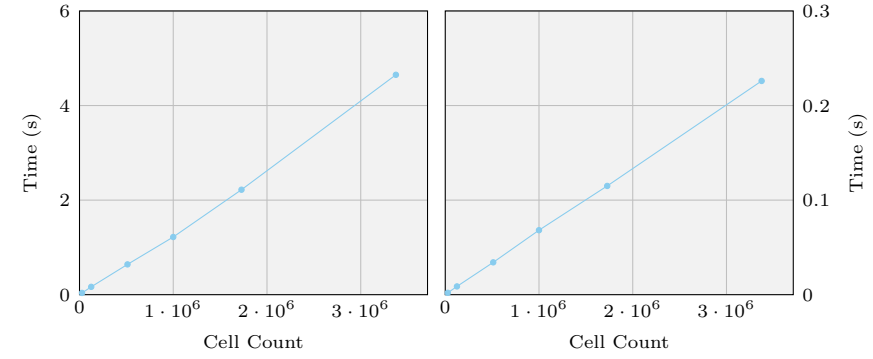
(a) Memory Management

(b) MDT



(c) Mlagh

(d) Mlagh Per Internal Mlag Loop



(e) Madv

(f) Shortprint

Figure 4.3: Hydra Function Mean Walltime per Iteration

amount of time is increased, in turn increasing the overall walltime. Likewise, the state of the simulation also affects *mlag*, the number of loops internal to the Mlagh function for a given iteration (see Figure 4.4 — lines 4-29). The count of both of these values for these experiments is provided in Table 4.3, from which it can be clearly seen that there exists a degree of variability (the total number of *mlag* iterations provided is inclusive of the total number of iterations as a factor, since there are one or more *mlag* iterations per global iteration). Figure 4.3 provides a breakdown by major function, normalised for a single Hydra iteration to compare the performance of the individual functions. From these values it can be seen that there exists a mostly linear relationship between the total cell count and their respective walltimes. However, there exist a few notable deviations:

- The Mlagh function has a noticeable curve — this however is attributable to the variation in the inner *mlag* loop iterations. When this is refined to the time taken per inner *mlag* loop in Figure 4.3(d), the relationship is linear;
- The *Memory Management* functions are linear for the most part, with the exception of small cell counts where the performance is improved;
- The Madv function is not a perfect linear relationship — while it is linear at small problem sizes, larger problems take longer than might be projected, possibly as a consequence of one or more non-linearly scaling components amongst linearly scaling contributors.

Identifying the cause of the unexpected non-linear relationship for Madv at larger problem sizes is addressed later in this thesis (Section 6.2). However, even with these exceptions, the overall outcome of these serial investigations enables us to identify a relationship between the cell count, the overall compute performance and the various kernels that comprise Hydra's critical path. A parallel environment introduces the additional impact of communication overheads, but these serial results provide us with the means to distinguish between the

Process Count	Total Iterations	Mlag Iterations 1	2	3	4	Mean Walltime (s)	Standard Error
1	217	157	18	10	32	418.70	0.31
2	217	157	18	10	32	468.52	1.18
4	217	157	18	10	32	564.68	0.34
8	217	157	18	10	32	655.61	0.44
12	217	157	18	10	32	668.99	0.50
16	217	157	18	10	32	671.10	0.49
24	217	157	18	10	32	674.86	0.58
32	217	157	18	10	32	678.20	0.13
48	217	157	18	10	32	681.67	0.71
64	217	157	18	10	32	689.28	0.30
96	217	157	18	10	32	692.87	0.23
128	217	157	18	10	32	708.31	8.88
192	217	157	18	10	32	697.32	0.18
256	217	157	18	10	32	700.35	1.38

Table 4.4: Minerva, Hydra Weak-Scaling Walltimes (100^3, Node-Fill)

impact of increasing the cell count and other contributing performance factors. In the following sections, this analysis is extended by examining Hydra in a parallel context, namely a weak-scaled (Section 4.5.2) and strong-scaled (Section 4.5.3) setup.

4.5.2 Weak-Scaling Results

This section sets out to investigate the impact of a parallel environment by focusing upon a weak-scaled experiment to capture the details of communication overheads. The purpose of selecting weak-scaling over strong-scaling initially is twofold. First, with a fixed compute size per process, the outcome of the serial investigation suggests that for weak-scaling the compute times should be relatively consistent; this knowledge can be used to quickly identify any discrepancies and investigate further if warranted. Second, if the compute time is fixed, it enables a better focus upon the impact of the communication overheads in isolation from other factors (though it is problematic to separate the two entirely due to the intertwined nature of communication synchronisation and compute load-balancing). The overall global problem size is scaled in such a manner that the decomposition of the problem is evenly spread amongst the three dimensions — i.e. the cell count is increased so that it is as close to a cubic problem as possible for the available process count, leading to a similar cubic construction in the cartesian distribution of the processes and their neighbours.

The maximum number of internal loops (*mlag*) in an *Mlagh* call for a single global iteration is 4 for all weak-scaled experiments, with a minimum of 1. Across this set of weak-scaled experiments the spread of iterations that have 1 *mlag* iteration, 2 *mlag* iterations etc. is consistent, permitting fair comparisons of the walltime directly (unlike for the serial experiments).

Table 4.4 introduces a set of weak-scaled experiments with a fixed local size of 100^3, using process counts that are both powers-of-two and multiples of twelve. Process counts that are multiples of two are conducted using OpenMPI-1.4.3, with the initial set of experiments using a naïve approach for process allocation where a node is fully packed before process allocation begins on the next node. Due to an error with version 1.4.3 that prevented their execution, process counts that are multiples of twelve were conducted using OpenMPI-1.4.4 (the most similar version available). A comparison of the two versions for the same process count (powers-of-two only) is provided in Table B.1, Appendix B.1, to validate that this change does not unduly modify the performance of Hydra, given the similarity in walltimes. The iteration counts, total and for various *mlag* values, are the same for both multiples-of-twelve and powers-of-two process counts in these experiments.

From these results it can be identified that the absolute walltime does not remain consistent across all process counts at this scale. It is to be expected that initially as the scale increases, the communication overheads also increase, thus a rise in walltime is not unexpected. This behaviour is confirmed in Figure 4.4, where a breakdown is shown of the mean time spent in various critical path components across multiple runs, including the compute and communication costs. The selection of process minimum or maximum values for specific components captures the overall breakdown in a representative manner. Due to the cost of synchonisation, it is a reasonable expectation that the process that spends the most time in compute also has the minimum time spent idle in the communication stages, and vice-versa. Since the instrumentation of the communication stages also includes synchronisation time, as well as buffer pack/unpack

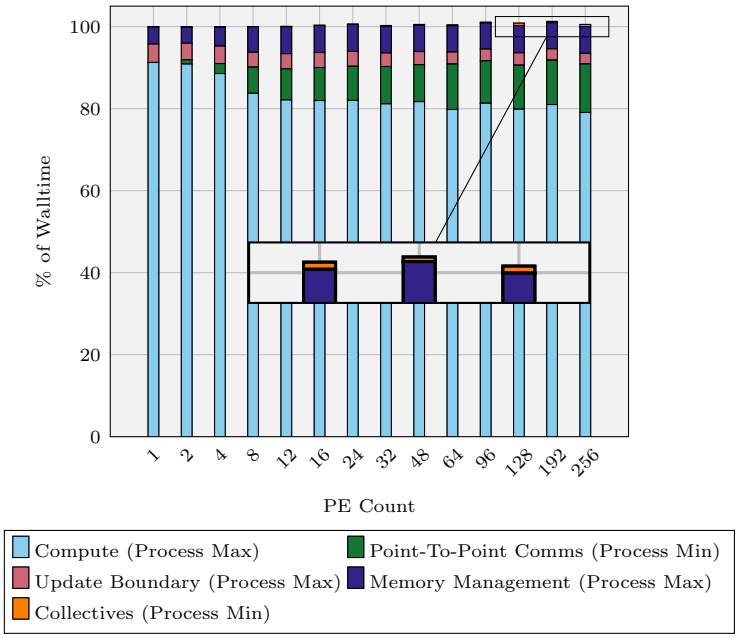
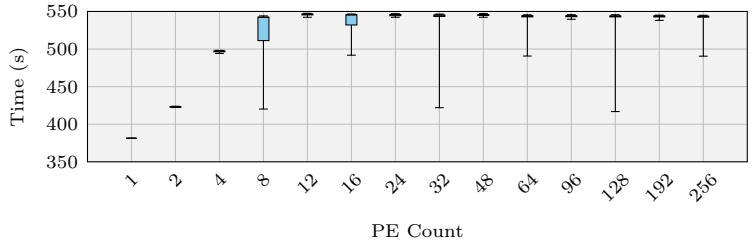


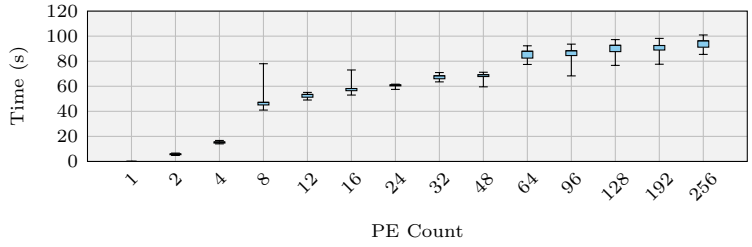
Figure 4.4: Max Walltime Breakdown — Weak Scaling — Minerva (Node Fill)

times, a summation of the maximum time spent in compute/memory components with the minimum time of the communication stages such as collectives or point-to-point data exchanges is used for the breakdown. This provides a good approximation of the overall walltime, with the total sum falling typically within 1% of the walltime of the maximum walltime across all process counts (resulting in a close to, but not exact, 100% of walltime measurement in Figure 4.4 due to a small degree of overlap as a result of using min/max across processes).

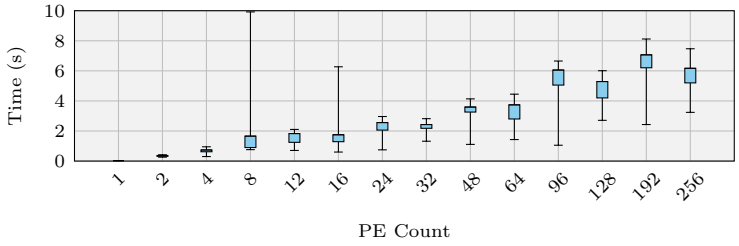
The node-fill approach to process allocation (i.e. no load-balancing) in these initial results has the potential to cause imbalance where the total process count is not divisible by the maximum number of cores per node. For example, in a 16 process setup the node configuration consists of two nodes, one with 12 processes and one with only four. These load imbalances manifest themselves as significant variability in the reported timings across the process counts. Figures 4.5(a) through 4.5(e) show the results of runs with the minimum walltime of all repetitions to minimise the impact of noise. The variability in performance



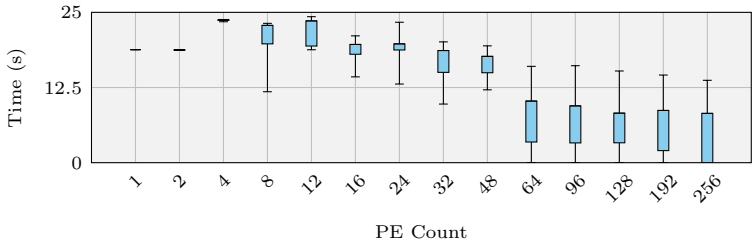
(a) Compute



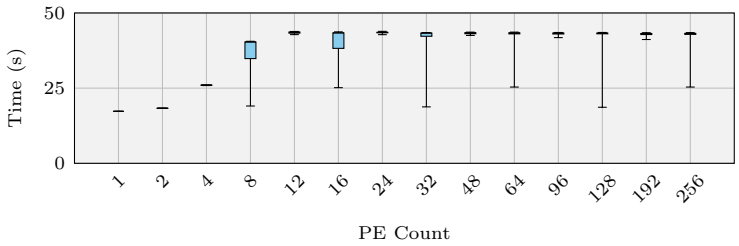
(b) Exchange



(c) Collectives



(d) Update Boundary



(e) Memory Management

Figure 4.5: Total Time Spent by Component Across All Ranks, Weak-Scaling — Minerva

Process Count	Nodes	Socket Mapping		
		Node-Fill	Load-Balance	Socket-Balance
8	1	(6) × [1]+ (2) × [1]	(6) × [1]+ (2) × [1]	(4) × [2]
16	2	(6) × [2]+ (4) × [1]	(6) × [2]+ (2) × [2]	(4) × [4]
32	3	(6) × [5]+ (2) × [1]	(6) × [2]+ (5) × [4]	(6) × [2]+ (5) × [4]
64	5	(6) × [10]+ (4) × [1]	(6) × [4]+ (5) × [8]	(6) × [4]+ (5) × [8]
128	11	(6) × [21]+ (2) × [1]	(6) × [18]+ (5) × [4]	(6) × [18]+ (5) × [4]
256	22	(6) × [42]+ (4) × [1]	(6) × [14]+ (5) × [8]	(6) × [14]+ (5) × [8]

Table 4.5: Socket Process Allocation. Format — (Socket Core Count)×[Number of Sockets]

is most evident in the compute kernels and communication costs for runs with power-of-two process counts, while those configurations that consists of solely fully-packed nodes (i.e. multiples of twelve) do not demonstrate this variability. This exposes some interesting mannerisms — the minimum compute time for 16/64/256 processes, where there exists one node with only four processes and the remainder with twelve, is roughly on par with that of the compute time for the four process experiment. Likewise, the minimum compute time for 32 and 128 processes, where there exists one node with only eight processes (six on one socket and two on another), is roughly on a par with the two/eight process experiments (where there exists only two processes on one socket). Conversely, the maximum compute time is relatively fixed when there exists at least one node in a configuration that has twelve processes allocated. There is a clear tie between the number of processes per socket/node and the compute time taken by a process, with better compute performance the fewer processes there are. Given the memory benchmarking results first introduced in Section 3.3.1, it is potentially the case that bottlenecks in the memory bandwidth are responsible for such behaviour.

This link between compute performance and number of cores per socket can be verified through the use of a load-balancing, rather than node-fill, approach to process allocation. Using the `-loadbalance` option of OpenMPI, the processes

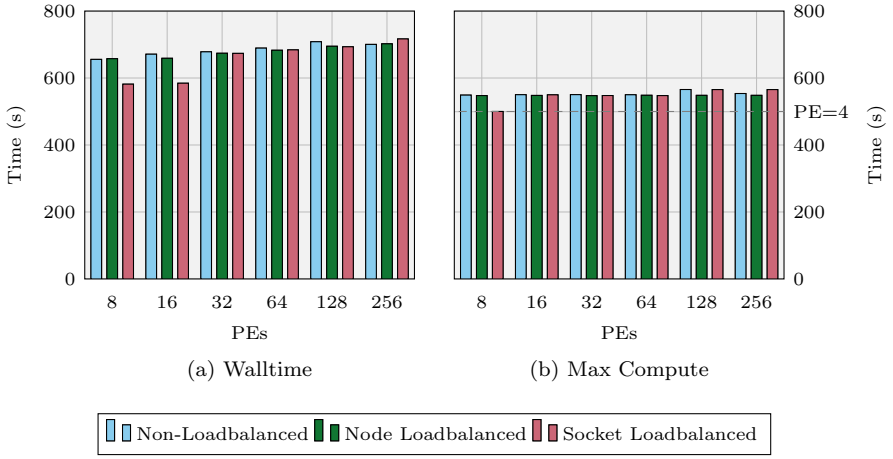


Figure 4.6: Hydra Socket/Node Load Balancing - Minerva

are spread as evenly across the nodes as possible. This leads to the process spread under "Load-Balance" found in Table 4.5.

The initial outcome of using load-balancing is underwhelming, presenting a few minor improvements in walltime as per Figure 4.6(a). However this approach only load-balances across nodes, not sockets. As such, even at 16 processes, there exists at least one full socket, as per the socket mapping described in Table 4.5. By applying the use of the OpenMPI option `--by-socket` in combination with load-balancing, the processes are evenly spread between sockets on a node, albeit in such a fashion that consecutive processes are on different sockets or nodes, potentially modifying the communication behaviour. With this more restricted approach for 8 and 16 processes, where only a maximum of 4 cores per socket is used, a much reduced total walltime is observed, as per Figure 4.6(a). Contrasting the compute performance against one another in Figure 4.6(b) it can be seen that the timings for this approach are now comparable to that of four processes in the initial investigation (as indicated by the line *PE=4*), where it is also the case that only four cores per socket are in use. This lends credence to the theory that the compute performance is tied to the number of cores per socket, likely as an outcome of the memory bandwidth

available to each being a dominant factor.

Beyond these process counts, it can be seen that at larger scales such load-balancing has a vastly reduced influence given a much more restricted environment for spreading the compute load with a fixed number of nodes. Regardless of the initial improvements at small scale, it has been established that the dominant factor for compute is the maximum number of cores per socket in the *worst* case — i.e. the existence of even a single socket with all cores in use means that a reduced number of cores on the remaining sockets has little impact. While it is possible to improve the spread across the sockets for 16 processes due to a number of spare cores in a two-node configuration, as the process count increases it becomes more difficult to provide an even spread where there is not at least one socket fully loaded without increasing the number of nodes, as is the case past 32 processes. Increasing the number of nodes available would improve this spread, but at the cost of a greater number of idle cores and a potentially greater communication cost due to more inter-node connections. Further to this, there is a potential increase in monetary cost associated with a higher node usage but same effective core count, an overall reduction in machine utilisation. Whether such an approach is worthwhile is explored later within this thesis.

Considering the now established variable compute behaviour in the initial set of results, the impact upon the communication is also non-negligible, mainifesting itself as idle time spent in synchronisation barriers that are attributed to collective/point-to-point blocking. With more significant compute ranges the idle time is likely to be higher on the better performing cores, thus there is a corresponding wider range of communication timings. For multiple-of-twelve process counts a reduced range of compute times can be seen, resulting in a similar reduction for the communication times, both point-to-point and collectives. In all cases, the minimum communication times (both collectives and point-to-point) follow a regular increasing trend as the process count increases.

The update boundary kernels are somewhat more irregular given that they are also tied to the existence of external boundaries, making their behaviour

P	Iterations					Walltime (s)			
	Total	Mlag				DawnDev (s)		Hera (s)	
		1	2	3	4	50^3	75^3	50^3	75^3
32	217	157	18	10	32	—		253.30	806.94
64	217	157	18	10	32	505.23	1665.80	291.58	901.91
128	217	157	18	10	32	525.15	1702.63	295.74	905.04
256	217	157	18	10	32	534.24	1718.68	310.06	974.75
512	217	157	18	10	32	528.76	1707.78	325.15	1002.52
1024	217	157	18	10	32	544.43	1729.02	337.54	1039.30
2048	217	157	18	10	32	584.97	1779.78	398.10	1172.02

Table 4.6: Hera/DawnDev, Hydra Weak-Scaling Walltimes [44]

predominantly tied to changes in the decomposition — e.g. 64 processes is the first case where there exists a process with no external boundaries at all, hence the sudden drop of the minimum cost to basically zero. The variation between these maximums and minimums is predominantly tied to the differences in the number of boundaries, though given the evidence of performance variations due to node-fill behaviour, it is possible that this is also a contributing factor.

To examine whether these properties are exhibited on multiple architectures, the outcome of a set of weak-scaled experiments for two different problem sizes, 50^3 and 75^3 on both the DawnDev and Hera machines in Table 4.6 are also presented.

As with Minerva, there is a slight upward trend in the time taken for a Hydra run that is tied to the increasing cost of communications associated with these parallel executions. As well as the associated increase in collective costs with the rising process count, the increase in inter-node connections contributes to a higher point-to-point communication cost. Given DawnDev and Hera's cores per node count of 4 and 16 respectively, for a given process count Hera will have a reduced number of inter-node connections compared to DawnDev, resulting in the trend of jumps in communication costs occuring at higher process counts for Hera than DawnDev.

Figure 4.7 reveals that, as with Minerva, the compute cost is relatively fixed while the higher walltimes result from an increase in the point-to-point and collective overheads. Again, using a mixture of maximum compute and minimum communication costs provides a reasonable approximation of the overall wall-

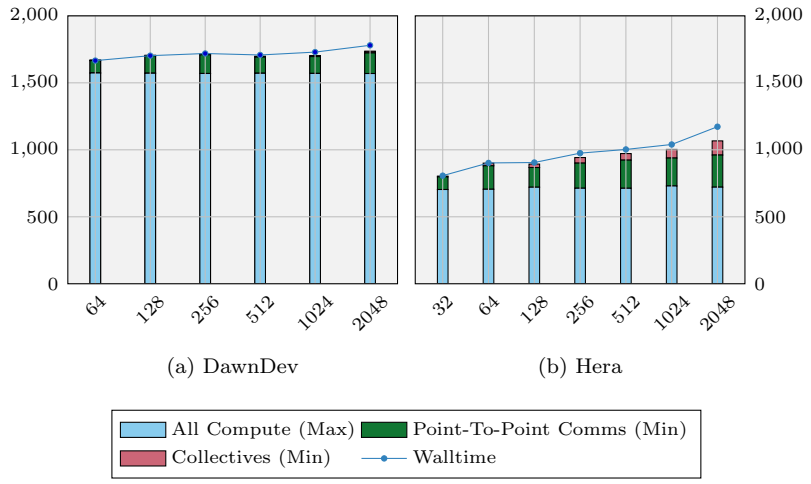


Figure 4.7: DawnDev/Hera 75^3 Weak-Scaling Hydra Walltime by Component

time, albeit with a slight shortfall for Hera at 2048 processes due to the nature of using such broad aggregate values to capture an environment that includes periods of process independent progression with fixed synchronisation points.

In this section it has been demonstrated that while the communication costs do increase as expected for a weak-scaled problem, the number of cores in use per socket is also an influencing factor, and that the computation costs are not necessarily fixed solely on the basis of the problem size. In addition, a pattern has been identified in the contribution of the various components that allows us to reasonably predict the walltime while simultaneously also validating that profiling efforts have captured the major performance "hot-spots".

4.5.3 Strong-Scaling

In a strong-scaled approach there will exist a change in both the compute and communication behaviour due to both (a) the reducing local problem/message size and (b) the number of messages increasing as the scale increases. While it is expected that the overall walltime should reduce at scale, the rate at which it does so is restricted by the potentially increasing communication costs (both

Problem	Total Iterations	Mlag Iterations 1	2	3	4	Walltime (s)	Standard Error
1	258	136	18	10	94	1941.77	0.63
2	258	136	18	10	94	1035.26	1.32
4	258	136	18	10	94	601.47	0.19
8	258	136	18	10	94	367.41	0.27
12	258	136	18	10	94	257.23	0.32
16	258	136	18	10	94	187.30	0.15
24	258	136	18	10	94	128.47	0.18
32	258	136	18	10	94	97.21	0.48
48	258	136	18	10	94	67.95	0.12
64	258	136	18	10	94	54.62	0.04
96	258	136	18	10	94	37.61	0.15
128	258	136	18	10	94	31.80	0.25
192	258	136	18	10	94	21.71	0.03
256	258	136	18	10	94	19.97	1.86

Table 4.7: Minerva, Hydra Strong-Scaling Walltimes (150^3, Node-Fill)

point-to-point and collectives) — if these costs increase beyond the rate at which the compute cost is reduced, little is offered by scaling any further. Thus capturing the behaviour of these communication stages is important to accurately predicting the behaviour at even greater scales. To explore this, a similar set of experiments to that of the weak-scaled node-fill approach are repeated (including those that are a multiple of twelve), but using a fixed global problem size of 150^3 in a strong-scaled setup.

In Figure 4.8 a breakdown by the various contributing components is provided for a node-fill process allocation approach. Since a maximum/minimum across all processes is used, rather than a single fixed process, the overall sum only approximates the total measured walltime. Once again, there is a shift in the performance hot-spots away from compute towards the communication costs, as might be expected. However, unlike with weak-scaled the compute cost is reducing, leading to a more rapid transition in comparison.

In the context of the absolute values, many of the expected trends of strong-scaling are observed (see Figure 4.9). As the number of processes is increased, the absolute overall compute cost decreases as expected. The variability of weak-scaling is not strongly observed, though this is likely due to the significantly reduced absolute values as the scale increases.

The point-to-point exchange costs, despite dominating more of the overall walltime, decrease in absolute value with the introduction of more processes

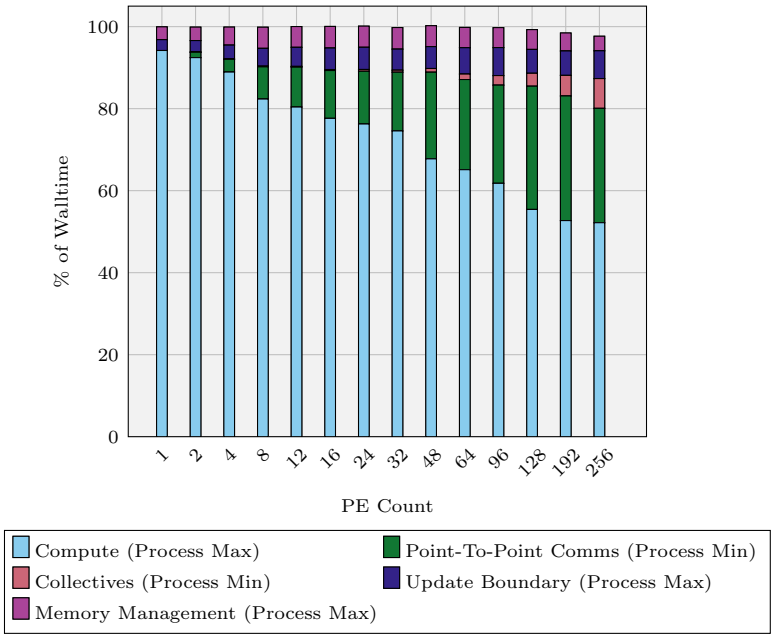
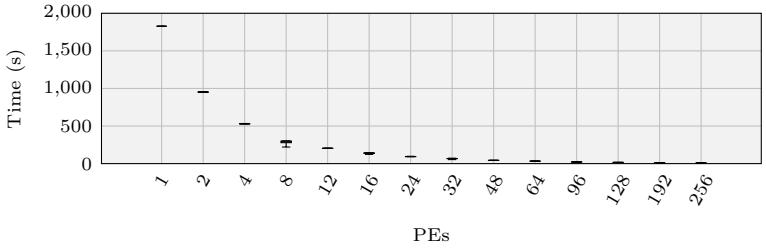


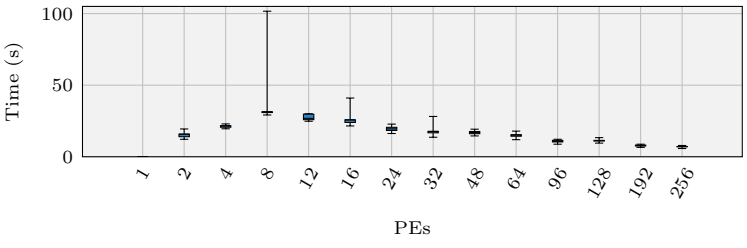
Figure 4.8: Max Walltime Percentage Breakdown

after an initial increase. This initial rise is partially down to the introduction of messages that were not previously present, as well as the transition to inter-node rather than intra-node communication. As the scale increases, the reduction in message size appears to have a greater impact than the introduction of additional messages, suggesting that the problem is initially bandwidth rather than latency bound. Given a worst case node sending messages for all six faces of its local grid, it is also expected that any increase from a rise in the number of messages per process would taper off as a peak is achieved. Any further increases in cost would be attributed not to an increase in messages per node, but rather synchronisation/network delays that might arise from a node being part of a chain of messages.

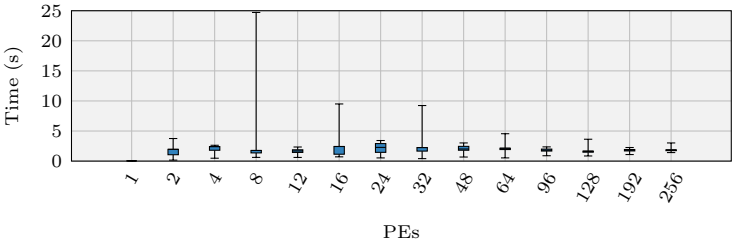
The collective costs once again exhibit a significant difference between the minimum and maximum values, but this reduces at scale, likely in part due to the reduction in the range of the compute values resulting in smaller synchronisation costs. Despite this, the minimum collectives costs remain on a par with the weak-scaled costs. This reinforces the idea that these particular costs are



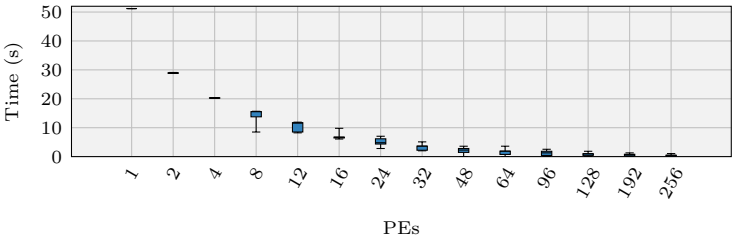
(a) Compute



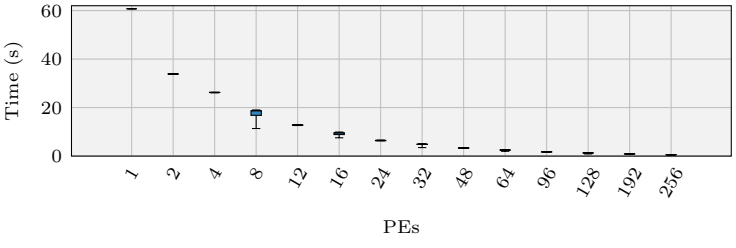
(b) Exchange



(c) Collectives



(d) Update Boundary



(e) Memory Management

Figure 4.9: Total Time Spent by Component Across All Ranks, Strong-Scaling — Minerva

primarily attributed to the number of processes, with their size/number being independent of the problem parameters.

The update boundary functions also act in a manner similar to that of weak-scaling in behaviour.

4.5.4 Dynamic Central Processing Unit (CPU) Scaling

The variation in compute performance shown in previous experiments implies that there is an additional behaviour influencing the rate at which work is completed. Serial investigations have revealed that the relationship between the maximum walltime for compute and the local amount of work is close to linear, yet in weak-scaled investigations the fixed amount of work per process has not demonstrated this property for select process counts; instead, it exhibits a steady increase in the compute time up to 16 processes. Since the primary experimental change in the weak-scaled investigations is the number of processes, it would appear that one of two factors is the cause — either the assumption that the amount of work is fixed is incorrect, or the performance per cell is changing.

It is important to eliminate one such possibility that could unduly influence the experiments while not being a part of Hydra itself — that of dynamic CPU scaling. If the clock speed changes as the process count is increased, it could potentially cause a change in the performance per cell. By using the `cpufreq-info` tool available from the SUSE Linux Enterprise OS it is possible to identify that Minerva is configured to have a dynamic scaling range between 1.60 GHz and 2.79 GHz with an "ondemand" policy, potentially influencing the CPU performance of runs. To identify whether this is the case, the current CPU clock speed is sampled approximately every 0.07 seconds during the course of an execution. The mean, standard deviation and variance of these clock speeds for a strong-scaled and weak-scaled experiment are presented in Table 4.8.

From this table it can be seen that there is a decline in mean clock speed performance, and a corresponding increase in variance, but only at 16 cores or greater. This decline is most appreciable for the strong-scaled experiments,

P	Strong-Scaled - 80^3			Weak-Scaled - 100^3		
	Mean Clock Speed (GHz)	Std. Dev (GHz)	Variance (GHz)	Mean Clock Speed (GHz)	Std. Dev (GHz)	Variance (GHz)
1	2.792	0.038	0.001	2.789	0.066	0.008
2	2.790	0.052	0.003	2.792	0.037	0.000
4	2.790	0.061	0.004	2.790	0.077	0.008
8	2.789	0.063	0.004	2.790	0.061	0.006
16	2.417	0.551	0.303	2.727	0.267	0.071
32	2.341	0.578	0.334	2.776	0.149	0.019

Table 4.8: Minerva, Hydra, Process 0 Clock Speeds

but weak-scaled experiments also exhibit a similar trait, albeit not to the same degree. Our prior breakdown of the experiments for weak-scaling demonstrate little variability in compute performance past 16 cores for a fixed workload suggesting that, during the compute at least, no scaling is occuring. In turn, strong-scaled experiments still exhibit a roughly linear scaling for compute on a par with that demonstrated in the serial experiments at the same process scale. The only suggestion of a poorer compute performance per cell occurs at 8 cores or fewer, for which these dynamic scaling experiments suggest there is little to no change in the scaled clock speed. As such, this posits that the variance in compute performance per cell is not attributable to dynamic clock scaling, but some other factor. The dynamic scaling decline in mean clock speed is likely caused by idle time during communication where, for strong-scaling at least, such stages become a more dominant part of the overall walltime as the scale increases, alongside a reduced overall walltime, potentially explaining its more significant decline over weak-scaling. This would also explain the dramatic change in mean and variance values between 8 and 16 processes, where the first inter-node communications are introduced (at 12 cores).

Given the behaviour of the different load balancing approaches in the weak-scaled investigations, where the compute performance was clearly tied to the number of cores per socket in use (rather than just the total cores per CPU), it is likely that the variance is caused by issues related to memory contention. The STREAM and CacheBench benchmarks of Section 3.3.1, where the bandwidth as the thread count increased was capped at approximately 30-35GB/s past four

threads, demonstrate such behaviours can occur. If the kernels within Hydra were to exhibit memory-bound rather than compute-bound properties, such bottlenecking behaviours would naturally manifest themselves as the system's performance would become restricted by the memory bandwidth, potentially accounting for the results presented here. Regardless, since such behaviours can be identified and categorised based on the number of maximum number of cores per socket in case, these details can be accounted for and incorporated into any modelling efforts, without knowing their exact cause.

4.6 Summary

This chapter has introduced Hydra, a benchmark code of interest to AWE. By establishing the critical path and code behaviours that dominate the performance of the application, a number of interesting areas have been identified for further investigation:

- A selection of input parameters have been shown to influence the behaviour of Hydra, including both simulation and machine chacteristics. Isolating their individual contributions is crucial to constructing a portable model;
- The full critical path has been instrumented, identifying all major contributors to the overall performance;
- Through serial, strong-scaled and weak-scaled experiments the differences in the performance of compute and communications have been captured, as well as exposing unexpected behaviours that warrent further investigation. Understanding these behaviours is crucial to predicting performance at scale;
- There exists an association between the performance per cell and the number of cores in use per socket, suggesting that memory-bound performance issues may exist. The impact of the cores per socket factor demonstrated is

significant enough that using extra nodes (and the additional communication costs this incurs) may be a more performant configuration, warrenting further investigation;

- At large enough problem sizes the serial investigations show a degradation in performance for a selection of functions, suggesting a threshold at which unoptimised behaviours occur.

Hydra has shown consistent patterns in its behaviour with regards to its input parameters. However, discrepancies appear to exist under select scenarios which could prove of interest to those seeking to improve its performance. To this end, a performance model may either reinforce the understanding of the code or reveal where deviations between expected and actual behaviour occur, leading both correction and performance optimisation efforts, a technique demonstrated in other works [89, 141]. To this end, the creation of an effective performance model is the first step towards such efforts.

CHAPTER 5

Modelling Hydra - A Performance Prediction Case Study

During a procurement process, the availability of suitable hardware for assessing the performance of machines often proves to be limited. Since the machine of interest often does not yet exist, smaller scale or similar hardware is instead provided for benchmarking purposes. However this by itself is only suitable for speculating or extrapolating performance on the final product. Performance models provide the means by which these metrics can be applied in an intelligent fashion to provide insight into the potential suitability of a machine. In this chapter, the process and construction of a performance model for the Hydra benchmark is introduced, including validation on multiple machines.

Specifically, the following goals are addressed:

- The first analytical model of Hydra is introduced, a benchmark of importance to the workflow of the Atomic Weapons Establishment (AWE);
- The model accuracy is compared across serial, weak-scaled and strong-scaled problem cases achieving accuracy within 15%;
- The model is validated for three distinct High Performance Computing (HPC) machines, DawnDev, Hera and Minerva; each possesses a distinct architecture, showing the transferability of the approach.

5.1 Input Parameters

For any analytical model, the selection of suitable model parameters is key to ensuring that the full range of behaviours are represented. From Chapter 4,

Parameter	Definition	Type
cps	Max cores per socket	Configuration
cpn	Max cores per node	Configuration
$iter(n)$	Number of iterations with n Mlagh inner loops	Derived
m	Max. number of Mlagh sub-iterations per cycle	Bench. input
Lx	Local X grid cell size	Derived
Ly	Local Y grid cell size	Derived
Lz	Local Z grid cell size	Derived
Nx	Global X grid cell size	Bench. input
Ny	Global Y grid cell size	Bench. input
Nz	Global Z grid cell size	Bench. input
P	Number of processing elements	Configuration
pe	Processing Element (PE) ID	Model input
Px	X Dimension PE count	Bench. input
Py	Y Dimension PE count	Bench. input
Pz	Z Dimension PE count	Bench. input

Table 5.1: Model Summary - Hydra Input Parameters

a range of benchmark characteristics were identified that influence the performance of Hydra, summarised in Table 5.1; it is these parameters that are used within this work for the model inputs of Hydra. However, it should be noted that not all model inputs are mutually exclusive.

Some parameters, such as P, Nx, Ny, Nz etc. are inputs to the Hydra benchmark itself, and as such are directly modifiable as part of any empirical experiment. These form the basis of any variation in our experimental setup.

Other parameters, such as $iter$ and $mlag(n)$ are indirectly determined by these benchmark inputs, resulting from the progressive state of the benchmark across the course of a run, the state naturally being determined by the model initalisation and termination conditions. A select few, such as Px, Py and Pz can be either input directly (assuming they are valid values), or left to the benchmark to determine a suitable value. The state-derived parameters $iter$ and $mlag(n)$ are populated within the model from historical data, since they cannot be predicted without executing the benchmark itself, but are independant of a machine enabling their reuse.

Finally configuration parameters are those which typically affect the performance of the benchmark but not the result – e.g. the number of cores/nodes the job is distributed across etc.

There also exists a small set of parameters, defined as Sx, Sy and Sz, that

influence the benchmark state, as previously mentioned in Section 4.2.4. These parameters represent the simulated spatial size per cell in the X, Y and Z dimensions, but their primary impact is to influence the values of $iter$ and $mlag(n)$. Since $iter$ and $mlag(n)$ are already captured within the model from historical data, Sx, Sy and Sz are omitted, with empirical investigations in this work typical keeping them at fixed values where possible.

5.2 Iteration Model

From the analysis of Hydra in Section 4.4, the critical path that constitutes the vast majority of the overall walltime taken in an execution is known, and the control flow is independent of the architecture (but not the performance). It is assumed that each of the processes is following a similar critical path, and that in the case of small deviations the longest running critical path is selected from amongst them. In addition, for the modelling work within this chapter it is assumed that the architecture in question is a homogeneous one – that is to say each of the individual PEs consists of the same hardware, with the same network interconnects and topology. In so doing, the performance of compute kernels or message communication times per unit of work or byte should be relatively similar, within deviations being down to configuration characteristics that can be captured by the model (such as workload balance or cores in use per socket. Adopting a top-down approach to modelling, the total walltime is the product of the time taken for an iteration with n inner mlag loops on a process pe, and the number of iterations with n *Mlagh* inner loops, summed for all values of n as in Equation 5.1.

$$T_{Walltime}(pe) = \sum_{n=1}^{m} T_{iter}(n, pe) * iter(n) \tag{5.1}$$

The overall walltime is dictated by the process with the highest walltime; however, given relatively frequent communication synchronisation points (both collective and point-to-point), the walltimes across the various processes are typi-

Equation	Function
$T_{walltime}(pe)$	Time – Walltime on process pe
$T_{iter}(n, pe)$	Time – Iteration with n $Mlagh$ inner loops on process pe
$T_{func}(pe)$	Time – Function $func$ on process pe
$T_{Mlagh}(n, pe)$	Time – $Mlagh$ function with n inner loops on process pe
$T_a(func, pe)$	Time – Memory allocation for function $func$ on process pe
$T_d(func, pe)$	Time – Memory deallocation for function $func$ on process pe
$T_c(k, pe)$	Time – Compute for kernel k on process pe
$T_p(s, pe)$	Time – Near-neighbour comms stage s on process pe
$T_{ag}(s, pe)$	Time – Global all-gather comms stage s on process pe
$T_{agv}(s, pe)$	Time – Global all-gather vector comms stage s on process pe
$T_{ub}(s, pe)$	Time – Boundary Update for stage s on process pe

Table 5.2: Model Summary - Iteration Model Overview

cally similar. These synchronisations points should have the least impact if the hardware and configuration is homogeneous across all nodes. A calculation can be made even if the hardware is not homogeneous; however the inputs/timings for the model would have to be obtained from the slowest node, since it will place constraints on the underlying performance due to synchronisation etc.

To provide the level of accuracy required to create a predictive model, the relationship between the machine and the behaviour of Hydra must be clearly established. It is neccessary to model the compute, point-to-point and collective components of the Hydra benchmark individually with regards to each function; doing so will reveal how the machine impacts upon these various components and provides the level of refinement neccessary to make the model portable across architectures. Substituting in function sub-models, the time for a single iteration is given as:

$$\begin{aligned} T_{iter}(n, pe) = & T_a(MDT, pe) + T_{MDT}(pe) + T_d(MDT, pe) + \\ & T_a(Mlagh, pe) + T_{Mlagh}(n, pe) + T_d(Mlagh, pe) + \\ & T_a(Madv, pe) + T_{Madv}(pe) + T_d(Madv, pe) + \\ & T_{Shortprint}(pe) \end{aligned} \tag{5.2}$$

The remainder of this model definition will elaborate further on how the various functions are broken down into their respective compute and communication components.

The following sub-sections present models for each function in a Hydra iteration. Within each model, a distinction is made between differing categories of behaviour, namely (a) grid kernel identifiers (compute), (b) update-boundary kernels (compute), (c) Message Passing Interface (MPI) point-to-point stages (network) and (d) MPI collective stages (network). These different categories are elaborated in further detail in Sections 5.4.1, 5.4.2, 5.5.2 and 5.6 respectively.

5.2.1 MDT

Compute Kernel identifiers: MDT_1 and MDT_2

Point-to-point communication stages: None

Update Boundary Stages: None

Global collective stages: MDT

Function calls: Leosdrv and Lartvis

$$T_{\mathrm{MDT}}(pe) = T_c(\mathrm{MDT}_1, pe) + T_c(\mathrm{MDT}_2, pe) + T_{\mathrm{Leosdrv}}(pe) + T_{\mathrm{Lartvis}}(pe) + T_{ag}(MDT, pe) \quad (5.3)$$

5.2.2 Mlagh

Compute Kernel identifiers: Mlagh_{Init}, Mlagh_1, Mlagh_2, Mlagh_3, Mlagh_4, Mlagh_5, Mlagh_6, Mlagh_7, UpdVel, Mdivu and MBFlux

Point-to-point communication stages: Mlagh_1 and Mlagh_2

Update Boundary Stages: $Mlagh_1$, $Mlagh_2$ and $Mlagh_3$

Global collective stages: $Mlagh_1$ and $Mlagh_2$

Function calls: Leosdrv, Lartvis and Mvolflx

$$T_{Mlagh}(n, pe) = T_c(\text{Mlagh}_{Init}, pe) + T_p(\text{Mlagh}_1, pe) + T_{ub}(Mlagh_1, pe) + \tag{5.4}$$

$$\begin{aligned}
&(T_c(\text{Mdivu}, pe) + T_c(\text{Mlagh}_1, pe) + T_{ag}(Mlagh_1, pe) + \\
&T_c(\text{Mlagh}_2, pe) + T_{\text{Leosdrv}}(pe) + T_c(\text{Mlagh}_3, pe) + \\
&T_p(\text{Mlagh}_2, pe) + T_{ub}(Mlagh_2, pe) + T_c(\text{UpdVel}, pe) + \\
&T_{ub}(Mlagh_3, pe) + T_c(\text{Mlagh}_4, pe) + T_c(\text{MBFlux}, pe) + \\
&T_c(\text{Mdivu}, pe) + T_c(\text{Mlagh}_5, pe) + T_{ag}(Mlagh_2, pe) + \\
&T_c(\text{Mlagh}_6, pe)) \times n + \\
&(T_{\text{Leosdrv}}(pe)) + T_{\text{Lartvis}}(pe)) \times (n-1) + \\
&T_c(\text{Mlagh}_7, pe) + T_{\text{Mvolflx}}(pe)
\end{aligned}$$

5.2.3 Madv

Compute Kernel identifiers: Madv_1, Madv_2 and Purge

Point-to-point communication stages: Madv

Update Boundary Stages: $Madv$

Global collective stages: $Madv$

Function calls: Madvx, Madvy and Madvz

$$\begin{aligned}
T_{Madv}(pe) =& T_c(\text{Madv}_1, pe) + T_c(\text{Purge}, pe) + \\
&T_p(\text{Madv}, pe) + T_{ub}(Madv, pe) + T_{\text{Madvx}}(pe) + \\
&T_p(\text{Madv}, pe) + T_{ub}(Madv, pe) + T_{\text{Madvy}}(pe) + \\
&T_p(\text{Madv}, pe) + T_{ub}(Madv, pe) + T_{\text{Madvz}}(pe) + \\
&T_c(\text{Madv}_2, pe)
\end{aligned} \tag{5.5}$$

5.2.4 Madvx

Compute Kernel identifiers: Madvx_1 and Madvx_2

Point-to-point communication stages: None

Update Boundary Stages: None

Global collective stages:

Function calls: Madvmx

$$T_{\text{Madvx}}(pe) = T_c(\text{Madvx}_1, pe) + T_c(\text{Madvx}_2, pe) + T_{\text{Madvmx}}(pe) \tag{5.6}$$

5.2.5 Madvy

Compute Kernel identifiers: Madvy_1 and Madvy_2

Point-to-point communication stages: None

Update Boundary Stages: None

Global collective stages: None

Function calls: Madvmy

$$T_{\text{Madvy}}(pe) = T_c(\text{Madvy}_1, pe) + T_c(\text{Madvy}_2, pe) + T_{\text{Madvmy}}(pe) \tag{5.7}$$

5.2.6 Madvz

Compute Kernel identifiers: Madvz_1 and Madvz_2

Point-to-point communication stages: None

Update Boundary Stages: None

Global collective stages: None

Function calls: Madvmz

$$T_{\text{Madvz}}(pe) = T_c(\text{Madvz}_1, pe) + T_c(\text{Madvz}_2, pe) + T_{\text{Madvmz}}(pe) \tag{5.8}$$

5.2.7 Madvmx

Compute Kernel identifiers: Madvmx_1

Point-to-point communication stages: Madvm

Update Boundary Stages: $Madvm$

Global collective stages: None

Function calls: None

$$T_{\text{Madvmx}}(pe) = T_p(\text{Madvm}, pe) + T_{ub}(Madvm, pe) + T_c(\text{Madvmx}_1, pe) \quad (5.9)$$

5.2.8 Madvmy

Compute Kernel identifiers: Madvmy_1

Point-to-point communication stages: Madvm

Update Boundary Stages: $Madvm$

Global collective stages: None

Function calls: None

$$T_{Madvmy}(pe) = T_p(\text{Madvm}, pe) + T_{ub}(Madvm, pe) + T_c(\text{Madvmy}_1, pe) \quad (5.10)$$

5.2.9 Madvmz

Compute Kernel identifiers: Madvmz_1

Point-to-point communication stages: Madvm

Update Boundary Stages: $Madvm$

Global collective stages: None

Function calls: None

$$T_{Madvmz}(pe) = T_p(\text{Madvm}, pe) + T_{ub}(Madvm, pe) + T_c(\text{Madvmz}_1) \quad (5.11)$$

5.2.10 ShortPrint

Compute Kernel identifiers: ShortPrint_1

Point-to-point communication stages: None

Update Boundary Stages: None

Global collective stages: $ShortPrint_{Ag1}$ and $ShortPrint_{Agv1}$

Function calls: None

$$T_{\text{ShortPrint}}(pe) = T_c(\text{ShortPrint}_1, pe) + T_{ag}(ShortPrint_{Ag1}, pe) + T_{agv}(ShortPrint_{Agv1}, pe) \quad (5.12)$$

5.2.11 Lartvis

Compute Kernel identifiers: Lartvis_1

Point-to-point communication stages: Lartvis

Update Boundary Stages: $Lartvis$

Global collective stages: None

Function calls: None

$$T_{\text{Lartvis}}(pe) = T_p(\text{Lartvis}, pe) + T_{ub}(Lartvis, pe) + T_c(\text{Lartvis}_1, pe) \quad (5.13)$$

5.2.12 Leosdrv

Compute Kernel identifiers: Leosdrv_1

Point-to-point communication stages: None

Global collective stages: None

Function calls: None

$$T_{\text{Leosdrv}}(pe) = T_c(\text{Leosdrv}_1, pe) \quad (5.14)$$

From prior domain knowledge of the Hydra benchmark, it is known that the functional behaviour within the various "categories" of sub-model is similar, enabling the production of general models whose output can be modified with a small selection of inputs. This, in turn, permits the establishment of a relationship between the machine hardware and the benchmark performance. The following sections introduce not only the work decomposition of Hydra (defining a number of these influencing input parameters), but how the behaviour of the compute and communication stages translates into their respective models.

5.3 Process and Cell Layout

The processes of Hydra, as established earlier in this work (Section 4.3.1) are laid out in a 3D cartesian grid in a $P_x \times P_y \times P_z$ arrangement. An $N_x \times N_y \times N_z$ global grid is then decomposed across these processes in a relatively even manner. This layout is important due to the association between process id (pe) and work allocation; specifically the size of the local grid, the identification of neighbouring processes in the 3D cartesian grid, whether communications will be intra-node or inter-node and whether a process has internal or external boundaries (i.e. whether it shares a grid decomposition boundary with another process). Capturing these behaviours is important to construct an accurate depiction of how a workload is distributed and processed for both compute and communication models. To do this, it is neccessary to determine a number of properties — the position of a process within the cartesian grid, the size of the local grid in all three dimensions and whether each of the six faces is an internal or external boundary. In all models the process id and process coordinates are zero-indexed. The remainder of this section introduces the models for the default behaviour used in our experimental approach, that of a node-fill process allocation with a relatively even grid decomposition (though in exploratory scenarios, alternate models can be substituted). These models are summarised in Table 5.3.

Function	Description	Ref.
$pc(dm, pe)$	Returns the position of process pe in the process grid layout for dimension dm.	5.17
$lc(dm, pe)$	Returns the number of cells in the local grid for dimension dm on process pe.	5.18
$ib(dm, dr, pe)$	Determine whether an internal boundary exists in dimension dm on the face in direction dr for process pe.	5.20
$eb(dm, dr, pe)$	Determine whether an external boundary exists in dimension dm on the face in direction dr for process pe.	5.21
$pn(pe, cpn)$	Returns the node id of process pe where cpn is the number of cores per node (node-fill).	5.22
$pcs(P, pe, cps)$	Gives the number of active cores for the socket containing process pe, given cps cores per socket and P total processes.	5.23
$cn(cdm, pe, dm, dr)$	If dimension coordinate cdm does not equal dimension dm, return the current process position in dm for process pe else return the process position in dimension cdm for a neighbouring process (if such a neighbour exists).	5.23
$pid(cx, cy, cz)$	Given the cartesian coordinates cx, cy and cz, return the process id.	5.24
$nb(dm, dr, pe)$	Find the process id of the process neighbouring pe in dimension dm and direction dr.	5.25

Table 5.3: Model Summary - Process and Cell Layout

The coordinate (c_x, c_y, c_z) is defined to be the 3D cartesian position of a process pe within the process layout grid, with the sets S_{dm} and S_{dr} describing the potential configuration of any face's dimension and direction (Low and High being the opposite faces) respectively.

$$S_{dm} = \{X, Y, Z\} \tag{5.15}$$

$$S_{dr} = \{Low, High\} \tag{5.16}$$

The relationship between these coordinates and a process id is described in Equation 5.17, where pe is the processing element id and dm is the dimension of the coordinate (either X, Y or Z). The coordinates c_x, c_y and c_z are equivalent to $pc(X, pe)$, $pc(Y, pe)$ and $pc(Z, pe)$ respectively.

$$pc(dm, pe) = \begin{cases} pe \bmod Px & \text{if } dm = X \\ (\lfloor pe/P_x \rfloor) \bmod Py & \text{if } dm = Y \\ (\lfloor pe/(P_x \times P_y) \rfloor \bmod Pz & \text{if } dm = Z \end{cases} \tag{5.17}$$

Since the performance is typically tied to the number of cells a process must operate upon, understanding how the data for computation is distributed is neccessary to identify the maximum number of cells on an individual process. Equation 5.18 captures the allocation of cells from the global grid across a set of decomposed processes, assuming a zero-index process id.

$$lc(dm, pe) = \begin{cases} \lfloor N_x/P_x \rfloor + \min(1, \lfloor (N_x \mod P_x)/(pc(X, pe) + 1) \rfloor) & \text{if } dm = X \\ \lfloor N_y/P_y \rfloor + \min(1, \lfloor (N_y \mod P_y)/(pc(Y, pe) + 1) \rfloor) & \text{if } dm = Y \\ \lfloor N_z/P_z \rfloor + \min(1, \lfloor (N_z \mod P_z)/(pc(Z, pe) + 1) \rfloor) & \text{if } dm = Z \end{cases} \tag{5.18}$$

In the event that, for a dimension dm, N_{dm}/P_{dm} is a perfect division then there is an equal spread of cells in that dimension. If this is not the case then there is a slightly uneven distribution of cells. To minimise the imbalance this could cause, the remainder of cells, r, are spread across the first r processes encountered in the decomposition of dimension dr (i.e. $pc(dr, pe) < r$), increasing their local cell count by one in this dimension.

Of note is the distinction between the management of processes within Hydra and that of the MPI implementation. Hydra manages processes from a data perspective – it controls the allocation of data to each process, and which processes possess data upon which another process is dependant; neighbouring processes in the process cartesian grid also contain neighbouring grid data. However, the manner in which these processes are mapped to hardware is handled by the MPI implementation and configuration – neither step influences the other without manual configuration, and there is no guarantee that two processes that contain

neighbouring data are physically neighbours – i.e. on the same socket/node.

The consequence of this is that it must be estabished whether any communications between neighbouring processes are intra-node or inter-node, as well as whether any such communications exist. This is dependant upon two factors – which processes are communication neighbours and which core/node these neighbour processes have been bound to. Equations 5.20 and 5.21 model the existence of an internal and external boundary for a face in dimension dm, direction dr on a process pe.

$$pd(dm) = \begin{cases} Px & \text{if } dm = X \\ Py & \text{if } dm = Y \\ Pz & \text{if } dm = Z \end{cases} \tag{5.19}$$

$$ib(dm, dr, pe) = \begin{cases} 0 & \text{if } (pc(dm, pe) = 0) \text{ and } (dr = Low) \\ 0 & \text{if } (pc(dm, pe) = pd(dm) - 1) \text{ and } (dr = High) \\ 1 & \text{else} \end{cases} \tag{5.20}$$

$$eb(dm, dr, pe) = (ib(dm, dr, pe) + 1) \bmod 2 \tag{5.21}$$

An internal boundary, signifying a dividing line in the cell grid where cells on either side are assigned to differing processes, can result in either an intra-node or inter-node communication, dependent on the relative physical mapping of the neighbouring process. External boundaries, in contrast, signify where the face of a local grid also aligns with the edge of the global grid – i.e. there are no neighbouring cells and therefore no neighbouring processes past this boundary. Both are determined by the process-to-hardware mapping policy of MPI. In this work a node-fill allocation is typically applied, but alternate process mapping policies can be substituted into the model to explore alternate scenarios of interest (e.g. a node/socket round-robin policy). Equation 5.22

models this node-fill policy, returning the node position for a process pe with a maximum of cpn cores per node.

$$pn(pe, cpn) = \lfloor pe/cpn \rfloor \tag{5.22}$$

In addition, since it was previously identified in Chapter 4 that the number of active cores on a socket influences the compute/memory performance, Equation 5.23 applies the node-fill approach (effectively also a socket-fill approach) to derive the number of active cores on the same socket as process pe, for a system with cps cores per socket.

$$pcs(P, pe, cps) = \begin{cases} P & \text{if } P <= cps \\ (P \bmod cps) \times \lfloor pe/(\lfloor P/cps \rfloor \times cps) \rfloor) + & \\ (cps \times (1 - (\lfloor pe/(\lfloor P/cps \rfloor \times cps) \rfloor))) & \text{if } P > cps \end{cases}$$

From these hardware mapping models, knowledge of which PEs are neighbours combined with their node allocation can be modelled (Equations 5.23 – 5.30) to show whether the MPI communications are intra-node or inter-node.

$$cn(cdm, pe, dm, dr) = \begin{cases} pc(cdm, pe) & \text{if } cdm \neq dm \\ pc(cdm, pe) - ib(dm, Low, pe) & \text{else if } dr == Low \\ pc(cdm, pe) + ib(dm, High, pe) & \text{else if } dr == High \end{cases} \tag{5.23}$$

$$pid(c_x, c_y, c_z) = c_x + (c_y * P_x) + (c_z * P_x * P_y) \tag{5.24}$$

Equation 5.23 acts as a neighbour coordinate filter. For any immediate neighbour, two of the three coordinates between process pe and a neighbour must remain the same. Passing through the three dimensions to this function will

retrieve the coordinates of the potential neighbour by returning two of the coordinates unchanged and the third modified. In this equation, dm is the dimension in which the target process is a neighbour to the process pe (since it can only be an immediate neighbour in one dimension). dr represents the direction the neighbour lies in on the plane of dimension dm — i.e. Low or High being the left-hand face or the right-hand face. The equation returns the cartesian position in the overall process layout for the target neighbour in the dimension cdm, i.e. X for the coordinate c_x, Y for the coordinate c_y and Z for the coordinate c_z. In the event that there is no internal boundary, the coordinates for all three dimensions all match that of process pe (signifying the non-existence of a neighbour).

$$nb(dm, dr, pe) = pid(cn(X, pe, dm, dr), cn(Y, pe, dm, dr), cn(Z, pe, dm, dr)) \tag{5.25}$$

Equation 5.25 uses Equations 5.23 and 5.24 to retrieve the process id of a neighbour by (1) finding the coordinate position of pe, (2) determining the coordinates of the neighbour in dimension dm and direction dr and (3) constructing the process id of the neighbour using the neighbour coordinates. In the event that a neighbour does not exist then the process id pe is returned, and removed from the set of potential neighbours as per Equation 5.26.

$$S_{neigh}(pe) = \{nb(dm, dr, pe) : dm \in S_{dim}, dr \in S_{dir}\} \setminus \{pe\} \tag{5.26}$$

$$S_{neigh,X}(pe) = \{nb(X, dr, pe) : dr \in S_{dir}\} \setminus \{pe\} \tag{5.27}$$

$$S_{neigh,Y}(pe) = \{nb(Y, dr, pe) : dr \in S_{dir}\} \setminus \{pe\} \tag{5.28}$$

$$S_{neigh,Z}(pe) = \{nb(Z, dr, pe) : dr \in S_{dir}\} \setminus \{pe\} \tag{5.29}$$

$$S_{neigh}(pe) = S_{\text{neigh},X}(pe) \cup S_{neigh,Y}(pe) \cup S_{neigh,Z}(pe) \tag{5.30}$$

5.4 Compute - Work Per Unit (Wg)

To derive a model for any compute portion of the benchmark, it is neccesary to construct some means of predicting a kernel's runtime. To do so, this work adopts the use of Wg values [124], values that represent the amount of time to process a basic "unit" of work. By identifying a relationship between a fixed set of known input parameters and the amount of basic work units required for a particular kernel, it is then possible to obtain a runtime prediction for a compute block within the benchmark.

The computation of suitable Wg values is achieved primarily via the use of linear regression. Previous investigations in this work have demonstrated the apparently linear relationship between the number of cells and the time spent in a compute kernel. However, a simple least-squares regression is not sufficient to provide accurate predictions in all scenarios. Select kernels such as Madvmz_1 have previously demonstrated unexpected, non-linear behaviour aross the full range of cell counts. In addition, the use of least-squares regression across the full range of values can skew accuracy towards those results with high cell counts. Small deviations in absolute value have little impact upon the accuracy of these results, but can have a significant effect upon the accuracy of smaller absolute values such as those in the lower cell count range.

One such example of this is the Lartvis_1 kernel in Figures 5.1(a) through 5.1(d). In Figure 5.1(a), it appears as a simple linear regression. However, splitting it into distinct sub-regions, as in Figures 5.1(b) through 5.1(d) it can be seen that the least-squares regression for the full data-set is off by a significant margin for the smaller cell counts. To address this, a piece-wise/segmented linear regression approach is adopted instead [115], where the overall data-set is split into these segmented regions, each of which has its own distinct least squares linear regression. To determine the region, a subset of the data is used to plot a linear regression, before the next piece of data in the sorted data-set is introduced. If the percentage error deviates outside of a pre-specified range

(for Hydra a value of 30% was used), then the prior linear regression is fixed, marking the end of a region. A new region is then begun with the next distinct set of data and its own distinct linear regression. As is evident in these Figures, a more accurate result is obtained.

This maintains the predictive capacity of the linear relationships, but also addresses both minute shifts in value across the large range of the independant variable, as well as capturing sudden behaviour changes in results such as for the Madvmz_1. The danger of such an approach is that if the relationship is not truely linear then the final result is effectively a linear-interpolation which has no predictive capacity for those values that exists between previously measured points. However, in the case of Hydra this seems unlikely, with the the error margin of 30% producing typically around 3-4 distinct regions. The piecewise approach offers a degree of accuracy sufficient for predictive purposes.

In Hydra, the predominant influencing parameter for a compute kernel is the number of local cells to be processed – the portion of the global grid that is allocated to a MPI process after decomposition. These kernels typically fall into one of two categories – *grid* kernels or *boundary* kernels.

5.4.1 Grid Kernels

Grid kernels are usually 3D nested loops that operate across the entirity of a local grid, repeating a fixed set of operations for each cell, lending themselves towards the Wg model due to the similiarity in compute time per cell. The size of the grid that is iterated across can vary slightly, depending upon the nature of the quantities/data arrays they are processing. Section 4.2.1 earlier in this work made a distinction between different types of meshes such as cell-centered or nodal, where the data is treated as being at the centre of a cell or at any of the cell's vertices. In addition the nature of parallel decomposition is such that transferred data from neighbouring processes, stored in buffer cells past the range of the local grid, may also require processing as part of a kernel computation, extending the range of the nested loops further. Depending upon the

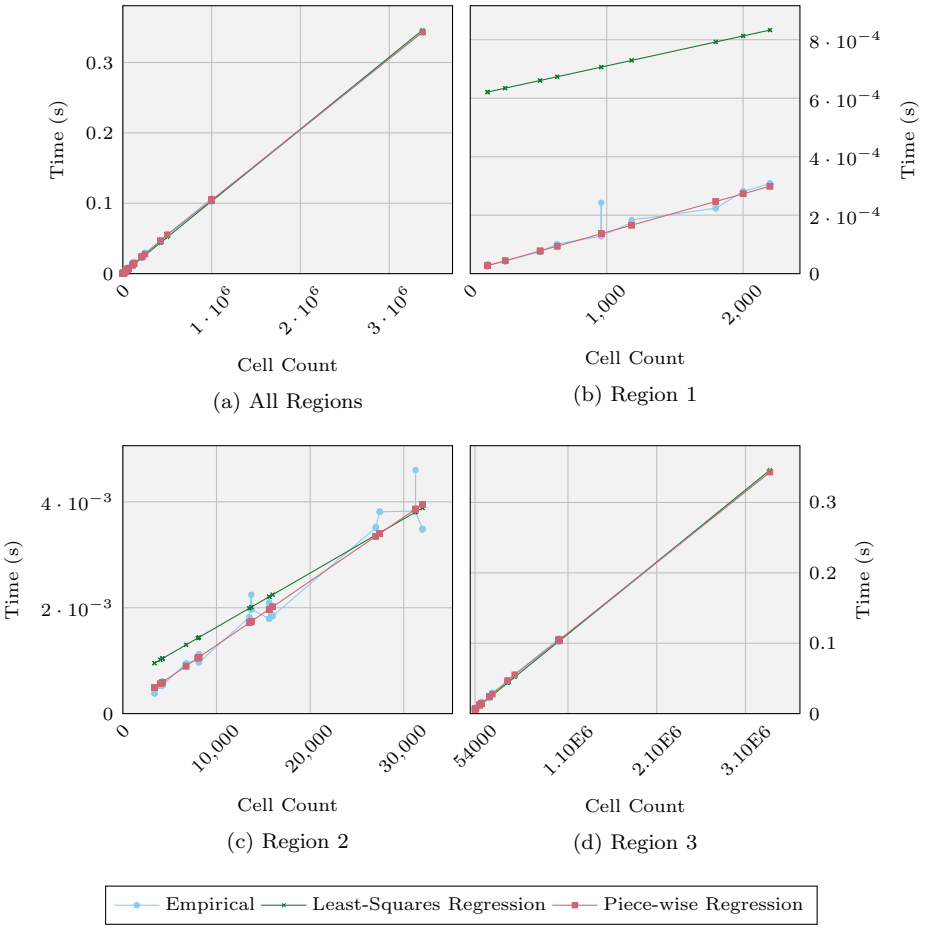


Figure 5.1: Lartvis$_1$ Kernel Timings – Minerva, 6 Cores Per Socket

kernel, while the dominating input characteristics are the values of Lx, Ly and Lz, the full range may extend by a few extra cells in both the lower and upper dimensions, resulting in an increased number of "work-units". While stencil kernels can touch data values from cells other than their own, they typically still cycle through the full grid of cells and are thus treated in the same manner as grid kernels. Equation 5.31 offers a simple summary of the computation for the compute time of a grid kernel.

$$T_c(func, pe) = Wg_{func} * lc(X, pe) * lc(Y, pe) * lc(Z, pe) \tag{5.31}$$

5.4.2 Boundary Kernels

Boundary kernels, unlike grid kernels, operate on only a subset of the overall local grid. They predominantly focus upon the outer faces, i.e. the boundaries of the grid, iterating over the outermost cells up to a limited cell depth. Some boundary kernels may operate solely upon external boundaries (where the local grid boundary is also the outmost boundary of the global grid), or internal boundaries (where there exist neighbouring processes and halo data for received MPI communications). A select number of kernels process both external and internal boundaries equivalently.

$$T_c(func, pe) = \begin{cases} Wg_{func} * lc(Y, pe) * lc(Z, pe) & \text{if X Boundary} \\ Wg_{func} * lc(X, pe) * lc(Z, pe) & \text{if Y Boundary} \\ Wg_{func} * lc(X, pe) * lc(Y, pe) & \text{if Z Boundary} \end{cases} \tag{5.32}$$

5.5 Point-To-Point Communication

5.5.1 Message Sizes

As previously identifed in Section 4.3.2, each point-to-point exchange phase consists of up to 12 messages per process, comprised of two different data types, per each of six faces (for near-neighbours in all directions). Each message in turn is an amalgamation of the data belonging to multiple quantities, packed in a setup phase and unpacked upon reception by a reverse of the process. This substitutes multiple small messages for one large message to minimise the impact of latency involved in sending many messages.

From Section 4.2.4 it is known that quantities with different properties (cell-centered, nodal or faced) possess different array dimensions, resulting in small variations in the amount of data to be communicated for a particular quantity. In addition, each communication stage has its own particular characteristics. It may pack one or more quantities into the buffer, and has a fixed halo size

Parameter	Description
fq_{cc}	Number of Cell-Centered Quantities
fq_{n}	Number of Nodal Quantities
fq_{fx}	Number of X-Faced Quantities
fq_{fy}	Number of Y-Faced Quantities
fq_{fz}	Number of Z-Faced Quantities
h	Width of the halo area
$bytes_{int}$	Size of an integer datatype in bytes
$bytes_{real}$	Size of a real datatype in bytes

Table 5.4: Model Message Size Parameters

– e.g. a halo size of one indicates each face communicated must be one cell deep, a halo of two is two cells deep etc. As per the exchange pattern in Section 4.3.2, the halo size will also influence the number of additional ghost cells to be communicated in the Y and Z steps. The number of quantities and the halo size directly influence the message size, and thus, along with the local cell count parameters Lx, Ly and Lz, form the crux of the models for determining message size introduced in the remainder of this sub-section, summarised in Table 5.5. There exists a relationship between the number of cells in a given dimension and which step is currently being conducted in the exchange process – i.e. the X, Y or Z dimension communication. For example, when communicating a face in the Y dimension, the cell-count in the X dimension is increased to include halo cells received from the exchange step in the X dimension (see Section 4.3.2). Similar behaviours exist for the X and Y dimension cells when communicating the Z step. This behaviour can be summarised as $X < Y < Z$, and is captured as a utility function in Equation 5.33.

$$dirrel(d1, d2) = \begin{cases} 1 & \text{if } (d2 = Y) \text{ and } (d1 = X) \\ 1 & \text{else if } (d2 = Z) \text{ and } (d1 = X) \\ 1 & \text{else if } (d2 = Z) \text{ and } (d1 = Y) \\ 0 & \text{else} \end{cases} \tag{5.33}$$

As well as incorporating these received halo cells, different classes of quantities can have similar but different numbers of cells in the three dimensions. Thus the size of a message, mostly dependant upon the overall size of a single face of the

Function	Description	Equation
$dirrel(d1, d2)$	Determine whether $d1$ is a dimension of an order higher than $d2$	5.33
$dp_{cc}(d, cd, h, pe)$	Number of data-points in dimension d for a message communicated in dimension cd with halo width of h from process pe single cell-centered quantity	5.34
$dp_n(d, cd, h, pe)$	Number of data-points in dimension d for a message communicated in dimension cd with halo width of h from process pe single nodal quantity.	5.35
$dp_f(d, cd, fd, h, pe)$	Number of data-points in dimension d for a message communicated in dimension cd with halo width of h from process pe single faced quantity with face direction fd.	5.36
$m_{cc}(cd, h, pe)$	Number of data points communicated for a single single cell-centered quantity communicating in dimension cd with a halo size of h from process pe	5.37
$m_n(cd, h, pe)$	Number of data points communicated for a single single nodal quantity communicating in dimension cd with a halo size of h from process pe	5.38
$m_f(cd, fd, h, pe)$	Number of data points communicated for a single single faced quantity (with face dimension fd), communicating in dimension cd with a halo size of h from process pe	5.39
$msgC(fq_{cc}, fq_n, fq_{fx}, fq_{fy}, fq_{fz}, cd, pe, h)$	The total number of data elements in a message that consists of fq_{cc} cell-centered quantities, fq_n nodal quantities, fq_{fx} faced quantities (X-faced), fq_{fy} faced quantities (Y-faced), and fq_{fz} faced quantities (Z-faced), in the direction cd with a halo size h from process pe	5.40

Table 5.5: Message Size Models – Summary

local cell grid, has an additional constraint on the class of quantities involved in a phase. The computation of a message size is thus broken into the following steps:

1. Identify all related quantities associated with the current point-to-point communication stage;

2. For a single data type (e.g. integer or double), compute the size of a face for the direction, dr and communication direction cdm for all quantities, based on which class each quantity is;

3. Sum the size of these faces to obtain the total number of cells to be placed

into a single buffer, and multiply by the byte size of the data type for the total message size in bytes;

4. Repeat steps 2 and 3 for each data type, since these are communciated separately from one another.

The series of dp Equations 5.34 to 5.36 return the number of "datapoints" (aka. cells) for cell-centered (cc), nodal (n) and faced (f) quantities respectively, for a particular dimension dm when communicating in dimension cdm with a fixed halo size of h for a process pe.

$$dp_{cc}(dm, cdm, h, pe) = \begin{cases} h & \text{if } dm = cdm \\ lc(dm, pe) + (h \times ib(dm, Low, pe) & \\ \times dirrel(dm, cdm) & \\ +(h \times ib(dm, High, pe) & \\ \times dirrel(dm, cdm)) & \text{else if } dm \neq cdm \end{cases} \tag{5.34}$$

$$dp_{n}(dm, cdm, h, pe) = \begin{cases} h & \text{if } dm = cdm \\ dp_{cc}(dm, cdm, h, pe) + 1 & \text{else if } dm \neq cdm \end{cases} \tag{5.35}$$

$$dp_{f}(dm, cdm, fd, h, pe) = \begin{cases} h & \text{if } dm = cdm \\ dp_{n}(dm, cdm, h, pe) & \text{else if } dm = fd \\ dp_{cc}(dm, cdm, h, pe) & \text{else if } dm \neq fd \end{cases} \tag{5.36}$$

The addition of further halo cells from prior communications in Equation 5.34 is dependent on (a) the existence of an internal boundary and (b) which step of the exchange has been reached, as per the description of the exchange process

Phase	**Message Type**	fq_{cc}	fq_n	$fq_{f,x}$	$fq_{f,y}$	$fq_{f,z}$	h
Lartvis	Real	1	0	0	0	0	1
Lartvis	Integer	0	0	0	0	0	1
Mlagh_1	Real	1	6	0	0	0	1
Mlagh_1	Integer	1	0	0	0	0	1
Mlagh_2	Real	3	0	0	0	0	1
Mlagh_2	Integer	1	0	0	0	0	1
Madv	Real	5	0	0	0	0	2
Madv	Integer	1	0	0	0	0	2
Madvm	Real	4	6	1	1	1	3
Madvm	Integer	0	0	0	0	0	3

Table 5.6: Pure Phase Type Quantity Frequency

in Section 4.3.2 (i.e. whether prior communications have occured). From these equations, the number of datapoints overall in the dimension of the communication is equivalent to the $facesize \times halosize$. The face size is itself directly tied to the number of cells in the local grid in dimensions dm – i.e. $lc(dm, pe)$, while the *halosize* is equivalent to a fixed input, determined by the communication stage, and is returned by the equation when dm is equivalent to cdm. This leads to Equations 5.37 through 5.39, capturing the *total* number of datapoints communicated for a single quantity of a particular class in the dimension cdm, capturing step 2 of the message size process.

$$m_{cc}(cdm, h, pe) = \prod_{dm \in S_{dim}} dp_{cc}(dm, cdm, h, pe) \tag{5.37}$$

$$m_n(cdm, h, pe) = \prod_{dm \in S_{dim}} dp_n(dm, cdm, h, pe) \tag{5.38}$$

$$m_f(cdm, fd, h, pe) = \prod_{dm \in S_{dim}} dp_f(dm, cdm, fd, h, pe) \tag{5.39}$$

Taken one step further, knowing the message size for a quantity of any particular class, the total message size can be derived from the frequency of various quantities in a communication for a particular datatype. This leads to Equation 5.40 which, when used in conjunction with domain knowledge of Hydra as presented in Table 5.6, can derive the total number of data points in a single MPI message using the appropriate frequencies. Finally, knowing the bytesize of a

datatype, Equation 5.41 produces the final message size in bytes, a value that can be used as an input to a network performance model. From this, a point-to-point communication stage can begin to be modelled with the knowledge of how much data is being transferred.

$$\begin{aligned} msgC(fq_{cc}, fq_n, fq_{fx}, fq_{fy}, fq_{fz}, cdm, pe, h) = &(m_{cc}(cdm, h, pe) \times fq_{cc}) \\ &+ (m_n(cdm, h, pe) \times fq_n) \\ &+ (m_f(cdm, X, h, pe) \times fq_{fx}) \\ &+ (m_f(cdm, Y, h, pe) \times fq_{fy}) \\ &+ (m_f(cdm, Z, h, pe) \times fq_{fz}) \end{aligned} \tag{5.40}$$

$$\begin{aligned} &msgBytes(fq_{cc}, fq_n, fq_{fx}, fq_{fy}, fq_{fz}, cd, pe, h) = \\ &msgC(fq_{cc}, fq_n, fq_{fx}, fq_{fy}, fq_{fz}, cd, pe, h) \times \{bytes_{int} \| bytes_{real}\} \end{aligned} \tag{5.41}$$

5.5.2 Intra/Inter-Node Communication

From Section 4.3.2, it is known that the overall point-to-point data exchange process in Hydra consists of multiple stages, repeated for each communication direction:

- A pack phase, where data is transferred from multiple distinct arrays into a contiguous buffer for communication;
- The initiation of non-blocking MPI ISend/IRecvs in up to two directions, per data-type, which are then blocked by an MPI Waitall;
- An unpack phase, where data is moved from the received data buffer into the ghost cell locations of various buffers.

The multiple components require distinct models for each part, in particular only a portion of the point-to-point communication stage consists of "meaningful" MPI communication over a network interface, the remainder being overhead/setup costs or synchronisation delays. Any model thus necessitates the separation of the pure MPI/synchronisation network costs and the pack/unpack components (more akin to the memory/compute kernels of Section 5.4). Modelling these features requires an understanding of both a machine's network characteristics, with and without contention, and the network communication patterns of Hydra with a scaling process count.

Focusing upon the Madvm communication stage, possessing the largest MPI messages, Figures 5.2(a) through 5.2(c) present the minimum and maximum exchange stage walltimes for the Madvmx, Madvmy and Madvmz functions respectively. Despite functionally executing the same communication stage/pattern, with the same message sizes, only the timings for Madvmx and Madvmz are similar – Madvmy demonstrates a more substantial variance for the maximum timings. However, the timings for process counts that are multiples of twelve remain consistent across each of the three functions – these peaks only occur at process counts where there is a socket in use with idle cores, suggesting a potential synchronisation/idle process problem that results in measurement spikes.

Given a fixed order of operations, each of the *Madvm* functions begin with a communication phase, which itself is preceeded by two Madvx, Madvy or Madvz kernels (depending upon the function), resulting in any compute imbalance manifesting itself in any synchronisation costs of the communication phase. Examining the difference between the maximum and minimum time spent in these compute kernels, it can be seen that the kernels for Madvy demonstrate a noticeably higher fluctuation, likely the cause of the behaviour demonstrated in Figure 5.2(b). Summing the minimum exchange time and this difference exhibits a pattern that closely mirrors that of the maximum exchange time, lending credence to this idea, with the presence of a higher compute variance

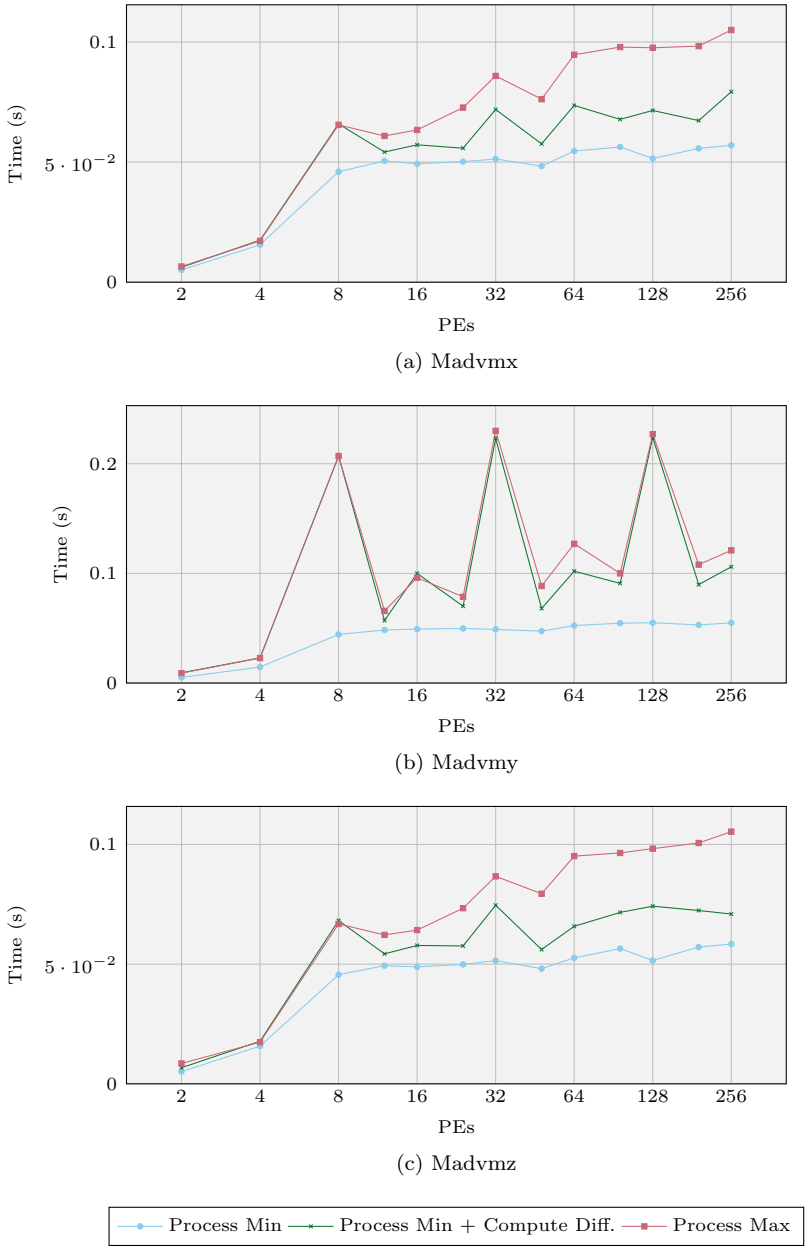


Figure 5.2: Madvm Exchange Stage Walltimes - Minerva, OpenMPI-1.4.4

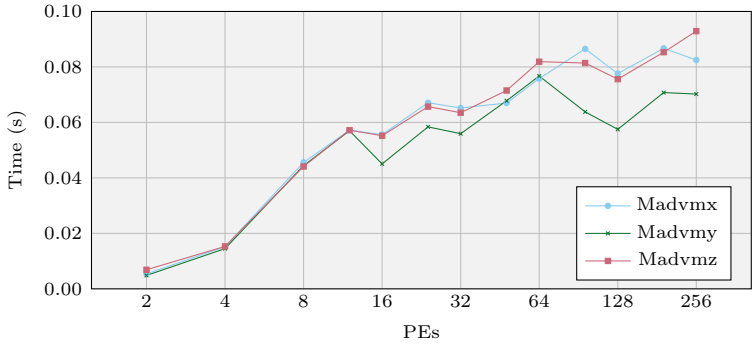


Figure 5.3: Maximum Exchange Time Minus Compute Difference

in the Madvy function also translating into a greater variance in the communication step, the removal of this compute imbalance leading to more similar performance as per Figure 5.3. However, this is not to say that the difference between the minimum and maximum time is solely attributable to synchronisation – variance in the number of messages for each process, the pack/unpack costs, the size and direction of these messages including whether they are intra or inter-node etc. and any contention behaviours are also possible factors. For example, the transitions at 12 processes sees the first introduction of processes with messages in two directions rather than one, resulting in a gap between the maximum time, and the sum of the minimum time + compute-imbalance, though due to the size of the compute imbalance this has a more noticeable impact on the maximum time for the Madvmx and Madvmz exchange stages. In addition, the point-to-point exchange stages are blocked only on local communications, not global, and as such can exhibit a degree of variation between processes for their completion time. Thus to carefully examine the impact of the network, it is necessary to separate these communication costs from the overheads and compute imbalance.

To examine the performance of the network individually the network usage responsible must be isolated, namely the MPI communications without idle wait time. For this purpose, the original variant of Hydra introduced in Chapter 4 is extended to include additional barriers, producing a point-to-point stage that

Listing 5.1: Barriered MPI Point-to-Point Data Exchange – Psudeocode

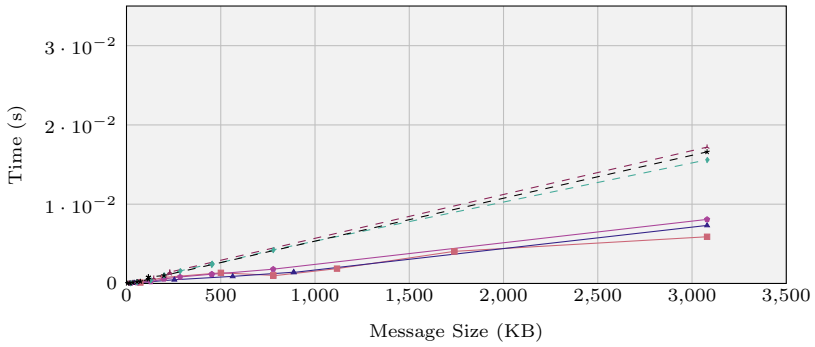
```
1 Begin Point-to-Point Comms. Stage:
2   MPI Global Barrier
3   Pack Buffer
4   MPI ISend\IRecv
5   MPI Waitall
6   MPI Unpack
7 End Point-to-Point Comms. Stage
```

PEs	Walltime (s)		Process Compute	Time in
	Master	Barrier	Range (s)	Barrier (s)
1	418.70	421.21	N/A	N/A
2	468.52	469.06	2.33E-3	3.16E-3
4	564.68	565.43	4.12E-3	4.41E-3
8	655.61	659.13	6.83E-2	7.33E-2
16	671.10	675.88	2.26E-2	2.43E-2
32	678.20	690.98	7.25E-2	7.52E-2
64	689.28	715.68	2.73E-2	2.66E-2
128	708.31	723.23	7.06E-2	7.66E-2
256	700.35	727.53	2.86E-2	3.00E-2

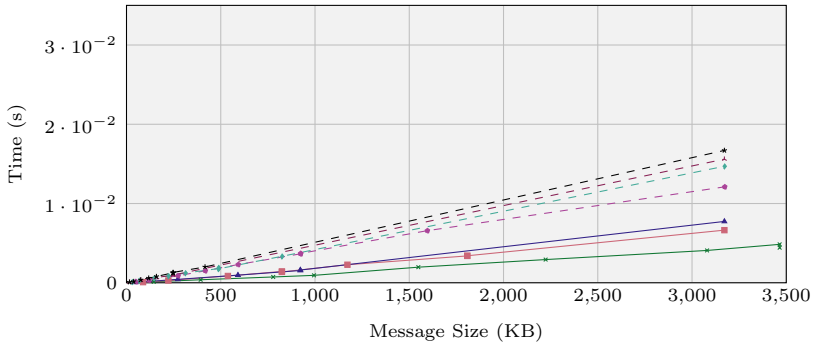
Table 5.7: Hydra Walltime - Original vs. Global Barrier - Minerva

follows the pattern in Listing 5.1, shifting any compute-imbalance synchronisation from the MPI Waitall to the MPI Barrier. While this modifies the communication interaction somewhat due to enforcing a global rather than only near-neighbour synchronisation pattern, the overall impact upon the walltime of Hydra is limited to within 5% (see Table 5.7). This small increase is deemed acceptable for the purposes of this section, due to the barriered version enabling a distinction to be drawn between synchronisation costs that might result from compute imbalance and time spent actively engaged in network communication, as well as other disparate costs such as buffer pack/unpack times.

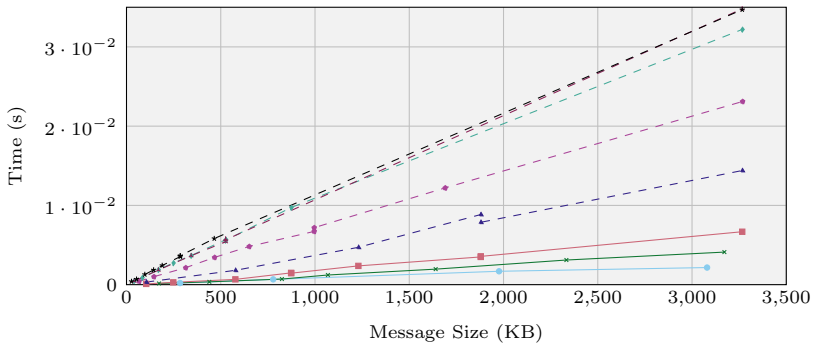
Figure 5.4 presents the MPI communication times without any extraneous synchronisation costs or data exchange overheads, focusing purely on the impact of network communication costs, with decomposition patterns consistent within a set of results for a process count. Examining a variety of problem/message sizes and process counts, it can be seen how these times scale not only with message size, but with the number of active communications, potential an outcome of contention due to the number of requests on a single communication link per node. It is also apparent that there exists a difference in performance depending



(a) X Dimension



(b) Y Dimension



(c) Z Dimension

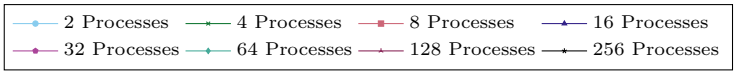


Figure 5.4: Hydra with Barrier MPI Scaling, Max Comm. Time – Minerva, OpenMPI-1.4.3 – Single Face (Solid Line) vs Double Face (Dashed Line)

P	Node	Decomposition	Intra-Node			Inter-Node		
			X	Y	Z	X	Y	Z
2	0	$1 \times 1 \times 2$	0	0	1	0	0	0
4	0	$1 \times 2 \times 2$	0	2	2	0	0	0
8	0	$2 \times 2 \times 2$	4	4	4	0	0	0
16	0	$2 \times 2 \times 4$	6	6	8	0	0	4
32	1	$2 \times 4 \times 4$	6	8	4	0	2	16
64	2	$4 \times 4 \times 4$	9	4	0	0	8	24
128	2/5	$4 \times 4 \times 8$	9	4	0	0	8	24
256	3/6	$4 \times 8 \times 8$	9	8	0	0	8	24

Table 5.8: Number of ISend/IRecv Pairs Total (Worst-Case Node)

upon the direction of communication, likely the result of different intra or inter node communication patterns depending upon Hydra's configuration. There are a small selection of behaviours that this can be attributed to.

First, the patterns usually fall into one of two sets – a single-face or double face communication. This is influenced by the decomposition pattern, where within a dimension a communication may be required in both directions rather than just one. This will result in two MPI messages rather than one and at a minimum should correspond to a doubling of the time for a communication over the single-face equivalent (assuming a single saturated communication link). Second, as the number of processes increases there is a shift in the number of both (a) simultaneous communication pairs and (b) the proportion of intra-node to inter-node communications for a particular dimension. The behaviours exhibited by each of the dimensions are dissimilar due to the differences in these decompositions, and are summarised in Table 5.8 as the number of ISend/IRecv Pairs. From this, the patterns for each of the dimensions can be identified as follows:

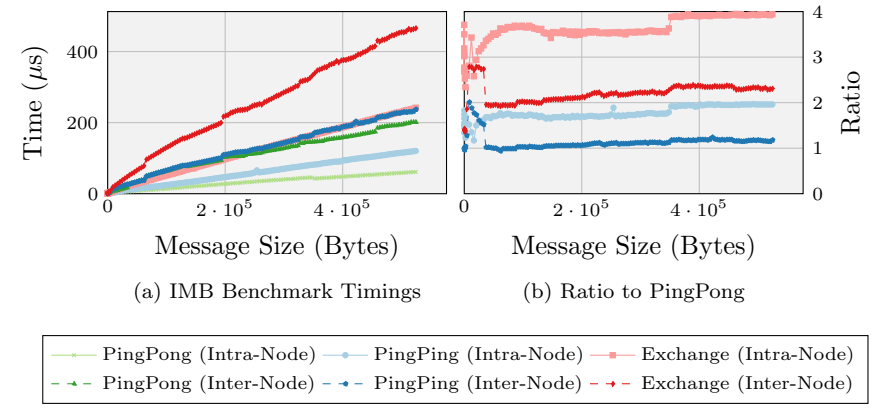
- In the X dimension, the decomposition for this process set never extends past 4, resulting in small process exchange chains that fit neatly into the node size of 12 processes (either 6×2 processes or 3×4 processes). As such, with an increase in the number of processes there is a corresponding increase in the number of simultaneous pairs of communications intra-

node, but no communications ever go inter-node at this scale. When the number of Send/Recv pairs reaches a cap, the timings tend to cluster around similar values;

- In the Y dimension a similar behaviour can be observed, but due to the gradual introduction of more inter-node communications the timings are somewhat more spread than for the X dimension;
- Finally, in the Z dimension, both a greater spread and higher magnitude can be observed due to the tendancy for these communications to be inter-node. Once again the clustering can begin to be seen due to the number of inter-node communications reaching a cap.

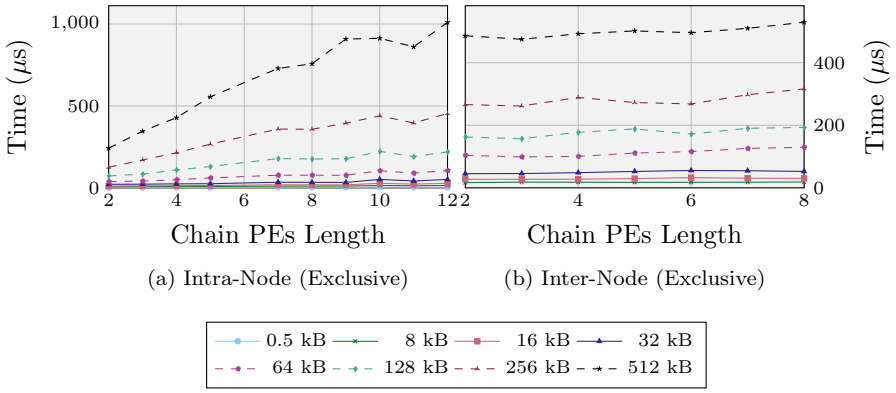
As the final modelling step, it is necessary to convert this knowledge of communication patterns/message sizes, in conjunction with network benchmarks, into predictions that are similar to the empirical communication times as measured from the Hydra benchmark.

Figures 5.5(a) through 5.7(b) present the outcome of various Intel MPI Benchmark (IMB) experiments into the impact of contention on the network behaviour of Minerva. The benchmarks used include PingPong, reporting half the round-trip time of a send/recv communication pairing between two processes; PingPing, the time to complete a send/recv between two processes simultaneously; and Exchange, the time to complete a send/recv communication with processes $n-1$ and $n+1$ for a process n across the set of all processes (this behaviour includes looping such that for x processes, process $x-1$ and process 0 communicate with one another). The Exchange benchmark is the closest in communication pattern to that of Hydra's Point-to-Point Exchange stages, although Hydra's communication patterns lack the looping behaviour present in the IMB benchmark. To investigate contention, the "multi" variants perform the same benchmarks, but with multiple pairs and/or chains of processes executing simultaneously.



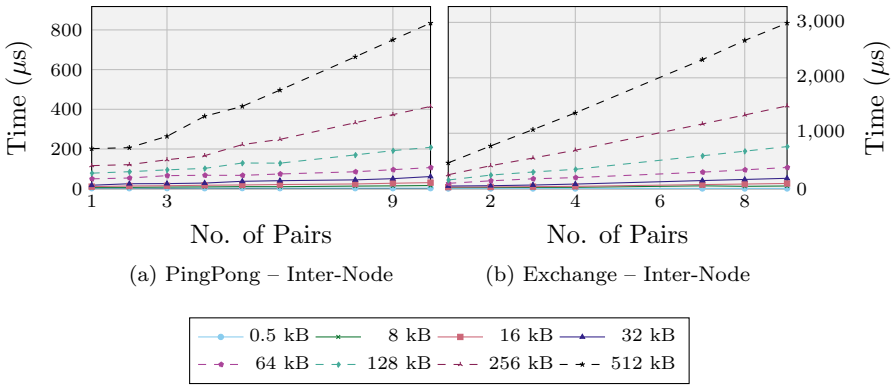
(a) IMB Benchmark Timings (b) Ratio to PingPong

Figure 5.5: IMB PingPong vs PingPing vs Exchange - Single Process Pair



(a) Intra-Node (Exclusive) (b) Inter-Node (Exclusive)

Figure 5.6: IMB Exchange (Process Chain Scaling)



(a) PingPong – Inter-Node (b) Exchange – Inter-Node

Figure 5.7: IMB Simultaneous Chains (Chain Length 2 PEs)

Figure 5.5(a) investigates only a single pair of processes to contrast the difference between PingPong, PingPing and the Exchange benchmarks. PingPong is reporting the time for a single Send/Recv (due to the measurement of the half round-trip time), while PingPing reports the time for two Send/Recv executed simultaneously while Exchange is conducting four Send/Recvs simultaneously due to the looping nature of the benchmark. As such it would be expected that, in the absence of contention with the PingPong time as a baseline, the PingPing benchmark should be roughly equivalent (twice as many messages, but overlapping), and the Exchange benchmark 2× as long (due to the setup here leading to the benchmark effectively mimicing a PingPing benchmark, but with two rounds of communications). Upon examination of the benchmarked times, this pattern appears to hold for inter-node communications, where there is a small increase for the PingPing benchmark of up to 15% when contrasted with the PingPong benchmark, and likewise for the Exchange benchmark vs 2× of the PingPong benchmark (Figure 5.5(b)). It is possible that this overlapping capacity is a feature of using full duplex communications, enabling a send and a recieve in both directions at once (at the cost of a small overhead). In the case of the intra-node communications, the assumption of no contention does not hold, with the PingPing benchmarks at larger message sizes being 2× that of the PingPong messages, and for the Exchange benchmarks 4×, corresponding heavily with the number of messages (Figure 5.5(b)). This would suggest that the load on the system is such that only one single Send/Recv pairing is occuring at any one time. The implications of this are that in the event of intra-node, multi-message communications such as that of the Hydra benchmark, heavy contention behaviour will be seen. The primary exception is for very small messages sizes, where latency plays a greater role than bandwidth.

To verify the above implication, Figures 5.6(a) and 5.6(b) explore the Exchange benchmark further, using a chain of processes greater than two to investigate the impact of a larger number of simultaneous connections. Following from Figure 5.5(a), the expectation is that the intra-node communications would

Intra-Node Single-Message (μs)			Inter-Node Single-Message (μs)		
Message Size (Bytes)	m **(Bytes/μs)**	c **(μs)**	**Message Size Bytes**	m **(Bytes/μs)**	c **(μs)**
< 16384	1.78E-04	5.37E-01	< 36864	4.83E-04	2.13E+00
< 352256	1.28E-04	1.82E+00	< 65536	5.50E-04	6.80E+00
> 352256	1.08E-04	4.57E+00	> 65536	3.21E-04	3.33E+01

Table 5.9: Minerva Communication Linear Regression Parameters

see heavy contention, while the exclusively inter-node communication will see a much smaller impact due to each extra process in the chain introducing an additional communication interconnect to distribute the load of additional messages, since each extra process is effectively an extra node in this setup. The outcomes of the Exchange benchmark suggest that this does appear to be the case. At larger message sizes the increase in time is sizeable for intra-node communication chains, while inter-node chains only see a marginal increase in cost.

Finally, it is neccessary to explore the impact of multiple communication pairs occuring between two nodes simultaneously. The above inter-node communication benchmarks explore the impact of introducing additional nodes, but this behaviour in isolation is unrealistic since it only enforces communication between a single core on each node, leaving the remainder idle. In doing so it does not heavily stress a single interconnect, and does not demonstrate the contention impact of multiple communications. In the Hydra communication patterns, it is possible for up to 12 pairs to be communicating between two nodes at any one time, or 24 pairs with three nodes (communication in both directions as per the Exchange benchmark). Figures 5.7(a) and 5.7(b) attempt to capture this behaviour, using the PingPong and Exchange benchmark to highlight the impact of multiple pairs of simultaneous communications between the same two nodes at once.

Using the knowledge from above, it is possible to derive a linear regression model for the time to communicate a single message, summarised in Table 5.9. In conjunction with the scaling behaviour of contending messages both intra-node and between two fixed nodes, it is possible to combine these models

with the overall critical path model and kernel models, producing an overall predictive walltime model for Hydra.

5.6 Collective Communication

`MPI_AllGather` is the primary MPI collective used within Hydra, frequently within `mdt` and with more limited use within `mlagh`.

In many respects the `MPI_AllGather` is similar to that of an `MPI_AllReduce`. However for `MPI_AllGather` as the number of communication steps required scale with P, the amount of data sent does also, assuming a pair-wise exchange, where the ranks are split into pairs and exchange data. New pairs are formed on a tree-like basis until all ranks have received from all other ranks, directly or indirectly as described in [19]. This results in a log_2 arrangement, where the amount of data sent doubles per step, and the resulting equation:

$$\begin{aligned} T_{allgather}(dts) &= \left(\sum_{i=0}^{(log_2(cpn))-1} T_{comm,intra,n}(2^i * dts) \right) + \\ & \left(\sum_{i=log_2(cpn)}^{(log_2(P))-1} T_{comm,inter,n}(2^i * dts) \right) \end{aligned} \tag{5.42}$$

where dts is the size of the initial data per process in bytes.

5.7 Model Validation

5.7.1 DawnDev/Hera

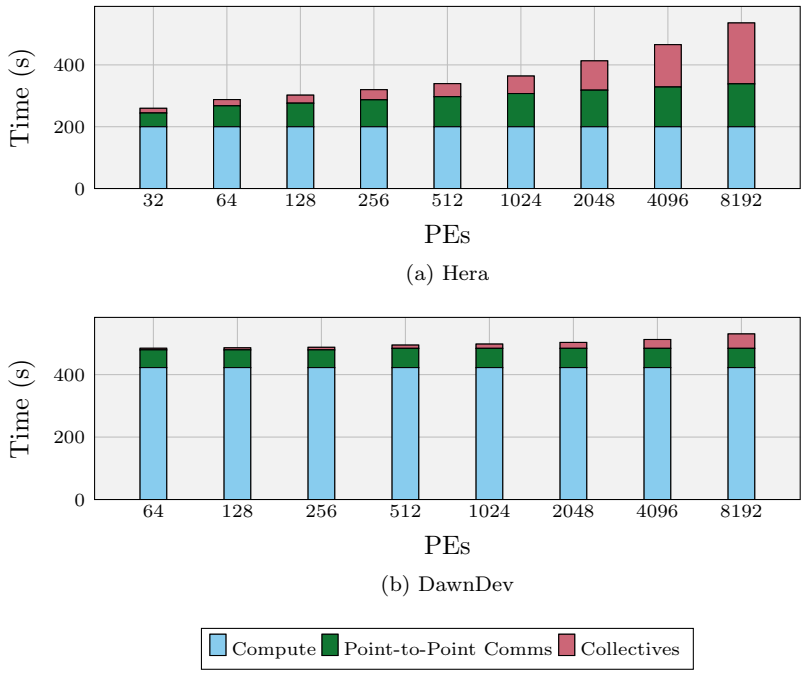
Earlier revisions of the Hydra Model were tested on two machines of interest, DawnDev (Section 3.3.3) and Hera (Section 3.3.4), in both cases up to a scale of 2048 cores as presented in Tables 5.10 and 5.11 for two different problem sizes, alongside a breakdown of the model result in Figures 5.8(a) and 5.8(b). In both cases, the error lay within 15% of the actual overall walltime, sans

	DawnDev			Hera		
Cores	**Empirical (s)**	**Model (s)**	**Err. (%)**	**Empirical (s)**	**Model (s)**	**Err. (%)**
32	-	-	-	253.3	259.55	2.44
64	505.23	484.37	-4.13	291.58	287.76	-1.33
128	525.15	485.77	-7.15	295.74	302.07	2.10
256	534.24	487.44	-8.76	310.06	319.77	3.10
512	528.76	494.53	-6.47	325.15	339.31	4.33
1024	544.43	497.75	-8.57	337.54	363.90	7.78
2048	584.97	503.04	-14.01	398.1	413.01	3.71

Table 5.10: Hera/DawnDev Model Validation, Weak Scaled, 50^3 Per Core [44]

	DawnDev			Hera		
Cores	**Empirical (s)**	**Model (s)**	**Err. (%)**	**Empirical (s)**	**Model (s)**	**Err. (%)**
32	-	-	-	806.94	783.36	−2.92
64	1665.8	1561.46	−6.26	901.91	832.89	−7.65
128	1702.63	1562.87	−8.21	905.04	859.41	−5.04
256	1718.68	1564.53	−8.97	974.75	888.32	−8.87
512	1707.78	1577.7	−7.62	1002.52	918.38	−8.39
1024	1729.02	1580.92	−8.57	1039.30	953.54	−8.25
2048	1779.78	1586.21	−10.88	1172.02	1013.88	−13.49

Table 5.11: Hera/DawnDev Model Validation, Weak Scaled, 75^3 Per Core [44]



(a) Hera

(b) DawnDev

Figure 5.8: Model Breakdown – Weak Scaled, 50^3 per Core, Hera [44]

I/O time which was omitted from both the predicted and empirical results. The demonstrated capacity shows that the model was reuseable across multiple machines/architectures, but had room for refinement.

From a breakdown perspective, the model predicts a higher collective cost attributed to Hera over DawnDev. This highlights a difference in the level of contribution associated with the network costs of the two machines; in particular how the walltime scaling differs between the two machines from 32 cores to 2048 cores, as seen in the empirical results of Table 5.10. From this, it can be posited that while Hera shows better walltimes at lower core counts, the scaling performance of the two machines would indicate that, due to the communication costs, DawnDev would reach a threshold point where it becomes the better performing machine of the two. Given knowledge of the BlueGene architecture from Chapter 2, and how it is designed around weaker individual chips in a dense interconnected fashion, it might be expected that the design philosophy of the machine would also hint at this conclusion.

5.7.2 Minerva

Using Minerva, three distinct cases were explored to further refine this model and demonstrate its applicability for modelling different scaling/compute/communication behaviours. Table 5.12 presents the outcome of a serial performance investigation, where the captured walltime is solely compute without any network/MPI communication. As demonstrated, the model is capable of providing an accurate prediction of the runtime for a serial execution, confirming the capture of the vast majority of the application's compute behaviour, as well as the critical path. For the most part the model captures the relatively linear relationship between the number of cells and the time taken. However, during the course of the modelling process a few select kernels were revealed to have unexpected behaviour, differing from the initially predicted linear relationship; these kernels were $Madvmz_1$ and to a lesser possible extent $Madvmy_1$. Nevertheless, this behaviour was handled via the use of piece-wise regression for

Problem	Empirical (s)	Model (s)	Err. (%)
30^3	11.18	11.40	1.99
50^3	50.94	49.96	−1.93
80^3	205.09	202.98	−1.03
100^3	418.70	415.57	−0.75
120^3	809.37	809.23	−0.02
150^3	1941.77	1933.57	−0.42

Table 5.12: Hydra Model Validation, Serial, Minerva Intel-12.0/OpenMPI-1.4.3

	Total Walltime (s)		
PEs	Empirical	Model	Error (%)
1	418.7	415.57	−0.75
2	468.52	460.61	−1.69
4	564.68	557.59	−1.26
8	655.61	668.92	2.03
16	671.1	677.56	0.96
32	678.2	688.92	1.58
64	689.28	699.38	1.47
128	708.31	699.40	−1.26
256	700.35	703.64	0.47

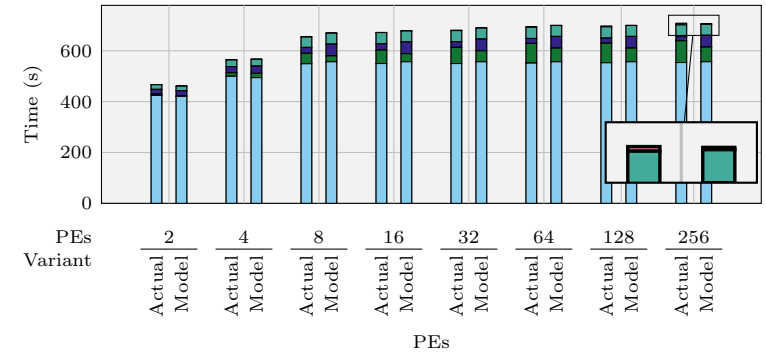
Table 5.13: Hydra Model Validation, Weak Scaling, Minerva Intel-12.0/OpenMPI-1.4.3

determining the Wg values and cell ranges.

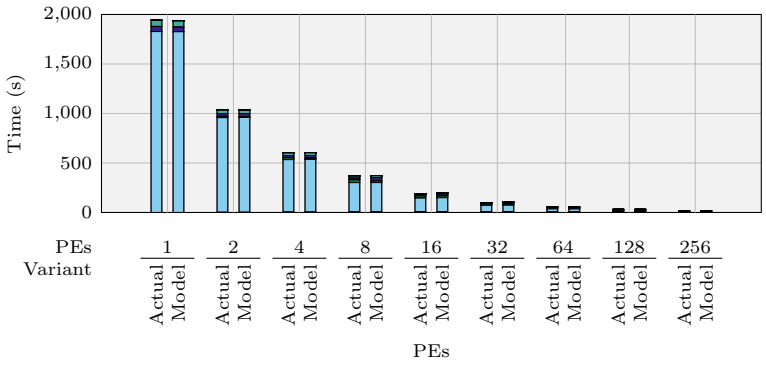
The final weak and strong-scaling results in Tables 5.13 and 5.14 demonstrate the capacity of the model to also incorporate the network communication costs, remaining within 10% of the empirical results. With the model validated on multiple machines for different compute and network hardware, this suggests that the model is applicable for the purposes of performance prediction on differing architectures. The introduction of a parallel element also demonstrates the model's capacity to predict the separate breakdown costs of various components. In Figures 5.9a and 5.9b, the overall walltime of both the empirical results and the model prediction have been broken down into five components; these components are the compute kernels, the overhead of dynamic memory allocation and deallocation, the update boundary kernels (distinct from compute kernels due to them being primarily involved in shifting data in memory on boundary cells only), the point-to-point MPI communication costs and the collective MPI costs. At this scale the collective costs are insignificantly small in comparison to the other contributors. The compute costs are relatively close, as expected from

	Total Walltime (s)		
PEs	**Empirical**	**Model**	**Error (%)**
1	1941.77	1933.57	−0.42
2	1035.26	1031.49	−0.36
4	601.47	601.92	0.07
8	367.41	370.37	0.81
16	187.3	195.65	4.46
32	97.21	103.27	6.23
64	54.62	55.48	1.58
128	31.8	31.16	−2.00
256	19.97	18.15	−9.12

Table 5.14: Hydra Model Validation, Strong Scaling, Minerva Intel-12.0/OpenMPI-1.4.3



(a) Weak-Scaling



(b) Strong-Scaling

Figure 5.9: Model vs Empirical by Component Breakdown, Minerva

the serial experiments prior. The communication costs underpredict a small amount but follow a similar scaling trend to that of the empirical results. The primary difference is due to the model predictions being based on a PE that has more external boundaries (process 0) but fewer internal boundaries. As such some time shifts from the communication costs towards the update boundaries costs due to synchronisation etc. The numbers for these Figures can be found in Appendix Tables A.25 and A.26. As a group the model-predicted components observe similar patterns to that of the actual Hydra runs, reinforcing the confidence that might be placed in the model predictions. As such, it is especially applicable when interest in the model lies in its application for procurement or investigative prediction for larger-scale machines – in such cases it is often the speedup trend rather than absolute numbers that are most important.

5.8 Summary

This section has introduced a performance model of the Hydra benchmark, demonstrating an error rate of 15% or less on three distinct architectures. it has shown a capacity for both serial and scaling benchmark predictions, showing how it accounts both compute and network costs, factors that are crucial in the model's viability for the purposes of performance prediction at greater scales.

CHAPTER 6

Optimisation

While the primary goal of a scientific simulation is to produce an accurate output, it is a given that the most desirable setup is one in which said results are obtained as quickly and reliably as possible. As the High Performance Computing (HPC) community and industry moves towards Exascale platforms, understanding scalability constraints becomes ever more crucial for parallel applications [20, 28, 155]. Bottlenecks that do not exhibit themselves at small scale can increase and supplant previous performance constraining factors, impeding effective machine utilisation. Identifying these bottlenecks can be difficult due to their nature; without reliable predictive capabilities they can only be uncovered by executing runs on large-scale machines, a costly and prohibitive procedure. In addition, with no Exascale machines yet in existence, insights regarding future scalability performance remains uncertain. Until such architectures are in production, it is only possible to theorise based on the performance of smaller scale machines through the use of analytical models and simulation. Using these techniques for the identification and elimination of potential bottlenecks is thus a priority for not only current scalable architectures but also emerging large-scale systems.

On current high-performance clusters, the expected performance of an application can fall short of the peak performance (as measured by LINPACK) due to differences in how the application operates. Given the expense involved in constructing and operating a typical supercomputing cluster, this raises concerns for the scientific return on investment. It may be preferable to fully capitalise on the capabilities of an existing machine before looking to adopt a new architecture or expand with additional nodes, especially considering the extensive list of

Variant	Description	Ref.	Based On
A	Original Hydra	4	N/A
B	Memory — 2D Loop-Interchange	6.2	A
C	Memory — 3D Loop-Interchange	6.2	A
D	Vectorisation — C Port	6.3	C
E	Vectorisation — Vector Intrinsics	6.3	D
F	MPI Overlap — No Comm Dependancies	6.4	C
G	MPI Overlap — Refactored Compute	6.4	F
H	MPI Overlap — Non-Blocking Overlap	6.4	G
I	Threading — OpenMP Static Schedule	6.4	C
J	Threading — OpenMP Dynamic Schedule	6.4	I
K	Threaded Overlap — Refactored Compute	6.4	J
L	Threaded Overlap — Comms Thread	6.4	K

Table 6.1: Summary of Hydra Variants

procurement factors that must be accounted for such as reliability, space, cooling, power consumption and maintenance. However, the improvement of both current and potential future large-scale performance is not mutually-exclusive.

Identifying the bottlenecks that lead to under-performance is crucial to improving the performance of these applications. When considering the source of a bottleneck on a given architecture, it is typically determined by the machine's components and their effective use. These potential bottlenecks are categorised as (a) compute-bound, (b) memory-bound, (c) I/O-bound, (d) network-bound and (e) algorithmic. Addressing only a single category offers the potential for improvement, but will see the shifting of the bottleneck elsewhere. In addition, it does not account for other influential factors that occur as a consquence of the interaction between them such as the impact of poor load-balancing upon both compute performance and the synchronisation stage of communication. Thus, a multi-faceted strategy is required.

This chapter investigates and explores a selection of optimisation techniques that are potentially applicable to Hydra. Table 6.1 summarises the various different implementations employed within this chapter, with reference to the section where each is introduced in more detail. Some of these variants are not direct optimisations but act as controls for comparitive purposes, so that performance changes that are an outcome of the code refactoring neccesary for implementation and the optimisations themselves can be differentiated. Some implementations expand upon previous code modifications, so an additional

reference is provided for which code each variant is based upon. A comparison between a new variant and its predecessor enables the attribution of any performance changes solely to the delta changes between the two.
In summary, this chapter addresses the following:

- From knowledge of the underperforming kernels in Section 4.5, relationships between kernel performance profiles and their underlying memory patterns can be drawn. This is validated using the Performance Application Programming Interface (PAPI) framework in conjunction with hardware counters. Using this knowledge, the application of improved memory access patterns leads to approximately a 1.3× to 1.4× speedup in walltime;
- Through the further use of PAPI/hardware counters the impact of vectorisation — a Single Instruction, Multiple Data (SIMD) technique that allows for the execution of multiple compute instructions simultaneously on a single core — is examined with regards to their machine-load (operations:memory access ratio) profiles. This identifies both compute and memory bound kernels;
- In addition to these memory and compute optimisations, a potential for improvement exists in network behaviours. Focusing predominantly upon the use of compute-communication overlap, new implementations are created that explore the use of two distinct techniques — MPI non-blocking overlap (in tandem with the requisite compute refactoring), and thread-based overlap where the use of an MPI/OpenMP hybrid allows for the use of a master communications thread alonside multiple compute threads.

6.1 Optimisation Potential

To guide optimisation efforts, it is necessary to establish what opportunities are available to ensure the effective use of valuable development resources. In an ideal scenario all machine resources would be fully utilised in perfect balance.

In reality however, it is far more likely that one or more components will become a performance bottleneck, hindering the capability of other components. The initial approach to any optimisation effort should be with the identification and elimination of such bottlenecks where possible. Such bottlenecks can take a number of forms, including:

Compute-Bound Optimisation

When a code is compute-bound, the kernel walltime is restricted by the performance of the Central Processing Unit (CPU); the CPU is incapable of processing data faster than it is being supplied either due its number of Instructions Per Cycle (IPC) or the number of cycles per second (its clock speed). Improving the number of instructions processed per cycle (e.g. with vector instructions) can aid in the overall performance of such kernels.

Memory-Bound Optimisation

Memory-bound kernels are restricted by the throughput of the memory architecture, being unable to supply data fast enough to prevent the CPU from sitting idle. The cause can be due to the effective memory bandwidth of the system, memory exclusive instructions that require no floating-point operations or ineffective memory-access patterns that lead to cache-misses upon a failure to preload the cache with the appropriate data, stalling the CPU. Minimising the number of memory operations or improving data access patterns can help to relieve such bottlenecks.

Network-Bound Optimisation

Network bottlenecks occur when parallel processes are required to wait idle on the resolution of blocking network communications to resolve remote data-dependancies. This ties the ongoing progress of the application to the speed at which said data can be received/transmitted, either due to the distance/overheads (latency) of a message transmission or the capacity of the

network (bandwidth). In parallel applications it is typically the case that as the number of distributed compute devices scales the cost of data communication can come to represent a significant cost in the overall computation. The use of communication overlap, where network operations are executed simultaneously with unrelated/independant compute operations, can help to mask the cost of such bottlenecks.

Input/Output (I/O)-Bound Optimisation

An HPC application can experience I/O-bound behaviour when waiting on significant data transmission to an underlying filesystem for the purposes of data retrieval or writing checkpoints (recovery points in case of failure). Addressing contention issues or the use of more efficient parallel filesystem operations can help reduce their impact upon performance [176, 177].

Algorithmic

Rather than attempting to relieve bottlenecks through the optimisation of existing operations, an alternate approach can be to seek the elimination of such operations entirely, removing by extension the bottleneck they introduced. Depending on the capacity of the underlying machine hardware, shifting the dominant application bounding behaviour from one category to another through algorithmic changes (e.g. compute-bound to memory-bound), can result in improved performance by placing the burden of operation on a machine's more capable sub-systems. Likewise, alternate algorithmic approaches may eliminate certain elements of compute entirely, improving the overall efficiency of the application and by extension its performance. However, it can be difficult to predict in advance whether this will result in improvements without in-depth knowledge of both the application and the hardware, unless the algorithm is objectively of a reduced complexity.

For Hydra I/O investigations are left to potential future work (any such timings are ignored or subtracted where appropriate), while the application remains largely the same algorithmically (bar changes made to support the optimisation of the other characteristics). For the remaining categories, identifying the impact of each requires suitable metrics that capture the performance of Hydra's various components, most notably:

- Floating-Point Operations per Second (FLOP/s) — Kernels that fall short of the potential peak FLOP/s rate of the system either fail to use all the available instructions per cycle (e.g. vectorisation), or stall due to insufficient data throughput. PAPI verification shows that the vast majority of Hydra's prominent compute kernels have no single-precision floating-point operations, and where present their number is relatively insignificant. Therefore in this work the attribute Double Precision Floating Point Operations (DPOPs) is taken to be interchangable with the number of floating-point operations; unless specified otherwise any FLOP/s values are calculated from the number of double-precision operations measured by PAPI and the kernel walltime.

- The number of Vector Operations (VECOPs) — This allows for establishing whether a kernel is fully vectorised.

- Cache Hit Rate — Establishing whether effective use is made of the memory heirarchy, and the effectiveness of the kernels at avoiding higher level, low bandwidth memory transfers. Optimal memory access patterns will support the effective reuse of cache where possible, minimising the potential for a memory bottleneck.

- Compute/Communication Breakdown — The use of some Message Passing Interface (MPI) parallel communication is inevitable in all but the most embarrassingly parallel of problems. As already captured in Section 4.5, where the contributions of both compute and MPI communications

towards the overall walltime was examined, it is this that limits the effective overall speedup as per Amdahl's or Gustafson's Law.

To this end, our modelling efforts allow for the use of both our understanding of the application and *model-led optimisation* — optimisation efforts that are guided by the predictive capabilities of performance models to either (a) examine areas that are underperforming beyond what might be expected, or (b) identify alternate configurations that may allow for speedup opportunities. Existing works have already demonstrated how such approaches can be used to identify disparities between expected and actual performance [89, 141], and how corrective efforts can be undertaken to restore performance.

Improving the level of optimisation is more than just identifying a machine's peak theoretical performance. The concept of *machine balance* [34, 110], the relationship between a machine's ability to perform (Floating-Point Operations (FLOPs)/cycle) and its ability to supply sufficient data for processing (Words/Cycle or Memory Bandwidth), is key to its real-world behaviour. A machine is in balance when the rate at which it can receive data and the rate at which it can process said data are equivalent; when this is not the case, approaching peak performance is prevented by one of these factors. Roofline [173] provides a visual model for capturing the performance behaviour of various kernels, drawing attention to the relationship between the peak memory bandwidth, an algorithm's arithmetic intensity and the corresponding achievable performance as a result of the machine's balance. This quantifies the potential performance improvements on offer from increasing the empirical, measureable arithmetic intensity, either through increasing the number of operations per cycle or reducing the quantity of memory transfers occuring. In doing so, it highlights the importance of establishing the machine balance of an algorithm and how the theoretical speedup of some optimisations may be inhibited by bottlenecks that inhibit such gains.

Figure 6.1 provides two metrics, FLOP/s and number of DPOPs per cache access, for a selection of kernels that constitute major performance hotspots.

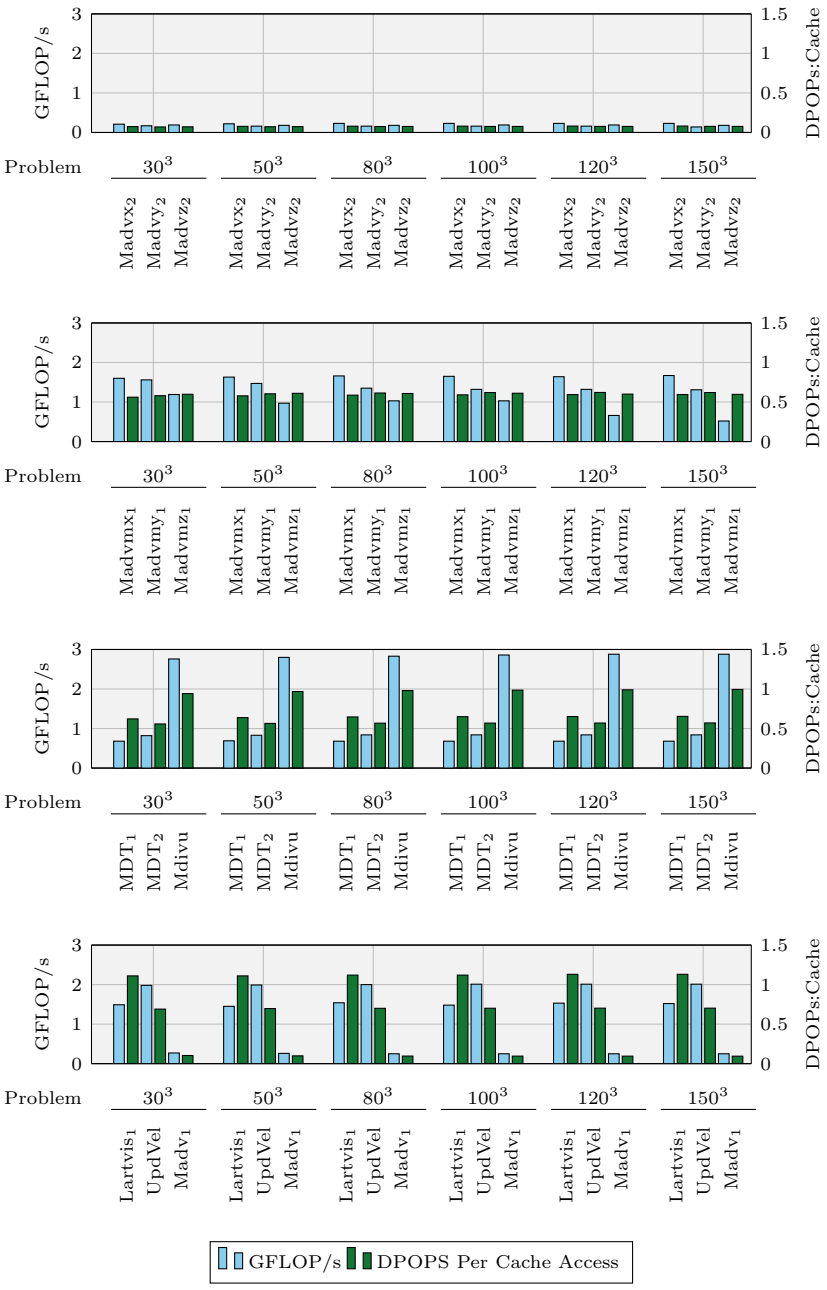


Figure 6.1: Hydra Kernel FLOP/s and DPOPs:Cache Access Ratio — Serial, Minerva (No Vectorisation)

Such metrics can prove useful in an initial analysis as the FLOP/s count may reveal underperforming kernels in terms of "useful" work, while the number of DPOPs per cache access (the machine balance) can reveal which kernels suffer more from memory overheads or instructions with high cycle latency. A low DPOPs per cache access can reveal a kernel that has relatively few floating-point operations generally (e.g. a memory copy) or has a very high number of accesses per operation (signifying potentially high cache miss rates). It is noted here that due to the restrictions of the available PAPI counters, with the L1 hit/total accesses counter being unavailable on Minerva, the total number of cache accesses was obtained from the readings of a separate machine using an Intel Xeon 3065 CPU. The same compiled binary was used to maintain consistency where possible. This restriction makes it difficult to draw precise conclusions about the behaviour on Minerva; nevertheless, it should suffice for the purposes of initial kernel ratio approximations — while the cache hit/miss ratios will vary with differing cache sizes, the number of total accesses would be expected to be more consistent with the same instruction stream across machines. All other PAPI metrics (DPOPs, L1 Cache Miss, L2 Cache Hit/Miss etc.) are obtained from Minerva readings exclusively.

As might be expected, the typical trend is that those kernels that perform more DPOPs per cache access also see higher FLOP/s counts. Given the wide variation in kernel performance, it is readily apparent that few to no kernels have hit the peak compute performance capable on the machine, therefore any reduction in memory overhead is likely to see a corresponding increase in FLOP/s. Those kernels that possess fewer DPOPs per cache access are expected to be more heavily weighted towards performing memory operations, such that they also see a reduction in the FLOP/s count. This can be because the time per memory access is taking longer, or because there are fewer floating-point operations total compared to the number of memory operations. While some operations present are memory-exclusive (such as data copies for out-of-place algorithms), the majority of these kernels' workload is oriented towards floating-

point operations (raw PAPI numbers for these operations are provided in Tables A.27 — A.38, Appendix A).

However, there are a few unexpected exceptions to these behaviours. The *Lartvis* kernel possesses the best DPOPs count per cache access ratio of all the kernels, yet falls short of the FLOP/s rates achieved by kernels with lesser DPOPs counts per cache access such as *Mdivu* or *Update Velocity*; the *MDT* kernels also exhibit this trait, albeit on a smaller scale. In addition to this, the *Madvmz* kernel (and to a lesser extent *Madvmy*), despite showing a slightly better DPOPs:Cache Access ratio than the *Madvmx* kernel has a significantly worse FLOP/s count as the problem size increases, implying that there is an additional factor that is detrimental to performance tied to the overall problem size. From this it is clear that, while useful for an initial analysis, these metrics are insufficient alone to capture the underlying performance issues.

Given the differences between the FLOP/s value and the DPOPs:Cache Access ratio, one of two conclusions can be drawn — (a) a DPOP in some kernels takes longer per cache access or (b) a cache access in some kernels takes longer than others on average; while some kernels may contain exclusively memory-only operations, these would be expected to bring down both metrics. From an understanding of the machine architecture alone (Section 3.3.1) it is known that different levels of cache access have varying access times as you progress through the memory heirarchy, as confirmed by the STREAM bandwidth results in Section 3.3.1. When a single cache access could consist of either an L1 cache hit or an L1 cache miss with an L2/L3/main memory hit it is therefore the case that knowing the ratio of cache hits to misses also becomes of great importance. In addition to this, there are a number of factors that impact the instruction per cycle rate, including branch mispredictions, instruction pipelining or the use of operations with higher instruction latencys such as square root functions [56].

6.2 Memory Optimisations

Section 6.1 has readily demonstrated that there exists a significant amount of variability in performance across the various kernels of Hydra. In an ideal scenario, all such kernels would fully utilise the CPU, assuming a compute-bound environment. However even without considering the peak, the difference in floating-point operations implies the existence of an additional bottlenecking factor that prevents maximal efficiency. It is likely that additional factors such as memory bandwidth are an important contributor to the time taken by these various kernels — those kernels that have a low DPOPs:Cache Access ratio typically also exhibit fewer FLOP/s.

When an application is found to have the majority of its walltime/performance dominated by memory operations, it is classified as a **memory-bound** code. The work of Wulf [178] on the memory wall and Wilkes [172] on the physical constraints of CMOS miniturisation highlight potential future impediments to performance from a mismatch in the rate of improvement of memory and CPU hardware, further works aptly demonstrate the impact of this balance between CPU and memory bandwidth/latency [33, 110]. Programs that exhibit these characteristics frequently involve the manipulation of a significant amount of data, with memory access performance remaining crucial to modern HPC applications [7, 8, 113, 126, 169].

Such trends threaten to overshadow gains made from increasing the number of processing elements local to a device; if the memory throughput is insufficient then data cannot be transferred rapidly enough to fully exploit this increase in compute power. The nature of the problem is such that scaling distributed systems will not suffer from memory-bound issues (beyond memory accesses required for communication), but scaling Shared Memory Parallelism (SMP) systems has the potential to introduce contention on an already saturated memory bus (and by extension, also applies to hybrid SMP/distributed systems such as clusters). This problem will manifest itself as the total number of cores on a

single socket/node increases.

To mitigate this, understanding a machine's memory architecture becomes crucial to engineering solutions that make effective use of the available resources. As well as the presence of different memory levels and speeds (see Section 2.2.2), more novel structures exist such as NUMA regions [24] or shared levels of cache that can complicate matters. Implementing memory access patterns that use the cache effectively, enforcing access patterns expected by preloading algorithms, is neccessary to ensure good throughput for any scientific application. Identifying areas that exhibit the signs of poor memory access patterns is thus key to achieving this goal.

In Section 4.5.1 it was noted that unexpected performance issues were exhibited by the compute — most notably that of the *Madv* function. When computing Wg values for the predictive model as per Section 5.4, it was apparent that this was especially prominent in a subset of the compute functions. As the problem size scales it is known that the performance of selected kernels such as Madvmz_1 is greatly diminished, especially when constrasted against a kernel that has similar functionality, Madvmx_1. This similarity gives cause to query the increase in cost for Madvmz_1, as it goes against expectations given the comparable levels of compute between the two kernels. The model using just linear regression rather than piece-wise regression would predict a substantially reduced walltime for this particular kernel when contrasted against the empirical measurements, hinting at some form of performance deterioration. Through the application of the PAPI framework and selected hardware counters, it is possible to investigate the potential causes.

Since a change in FLOP/s value can represent either an increase in DPOPs or a decrease in time taken, Figure 6.2 presents the FLOP/s values in a different manner distinguishing between the "useful" work done (DPOPs and the time taken to complete it. The results here are conducted with vectorisation disabled, thus one DPOP represents a single operation. It is confirmed that there is indeed a similarity in the amount of DPOPs performed within the two

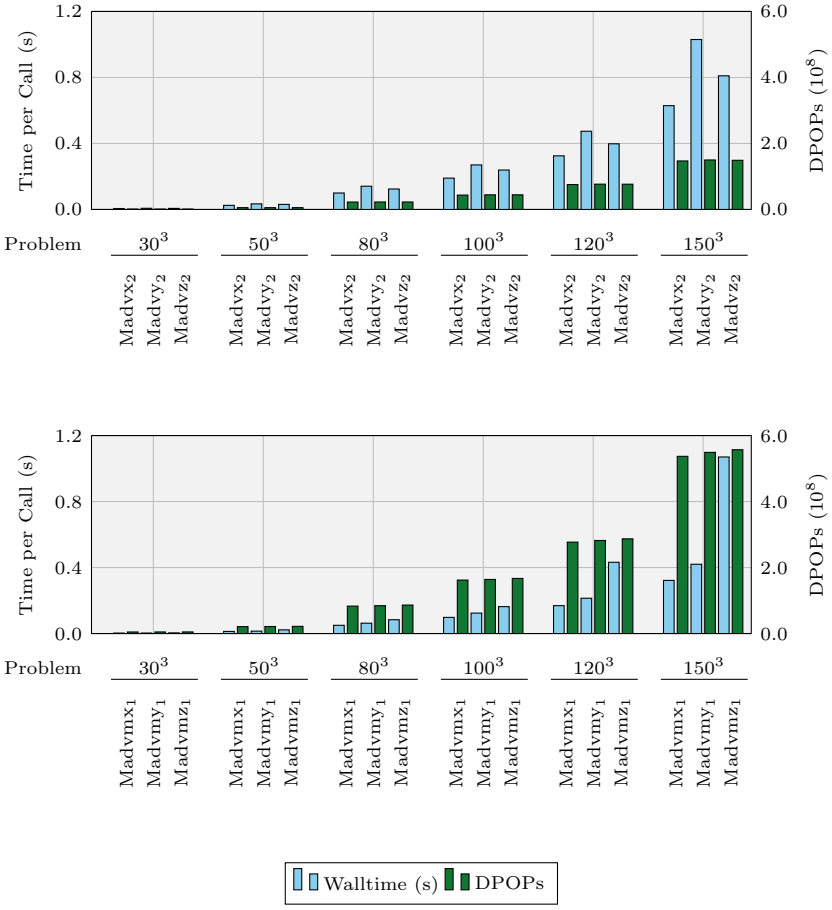


Figure 6.2: Minerva, Hydra Serial Execution, Walltime vs. Mean Kernel DPOPs

dominant sets of kernels in the *Madv* function, *Madvmx*/*Madvmy*/*Madvmz* and *Madvx*/*Madvy*/*Madvz*. However, this trend does not translate across to the walltime, where within the kernel sets the time per kernel execution is far more disparate, with the Y and Z dimension variants typically taking longer than the X variants. This is especially the case for the *Madvmz* kernel, where the walltime is over twice that of similar kernels *Madvmx* and *Madvmy* at 150^3. Further to this, comparing between the two sets of kernels reveals that *Madvx*, *Madvy* and *Madvz* exhibit a far more significant walltime in relation to their DPOPs count (as suggested by their FLOP/s rate in Figure 6.1 prior). A factor

other than the number of DPOPs occupies a significant amount of their wall-time, meaningful or otherwise. However, it is known from Figure 6.1 that, while there is some minor difference, the number of DPOPs per cache access is also roughly similar. Therefore it can only be concluded that the time taken to perform a DPOP is different, either in terms of instructions (such as latency) or in the time taken to retrieve data. This section focuses upon the latter of the two, exploring the underlying performance of a cache access for these kernels.

6.2.1 Memory Access Pattern Techniques

A number of techniques exist for improving memory access patterns, mostly oriented around promoting the reuse of data in cache through the use of spatial and/or temporal locality (as described in Section 2.2.2). Their application is often dependant upon the types of kernels/memory access patterns in use, and whether there are any existing data-dependancies. For breadth listed here are a selection of different techniques introduced or used by various works [62, 114, 174, 175]

Loop Fission

Loop Fission is the act of merging two or more loops into a single loop. In doing so it reduces the overhead of operating one or more loops into the overhead of a single loop. In addition, it aims to promote temporal locality if the same location was previously accessed in two separate loops. However, it can also inhibit temporal locality by increasing the number of memory accesses that occur within a single iteration, and thus must be used sparingly if a significant number of different memory locations are touched in a loop.

Loop Fusion

Loop Fusion is the inverse of loop fission. By separating a loop into two or more separate loops, it can improve temporal locality by reducing the number of memory accesses per loop, and thus reducing the chance of a cache-line

eviction before it is reused. This is most appropriate if a loop iteration consists of a number of memory accesses to disparate and independant locations.

Loop Interchange

Loop Interchange is used when there are two or more nested loops. The order of the nested loops is manipulated in order to change the data access pattern in memory, promoting spatial locality through the use of sequential memory accesses where possible. The optimal order is typically dependant upon the language used/underlying data storage pattern. For example, C utilises a row-major order for sequential storage whereas Fortran uses a column-major order.

Loop Blocking

Loop Blocking is the act of "chunking" a block of memory into segments. When compute relies on using data from the same chunk multiple times, the overall order of compute/memory access is modified such that each chunk is fully used until no such further data is required from the chunk, before proceeding onto the next chunk. This technique is most suitable for compute that has a high FLOP:byte ratio, such as that of matrix-multiply (n^3 operations with n^2 memory usage).

Loop Pipelining

Loop Pipelining is the process of preloading cache-lines by accessing data locations before they are required by looking ahead to the proceeding loop iteration — i.e. performing memory operations for iteration $i + 1$ while the compute for iteration i is ongoing. In doing so, this "pipelines" the memory and compute operations.

Counter	Description
L1_DCH	Number of L1 Data Cache Hits
L1_DCM	Number of L1 Data Cache Misses
L2_DCH	Number of L2 Data Cache Hits
L2_DCM	Number of L2 Data Cache Misses
L3_DCH	Number of L3 Data Cache Hits
L3_DCM	Number of L3 Data Cache Misses

Table 6.2: PAPI Hardware Counter Identifiers

6.2.2 Cache Optimisation In Hydra

By surrounding the compute kernels with cache-access PAPI counters (see Table 6.2), it is possible to capture the memory access profiles across the course of a Hydra execution. In empirical tests the kernel performance is explored by capturing the mean miss rate of both the L1 and L2 caches on Minerva across each kernel call; no such performance counters were available for the L1 Hit or L3 Hit/Miss due to the lack of hardware support. This section details the modifications made to Hydra to improve these cache hit:miss ratios, and the performance improvements that accompanied them.

When considering the underlying memory performance, it is important to reflect upon the implementation of the kernel memory access patterns. The nature of these kernels is such that they require the the use of stencils for various intermediate computations. Each operates with a consideration for dimensionality within the grid, updating X (*Madvx*, *Madvmx*), Y (*Madvy*, *Madvmy*) or Z (*Madvz*, *Madvmz*) dimensional quantities, with their various stencils acting in these same respective dimensions. To minimise memory storage space temporary 1D arrays are used to store any intermediate calculations across the innermost loops, reuseable across the outermost loops. Since the use of temporary storage space is kept to a minimum, an in-place algorithm enforces a strict order of operations to prevent the loss of required data (i.e. preventing a grid data-point from being overwritten before it is used in a subsequent operation) due to the introduction of data-dependancies.

Since these two sets of kernels largely perform similar operations, with the major difference being the order of operations (see Listings 6.1 through 6.3),

Listing 6.1: Madvmx$_1$ Order-of-Operations — Variant A

```
for z in nz
  for y in ny
    for subkernels 1...k
      for x in nx
        Compute -> Results(1D Array)
    for x in nx
      In-Place Cell Update (3D-Array)
```

Listing 6.2: Madvmy$_1$ Order-of-Operations — Variant A

```
for z in nz
  for x in nx
    for subkernels 1...k
      for y in ny
        Compute -> Results(1D Array)
    for y in ny
      In-Place Cell Update (3D-Array)
```

Listing 6.3: Madvmz$_1$ Order-of-Operations — Variant A

```
for x in nx
  for y in ny
    for subkernels 1...k
      for z in nz
        Compute -> Results(1D Array)
    for z in nz
      In-Place Cell Update (3D-Array)
```

Listing 6.4: Madvmz$_1$ Order-of-Operations — Variant B

```
for y in ny
  for subkernels 1...k
    for z in nz
      for x in nx
        Compute -> Results(2D Array)
  for z in nz
    In-Place Cell Update (3D-Array)
```

Listing 6.5: Madvmz$_1$ Order-of-Operations — Variant C

```
for subkernels 1...k
  for z in nz
    for y in ny
      for x in nx
        Compute -> Results(3D Array)
for z in nz
  for y in ny
    for x in nx
      Out-of-place update (3D Array Copy)
```

the primary difference between them responsible for the differences in their performance is likely the underlying memory access patterns. To combat the high rate of cache misses, the technique of loop interchange is applied and, to a lesser extent, loop fission to break some data-dependancies by re-ordering the intermediate calculations from sub-kernels in such a way that they iterate through more spatially local memory. However, since the loop pertaining to the kernel's dimensionality must remain within the sub-kernel structure (as per the listings), this effectively results in a partial or full out-of-place update algorithm rather than in-place update scheme due to the storage of the final result in a separate location which is then copied to the final target. This improvement in spatial locality also comes at a tradeoff — the overall memory usage is higher, requiring 2D or 3D arrays rather than 1D arrays for storage, potentially causing poorer temporal locality and introducing additional memory transfer overheads. Ultimately this results in two new variants, in addition to the basic version of Hydra:

- Variant A — The original Hydra codebase used for the scaling investigations in Chapter 4. This acts as the control for intial performance comparisons;
- Variant B — In each of the X, Y and Z dimensional kernels, the order of traversal is dependant upon the 3D nested loop ordering. This variant applies loop interchange, but only upon the two innermost loops, to improve the spatial locality while compromising to reduce the introduction of additional memory usage/overheads (2D rather than 1D temporary arrays). Listing 6.4 is one such kernel example. In each instance the X dimension is promoted to the innermost loop, with the outer loop allocations being dependant upon what dependancies must be maintained;
- Variant C — This variant applies a similar approach to Variant B, but interchanges all loops to enforce an $X \rightarrow Y \rightarrow Z$ ordering with multiple intermediate 3D temporary arrays. This also permits the removal of

—	Kernel					
Variant	Madvx_2	Madvy_2	Madvz_2	Madvmx_1	Madvmy_1	Madvmz_1
A	$z \to y \to x$	$z \to x \to y$	$x \to y \to z$	$z \to y \to x$	$z \to x \to y$	$x \to y \to z$
B	$z \to y \to x$	$z \to y \to x$	$y \to z \to x$	$z \to y \to x$	$z \to y \to x$	$y \to z \to x$
C	$z \to y \to x$	$z \to y \to x$	$z \to y \to x$	$z \to y \to x$	$z \to y \to x$	$z \to y \to x$

Table 6.3: Kernel Loop Ordering — Outermost → Innermost

further data-dependancies by enabling the use of a full out-of-place algorithmic approach, at the cost of more memory storage. Listing 6.5 is one such kernel example.

These orderings are summarised for each of the six kernels in Table 6.3. In addition to these changes, in both new variants a small linked list that iterates over selected quantities is eliminated within the *Madvx*, *Madvy* and *Madvz* kernels, instead substituting it for hard coded accesses directly to each quantity in question; the functionality remains the same. The next section contrasts the performance of these three variants, examining the impact such modifications on the memory access patterns have upon the cache hit rates and overall performance.

Results

In this work changes made to Hydra are predominantly focused upon modifying the implementation to be more efficient (*how* it is calculated) as opposed to algorithmic changes (*what* is calculated). As a result, it is expected that the overall number of floating-point operations to be computed should be relatively similar across the various optimisations, while other factors such as memory or network usage are improved. Figure 6.3 shows the cache behaviour and DPOPs of Variant B and Variant C for the six modified kernels; between the variants the number of double-precision floating-point operations for a kernel remains relatively the same, yet there is a significant variation in the memory profile, with the total number of cache accesses differing between the variants.

In Variant B, kernels *Madvx*, *Madvy* and *Madvz* all show a significant reduction in the number of cache accesses, measured as the sum of the PAPI L1 cache

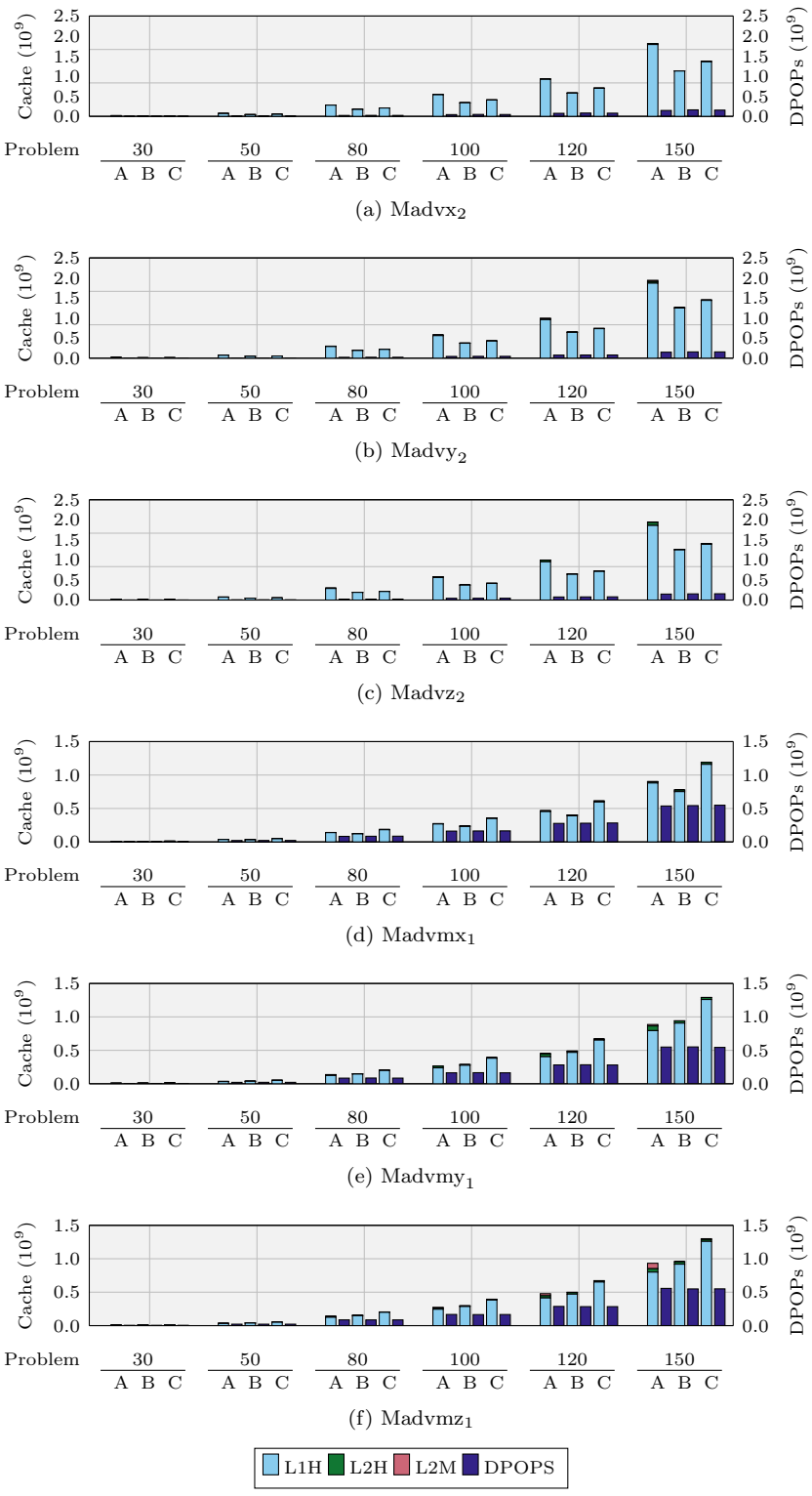


Figure 6.3: DPOPs, Cache Accesses — Hydra Variants A/B/C, Serial, Minerva)

hits and misses registered. The three remaining kernels remain relatively consistent, bar *Madvmx* which demonstrates a moderate improvement. From this it might be expected that an improvement in the performance of the kernel would be observed, given the reduction in memory overheads for the same fixed number of compute operations. For Variant C, a similar reduction in the number of cache accesses is shown for kernels *Madvx*, *Madvy* and *Madvz*, but the number of cache accesses for kernels *Madvmx*, *Madvmy* and *Madvmz* show a sizeable increase, a characteristic that might be expected given the increase in memory storage and transfers/copies caused by the use of multiple 3D-loops/arrays. It is possible that improvements in memory access patterns could be offset by such an increase in the number of total accesses. To clarify this, it is important to distinguish between the different types of cache accesses, that is to say the number of *hits* and *misses*.

Not all cache accesses are equal, and any hits to main memory or the L3 cache could easily overshadow any improvements in the number of cache accesses (or vice-versa). Figure 6.3 also presents the various overall PAPI counter frequencies. From this data, the memory access patterns of the kernels, while reasonably effective for the Madvx_2 and Madvmx_1 kernels, are shown to be poorer for the four remaining kernels (see Appendix A for the figure data in greater detail). Most notably, the Madvmz_1 kernel shows not only a sizeable number of L1 cache misses, but also a distinguishable portion of additional L2 misses, likely responsible for the poor performance when scaling the problem size observed in Section 4.5.1. Even though the absolute number of L1 and L2 Misses can be small in comparison to the total number of L1 Accesses, they prove to have a significant impact on performance nonetheless.

The two new variants are both successful in reducing these cache misses, and the impact of this is apparent in Figure 6.3. As might be expected, the reduction in both cache access rate and cache miss rate has resulted in an improvement for all six kernels in Variant B over the original Variant A. Variant C has a more mixed result. For kernels Madvx_2, Madvy_2 and Madvz_2, likely due to the

reduction in the number of total cache accesses, the performance of the kernels has improved when contrasted against Variant A. However, among the remaining three kernels only the Madvmz_1 kernel has a notable improvement, with the other two kernels exhibiting a decline in performance. For Madvmx_1 there is no reduction in the cache miss rate, meaning the increased total number of cache accesses is a performance penalty with no chance at any improvements in other areas, subverting attempts to improve performance. In the case of Madvmy_1, gains are made in the cache access miss rate, yet it appears to be insufficient to overcome the costs of more total cache accesses. Only for the Madvmz_1 kernel is any improvement observed, likely due to the highly significant L2 miss rate that is largely eliminated in the new variant.

In Figure 6.4 it can clearly be seen that, of the three variants, Variant B clearly demonstrates itself as the best performing, providing a significant improvement over the original Variant A. Due to the *Madv* kernel (which inclusively contains the kernels optimised here) dominating approximately 60-70% of the overall walltime, a significant impact can also be seen upon the overall walltime for the serial execution, as shown in Figure 6.4. Due to these improvements only affecting compute behaviour, they have a more limited impact upon parallel executions — especially strong-scaled execution where the impact of cache behaviour is minimised by smaller workloads per core as it is scaled. Nevertheless, the optimisation still offers some scope for improvement with weak-scaled executions where the workload per core is consistent and thus consistent improvements are present across all process counts. Variant C in contrast does not offer as significant an improvement as Variant B. Its advantage lies in the removal of a number of data-dependancies, enabling and improving the ease with which alternate optimisations can be explored in the future while still incorporating more optimal data access patterns. The improvement of the miss-rate over Variant A is still substantial, with a corresponding improvement in walltime. In subsequent experiments Variant C is used as the basis for further optimisation efforts, having both the fewest data-dependancies while also

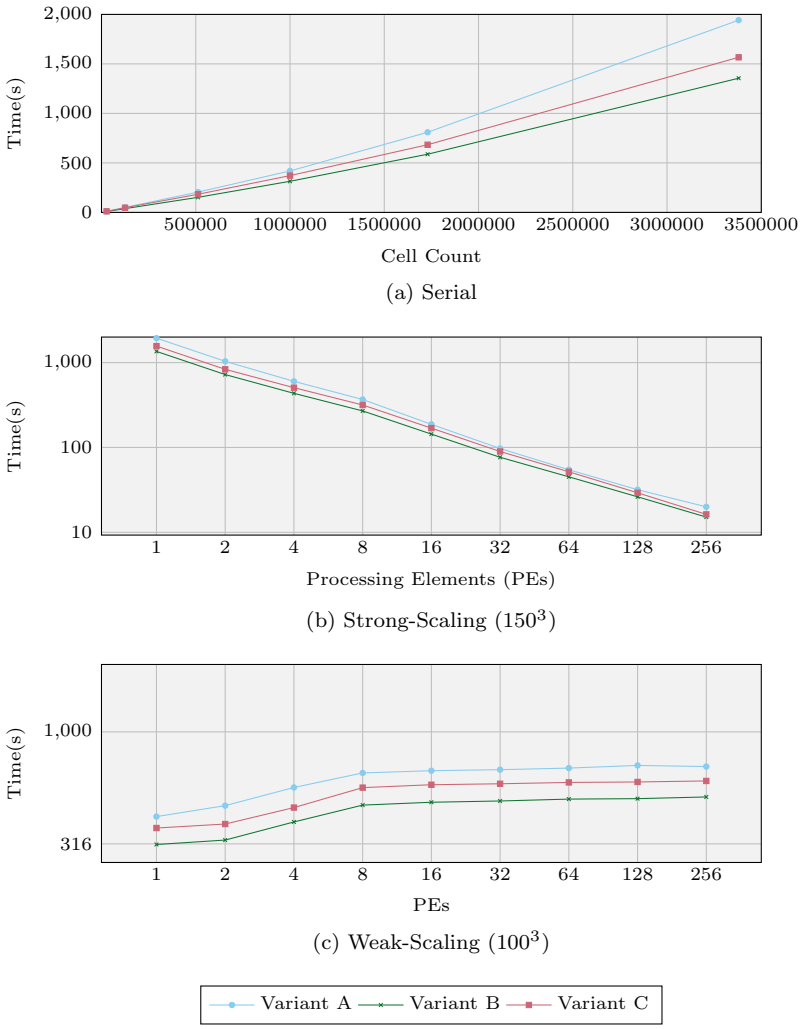


Figure 6.4: Memory Optimisation - Variant Total Walltimes

demonstrating a reasonable improvement in walltime over our base Hydra implementation. In the following sections this permits the continuation of these improvements while also enabling further optimisation opportunities.

6.3 Compute Optimisation

The vast majority of modern CPUs now typically come with some form of vectorisation support. Applying SIMD techniques enables the execution of a single operation across multiple elements of data using only a single instruction; the number of elements that can be operated upon simultaneously is dependant upon the size of the vector registers available in the hardware. For example, with a width of 32 bytes a Streaming SIMD Extensions (SSE) vector instruction can operate on up to four single-precision floating-point datapoints (assuming four bytes each), or two double-precision floating-point datapoints (assuming eight bytes each). The Advanced Vector Instructions (AVX) instruction set extends this further using a maximum vector register width of 64 bytes, effectively doubling the number of data-points it can process in a single instruction.

To make effective use of all a machines resources, code must be engineered in such a way that it can take advantage of such instructions, else it immediately places its potential peak at only a fraction of its theoretical maximum achievable FLOP/s. To do this, achieving the following requirements is neccessary:

- Eliminating inter-loop dependancies — An operation cannot be performed if it is still dependant upon the completion of an as yet uncompleted operation.
- Targeting portions of the code that repeat the same operation across large blocks of data - common in many scientific applications, this results in such techniques being amenable to embarrassingly parallel problems.
- The overhead of loading data into vector registers is not substantially higher than that of the vector operations — i.e., ideally reuse data in vector

registers where possible and target compute-bound rather than memory-bound code.

Of the various kernels, the majority exhibit a relatively consistent FLOP/s count. The exceptions are the Madvmy_1 and Madvmz_1 kernels, which exhibit variability at higher cell counts. This is consistent with the knowledge of the memory behaviour of their original implementation from Section 6.2.2, where the increasingly poor cache hit rate inhibits the overall performance of the kernel. For the remaining kernels, given the consistent nature of both the FLOP/s rate, the number of double-precision floating-point operations per cell and the cache miss rate established earlier in this work, it is unsurprising to see a steady walltime performance for these kernels, which is what enables the predictive power of the performance models.

Given the variation in FLOP/s between the various kernels, the typical trend appears to be that the better the machine work balance (DPOPs:Memory Access), the better its overall performance. The one exception to this is the *Update Velocity* kernel, which appears to have the best overall FLOP/s rate yet sits at approximately only $\approx 66\%$ of the next closest perfoming kernel in terms of the amount of work it performs per cell. Since all prior experiments were conducted with vectorisation disabled, it can be established that such variation is not due to some kernels being vectorised while others are not. Even without identifying the machine's overall peak, the existence of such differences in the FLOP/s rate is sufficient to suggest that, in terms of theoretical compute peak performance, some kernels are underperforming for reasons other than raw compute capacity.

This leads to two critical outcomes:

- If the overall performance is bottlenecked by a factor more dominant that compute performance, then the gains from vectorisation will also be bound by such factors, diminishing the overall improvements on offer.
- If this is the case, it would be prudent to identify those kernels most suited to vectorization and focus efforts upon these kernels. If possible,

this should be quantifiable.

The remainder of this section will attempt to confirm these two statements, vectorising a number of the key kernels using Intel SSE. Section 6.1 has already established evidence of memory bound behaviours exhibited by kernels such as Madvx_2 and others. However from Figure 6.1 prior it is also apparent that there exist kernels with DPOPS:Cache Access ratios weighted heavily towards compute — it is these kernels that constitute the main focus of vectorisation investigations.

6.3.1 Results

The overall implementation of SSE/AVX can be achieved in one of three ways:

- Automatic vectorisation by the compiler.
- Manual vectorisation via the use of compiler intrinsics.
- Manual vectorisation via the use of assembly instructions.

Of these three, automatic vectorisation is the simplest, minimising the complexity of implementation. It does however offer the least control of the process. Its opposite counterpart, assembly-based implementation, offers the most control but is significantly more complex and less portable. Compiler intrinsics offers a middle-ground between the two, offering a fair degree of control but still requiring implementation by the developer at a level higher than assembly. To explore vectorisation within Hydra, two new variants are implemented for a subset of the kernels present in Hydra, Variant D and Variant E. Variant D acts as a control case, porting these kernels to C but remaining unvectorised to identify whether the act of porting has modified the kernel charcteristics. Variant E provides a vector intrinsic implemention that uses SSE.

Before an investigation into the effective speedup on offer can be conducted, it is neccessary to establish whether vectorisation is possible for the compute available within Hydra. For an ideal SSE implementation in a double-precision

Cells	$\mathbf{MDT_1}$	$\mathbf{MDT_2}$	UpdVel	$\mathbf{Lartvis_1}$	Mdivu	Mvolflx	$\mathbf{Madvmx_1}$
30^3	0.47	0.50	0.48	0.50	0.50	0.49	0.40
50^3	0.47	0.50	0.49	0.50	0.50	0.50	0.41
80^3	0.47	0.50	0.49	0.50	0.50	0.50	0.42
100^3	0.47	0.50	0.49	0.50	0.50	0.50	0.42
120^3	0.47	0.50	0.50	0.50	0.50	0.50	0.42
150^3	0.47	0.50	0.50	0.50	0.50	0.50	0.42

Table 6.4: Minerva, Hydra Serial, Variant E SSE — VECOP:Total DPOP Ratio

code it would be expected that the number of vector instructions executed be approximately half that of the unvectorised instruction count (for the same number of DPOPs). The kernels selected are those that demonstrated some degree of potential compute-bound behaviour — kernels such as $Madvx_2$, $Madvy_2$ and $Madvz_2$ were omitted due to their apparent memory-bound nature from Section 6.2. In Table 6.4 the kernels are verified as successfully vectorised by comparing the ratio of PAPI measured DPOPs to the number of PAPI measured VECOPs. On Minerva a single VECOP represents a singular vector instruction, rather than the equivalent number of DPOPs, hence under SSE a single VECOP should be equivalent to two DPOPs, with an expected ratio of 0.5 VECOPs for each DPOP.

In an ideal scenario, the speedup of these kernels should match the increased throughput of DPOPs — i.e. a speedup of $2\times$ for twice the throughput — Figure 6.5 contrasts the speedup of Variant E, the vectorised C-language port, against both Variant C, the data-parallel variant, and Variant D, the un-vectorised C-language port.

The results of Figure 6.5 show that a number of the selected kernels demonstrate a reasonable improvement. However, a selection exhibit an equivalent or poorer performance, contrary to what might be expected from the DPOP:Cache access Ratio of Figure 6.1 earlier — in particular both Mdivu and Mvolflx see a slowdown, mainly because any speedup due to vectorisation is being offset by a more significant increase in walltime from the transition to a C rather than Fortran implementation. To explore this further, Table 6.5 shows a constrast of select PAPI statistics between the three variants for a fixed 100^3 problem

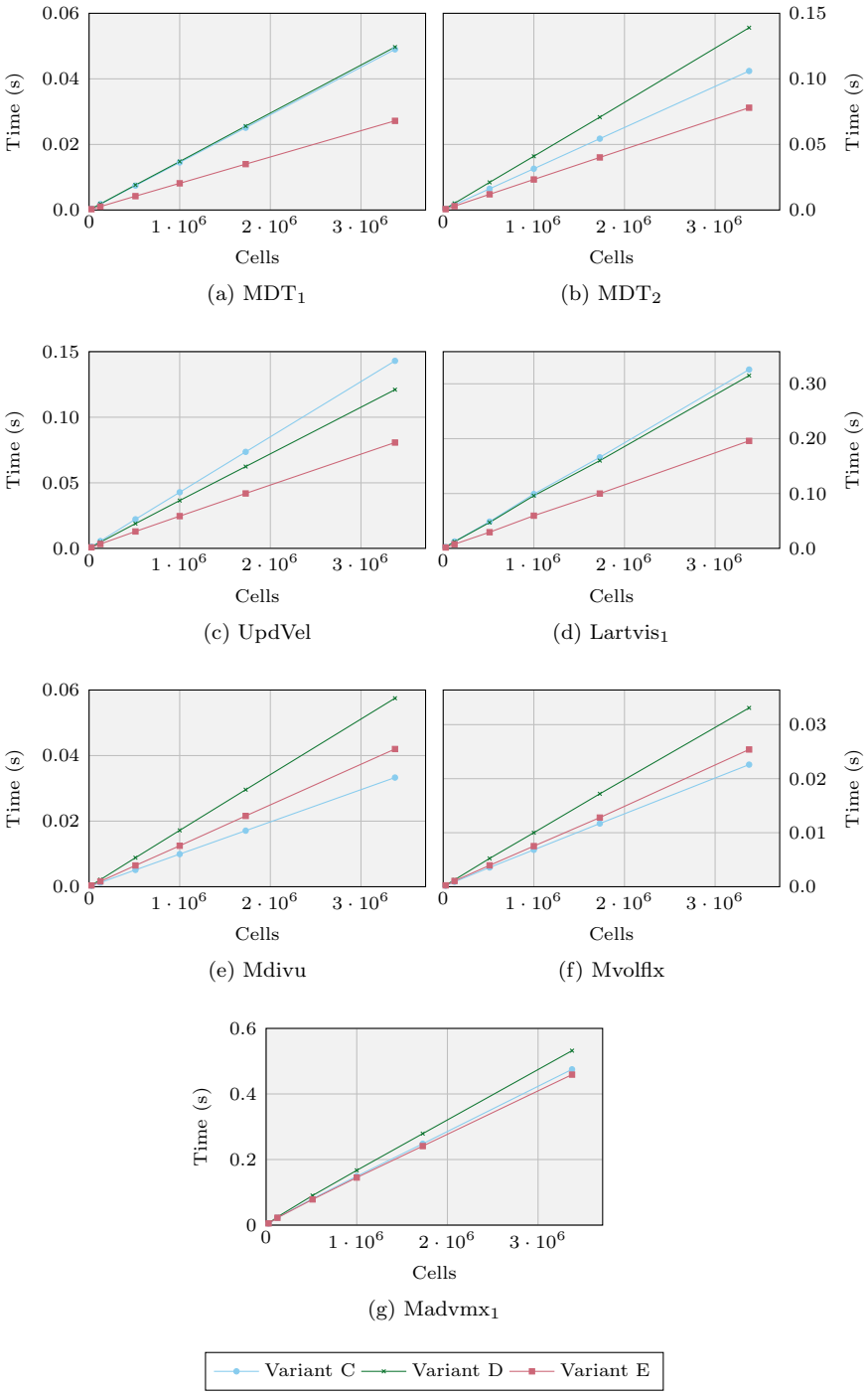


Figure 6.5: Hydra Variants C, Dand E— Kernel Walltimes

size on a small 2.4 GHz Intel X3430 workstation. Comparing the two machines, both demonstrate similar patterns for the various kernels. It can be seen that for those that underperform, once again, a speedup for the vectorised kernel is observed, but due to the increase in cost for Variant D, this improvement is at best offset or at worst insufficient to overcome the penalty.

For Mdivu, it can be observed that while the number of DPOPs remains relatively consistent, there is a sizeable increase in the number of L1 cache accesses, implying that the majority of the additional overhead is attributable to poor memory access behaviour. Mdivu is a stencil kernel that can touch the same memory locations multiple times (potentially with a reasonable amount of time between accesses). It is also possible that such accesses are unaligned due to stencil kernels using memory locations that are often offset from a fixed address (e.g. + or − 1 from a cell position).

In the case of Mvolflx and Madvmx_1, both the L1 Cache Accesses and DPOPs count are relatively consistent. However, in the case of the vectorised variant, it should be noted that while the total number of DPOPs is consistent, the actual number of instructions required should be roughly half (due to two double precision operations per an SSE vector operation), resulting in the number of vectorised instructions being roughly half that of DPOPs, as was the case for Minerva. A corresponding reduction in the number of cache accesses might also be expected — kernels such as MDT_1, MDT_2 and Lartvis_1 all see a significant reduction in the number of L1 cache accesses for the vectorised implementation, yet no such decline is present for the underperforming kernels. MDT_1 proves to be a curious exception here. While an increase in the number of DPOPs and L1 Cache Accesses is observed in the transition from Variant C to Variant D, this appears to be offset by a reduction in the number of stalled cycles, with the final implementation of Variant E seeing a sizeable speedup due to the reduction in instructions and, by extension, cycles overcoming this increase in cost.

MDT$_1$						
Variant	**Cycles**	**Stalled Cycles**	**Stalled %**	**Time (s)**	**L1 Accesses**	**DPOPs**
C	4.43E+7	3.25E+7	73.24	1.87E-2	1.64E+7	1.15E+7
D	4.52E+7	2.36E+7	52.27	1.91E-2	2.55E+7	1.31E+7
E	2.46E+7	1.01E+7	41.15	1.05E-2	7.64E+6	1.72E+7

MDT$_2$						
Variant	**Cycles**	**Stalled Cycles**	**Stalled %**	**Time (s)**	**L1 Accesses**	**DPOPs**
C	9.67E+7	5.60E+7	57.94	4.07E-2	6.07E+7	4.39E+7
D	1.25E+8	5.61E+7	44.86	5.27E-2	1.05E+8	4.32E+7
E	7.06E+7	3.87E+7	54.81	2.98E-2	4.53E+7	4.43E+7

Lartvis$_1$						
Variant	**Cycles**	**Stalled Cycles**	**Stalled %**	**Time (s)**	**L1 Accesses**	**DPOPs**
C	3.02E+8	1.94E+8	64.44	1.27E-1	1.31E+8	1.46E+8
D	2.93E+8	1.80E+8	61.41	1.23E-1	1.09E+8	1.44E+8
E	1.82E+8	1.06E+8	57.96	7.68E-2	7.00E+7	1.66E+8

Mdivu						
Variant	**Cycles**	**Stalled Cycles**	**Stalled %**	**Time (s)**	**L1 Accesses**	**DPOPs**
C	3.01E+7	1.95E+6	6.46	1.28E-2	2.89E+7	2.85E+7
D	5.31E+7	2.36E+7	44.47	2.25E-2	4.08E+7	2.92E+7
E	3.91E+7	1.11E+7	28.29	1.66E-2	3.37E+7	2.99E+7

UpdVel						
Variant	**Cycles**	**Stalled Cycles**	**Stalled %**	**Time (s)**	**L1 Accesses**	**DPOPs**
C	1.39E+8	6.46E+7	46.39	5.86E-2	1.22E+8	8.49E+7
D	1.23E+8	5.30E+7	42.92	5.20E-2	1.00E+8	8.00E+7
E	8.03E+7	3.31E+7	41.19	3.40E-2	6.32E+7	8.24E+7

Mvolflx						
Variant	**Cycles**	**Stalled Cycles**	**Stalled %**	**Time (s)**	**L1 Accesses**	**DPOPs**
C	2.10E+7	5.96E+6	28.39	8.91E-3	2.00E+7	1.62E+7
D	3.03E+7	3.06E+6	10.10	1.29E-2	1.81E+7	1.62E+7
E	2.27E+7	3.43E+6	15.14	9.68E-3	1.87E+7	1.68E+7

Madvmx$_1$						
Variant	**Cycles**	**Stalled Cycles**	**Stalled %**	**Time (s)**	**L1 Accesses**	**DPOPs**
C	4.08E+8	1.74E+8	42.81	2.00E-1	3.59E+8	1.66E+8
D	5.11E+8	2.11E+8	41.31	2.44E-1	3.60E+8	1.69E+8
E	4.20E+8	1.68E+8	39.93	2.05E-1	3.23E+8	1.91E+8

Table 6.5: Hydra 100^3, Serial, PAPI Statistics — Intel X3430

The implication from these behaviours is that the memory access patterns are inhibiting efforts to vectorise these kernels — their stencil nature causing either slower unaligned accesses or poor cache reuse from multiple vector loads touching similar memory locations, despite the previously good ratio of DPOPs:Cache Access ratio of some of these kernels prior. As such, it might be expected that the most dominant kernels in Hydra, such as Madvz_2 or Madvmx_1 would likely not benefit from vectorisation, given their similar properties to Madvmx_1 and their prior evidence of being memory-bound. Work such as the approach taken by Henretty[73] may help to alleviate this problem but is not explored within this Thesis, leaving it for a future exercise, due to the complexity of applying data-layout transformations to a larger code-base, especially one that operates across three dimensions

This leads to the conclusion that vectorisation without addressing these memory issues has little to offer in terms of optimisation for the Hydra benchmark, since without improving those kernels that dominate ≈ 60 of the compute time, there is a significant limit on the speedup that can be achieved as per Amdahl's Law.

6.4 Compute-Communication Overlap

The overall performance of a parallel code intra-node is dictated by the compute and memory performance of the worst performing PE and the distribution of its workload. In a parallel environment the overall impact of contention upon a single-node performance can be mitigated by sharing the workload further across more distributed PEs; the addition of further nodes does not contend with resources on the existing node, but does reduce the workload per individual node, lessening the impact of on-node bottlenecks somewhat. The trade-off for this is the introduction of a new bottleneck, that of the network.

The communication costs of such scaling are an inherent and unavoidable part of all but the most embarrassingly parallel of problems — the existence of

any form of data-dependancy on a neighbouring element in a structured problem will necessitate the retrieval of data from a remote source. Such dependancies exist within Hydra due to the use of stencil compute kernels that require data from surrounding cells in all three dimensions, resulting in the near-neighbour exchange steps described in Section 4.3.2. Minimising these overheads is of interest for not only our problem benchmark, but all similar scientific applications (such as those used in Chapter 7).

A straightforward approach to optimising network costs is to reduce or eliminate unneccesary data communication. However, in the absence of an alternate algorithm, such an approach depends upon the initial implementation being sub-optimal. Although worthwhile examining, it cannot be assumed in the general case to be a viable optimisation (though should constitute part of a standard code review process).

An alternate solution is to minimise the impact of latency costs when using MPI messaging. Each message sent incurs a latency cost involved with its delivery, regardless of its size. Thus the higher the frequency of communication, the greater the impact of a network's latency cost. By merging multiple small, frequent messages into larger, more infrequent messages the latency cost can be kept to a minimum while ensuring that the overall amount of data communicated remains the same. However. such an optimisation already exists within Hydra, relying on five separate "stages" to communicate data for a number of different quantities in large messages, the minimum possible due to a data-flow dependancy between the completion of specific compute before the communiation of its result. In addition, our analysis of Hydra's network performance thus far has revealed that it is typically dominated by bandwidth, not latency constraints, due to the size of its messages, thus the opportunities for optimisation here remain minimal.

Instead, the approach of overlapping communications and compute simultaneously is explored. This addresses the issue where a greater efficiency can be achieved within the system as a whole by minimising the amount of idle time

spent by one component, the compute, during the use of other components such as the network. No such feature is currently exploited within Hydra, but in theory such an optimisation is capable of masking either the communication overheads or compute overheads (whichever is the smaller of the two). This optimisation is reliant on a number of contributing factors, but provides opportunities for mitigating the cost of one or more idle CPUs waiting on MPI communications, resulting in an overall speedup at scale. With some re-engineering of how Hydra processes its communication and compute steps, the potential exists for the use of communication overlap as part of its approach. Understanding the factors that influence the effectiveness of this approach thus becomes crucial to ensuring its success.

6.4.1 Implementation

A number of works have explored the use of computation/communication overlap, addressing issues such as assessing the potential gains of MPI overlap [78, 97, 143, 153, 162, 170], independent progress in non-blocking communications [30], the impact of network hardware [149] or the use of Hybrid approaches that allow for separate communication and computation threads [147, 171]. In pursuit of these approaches, using the knowledge obtained from Chapter 4 on the data-dependancy patterns, it is possible to identify large chunks of work that are independent of MPI communications in Hydra. Cells identified as internal boundary cells, due to their dependancy upon ghost cells refreshed from other processes, require the completion of MPI communication before they can be updated as part of any computation. However, non-boundary cells have no such dependancy; they can be updated purely on data locally resident to the current process — overlap can thus be achieved by separating out the dependant and independant compute, overlapping MPI messages and only updating internal boundary cells once any relevant communication is complete.

In Variant A of Hydra, there are two factors that interfere with this approach. First, while non-blocking communications are used during Hydra's point-to-

point communication steps, the transmission of messages in the Y-dimension cannot be initiated before the completion of messages sent and received in the X-dimension due to the use of received ghost cells in the construction of the next message. The same restriction exists between Z-dimensional messages and their dependence on the Y-dimension. Second, due to the order cells are processed in some kernels, most notably the advection kernels $Madvx_2$, $Madvy_2$, $Madvz_2$, $Madvmx_1$, $Madvmy_1$ and $Madvmz_1$, there can exist inter-loop compute dependancies which require the processing of boundary cells before the internal cells can be processed. To address these two issues, Hydra is re-engineered to introduce both a separation of dependent and independent compute, while also eliminating the dependancy of the communication steps.

For the communication step, the existence of the dependency is to minimise the number of overall messages that must be communicated. Previously in Section 4.3.2 the interdependency of the MPI messages was addressed; there exist up to 26 data-dependent neighbours, yet the overall number of messages is reduced to a maximum of 6. In doing so, the latency impact of having a higher message count is reduced. It is apparent that the elimination of the strict order of communications can be achieved by implementing the reverse, that is to say returning to a state where a process communicates directly with its diagonal neighbours as well as its face-sharing neighbours. This has the potential to reintroduce an additional synchronisation or latency cost to the overall communication stage, but also enables the full overlap of communication and computation which, if effective, may mask any such increases in the communication overheads. Given the dominant influence of the bandwidth on network costs, the cost of additional latency overheads should be minimal.

For the inter-loop dependencies, the introduction of a "data-parallel" variant, Variant C, in Section 6.2 not only implemented some of the memory improvements but also removed the intra-kernel dependencies between multiple inner loops that enforced a strict cell processing order. Rather than a large 3D loop with multiple inner loops, the transition to multiple 3D loops allows

Listing 6.6: Variant G

```
1 Independant Compute
2 Exchange Stage
3 Dependant Compute
```

Listing 6.7: Variant H

```
1 Pack/ISend/Irecv
2 Independant Compute
3 Unpack/MPIWaitall
4 Dependant Compute
```

the seperation of compute in such a way as to separate between communication dependent and communication independent compute. Removing these dependencies allows us to persue two alternate approaches to overlap:

1. Explore the use of non-blocking communications in a context where overlapping can theoretically occur if the hardware and/or software is capable of supporting such a capability.

2. Implement a Hybrid MPI/OpenMP approach, where a dedicated thread handles communications while the remaining threads progress with compute.

The former offers a greater potential for gains but is more restrictive in its requisites for success, while the latter guarantees communication overlap will occur but at the cost of sacrificing a thread/core that could potentially overshadow any gains to be had.

MPI Non-Blocking Overlap

For the first approach, relying on the non-blocking overlap of the underlying MPI implementation, there are four variants for exploration.

Variant C

Already introduced in Section 6.2, this variant provides the removal of various data dependencies that enable further alteration of the code base for

exploring overlap. It is included here as a performance baseline.

Variant F

This variant, based on Variant C, removes the data-dependency that exists between the messages sent in the different dimensions during the MPI communication stage (as detailed originally in Section 4.3.2). This is achieved by enabling direct communication with diagonal neighbours in one or more dimensions, removing the need to communicate ghost data as part of any Y or Z dimension communication; this results in a maximum of 26 neighbours/message send-recv pairs per communication phase. No overlap yet occurs, but this action removes a potential block where, for example, the Y-dimension processes are ready for communication but the X-dimension processes are not. This control variant is intended to capture any performance changes that could be attributed to modifying the communication pattern (such as, for example, an increased latency or overhead cost).

Variant G (Figure 6.6)

Further extending Variant F, this variant modifies the order in which cells are processed, such that internal, non-boundary cells can be processed as independent compute distinct from those cells that are dependent upon the completion of the communication stage. This is crucial since only independent compute can be overlapped without violating the correctness of the program. Since this modifies the memory access patterns of the program, it is neccessary to draw a distinction between this and subsequent versions to capture any change in performance.

Variant H (Figure 6.7)

This variant extends Variant G to introduce the final change required for

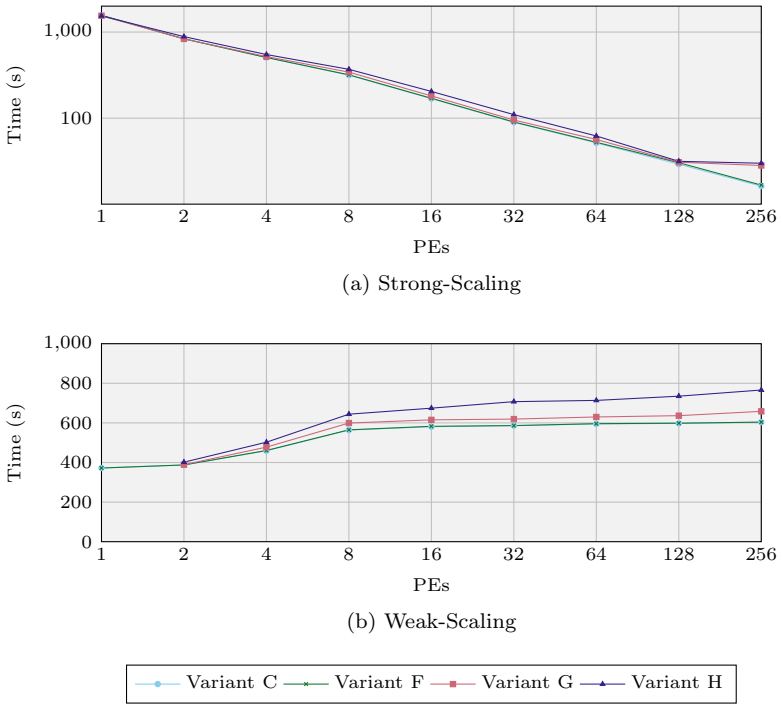
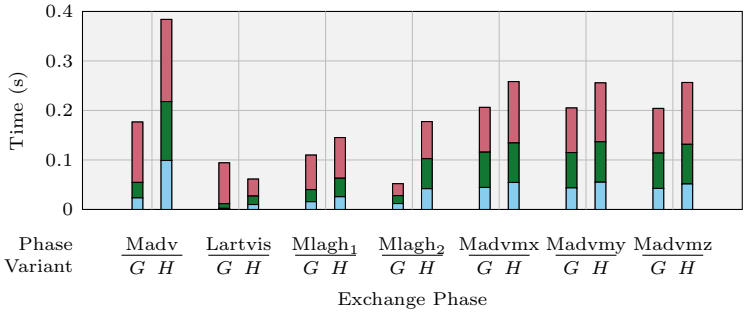


Figure 6.6: MPI Overlap Performance — Non-Blocking Variants

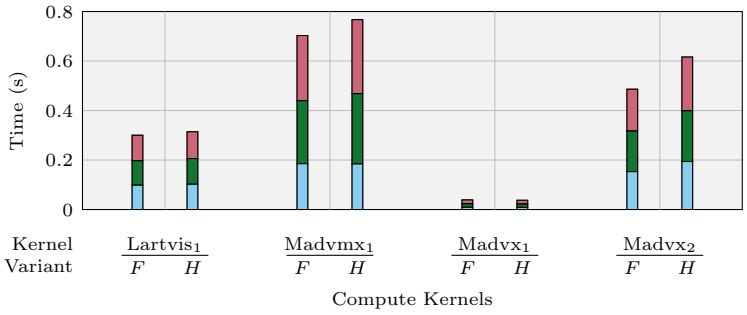
enabling an opportunity for MPI overlap. The original blocking commands to check for communications completion are shifted to immediatedly prior to the dependent compute stage, with this block preceeded by both the beginning of the communication stage and any independent compute. In a fully functional overlapping code, this independent compute will be overlapped with any MPI communications, masking the lesser cost of the two.

Results

From the overall walltimes it can be seen that the impact of removing communication dependencies is negligable (Variant F), as might be expected given the overall amount of data communicated is effectively the same. However, the separation of dependent and non-dependent compute for Variant G in relation to the communication stage does experience an overall increase in the walltime.



(a) Communication Stages - Variants G vs H



(b) Compute Kernels - Variants F vs H

Figure 6.7: Non-Blocking Madv Behaviour — Minerva 100^3 Weak-Scaling, 256 PEs

This increase is due to an extra cost attributed to the compute component of Hydra, and is likely a result of slower memory access patterns — the majority of dependent compute that is separated out acts upon the boundaries of the local grid, requiring multiple accesses to non-contiguous memory. This introduces an extra overhead to the cost of overlapping compute and communications — if this overhead is less than the potential savings then it becomes unworthwhile to proceed.

The final variation that modifies the order of operations and enables overlapping via the use of non-blocking communications proves to be somewhat surprising — an increase rather than a decrease is seen in the overall walltime, a cost that is ascribed mostly to an effective tripling of the communication costs

of the *Madv* function amongst other communication increases. In Figure 6.7 a breakdown of the Madv compute/communication balance between Variants G and H for the weak-scaled, 256 PEs experiment highlights this behaviour. Either no effective communication overlap is occuring, or the removal of the barrier causes some form of communication imbalance or other cost that prevents effective overlap, such as one process performing compute while another waits idle to communicate with it. Given the non-blocking behaviour is left to the implementation of MPI, an alternate approach is required to enforce a more strict interpretation of compute/communication overlap — allocating dedicated resources through the use of threads.

6.4.2 OpenMP Threaded Hydra

To enable the use of threads for the purposes of overlapping communications, a threaded version of Hydra is neccessary to enable the distribution of compute and communication work among the threads. Two different variations are implemented and contrasted here to provide both a baseline for comparison and to establish the impact of threading upon the performance of Hydra.

Variant I

The initial threading variant uses OpenMP static scheduling in combination with 2D collapsed loops to reduce each block of compute to that of an innermost loop. Threading is implementated at a kernel level, entering and exiting a new parallel block upon entrance and exit of each major kernel.

Variant J

Built upon Variant I, but using dynamic as oppossed to static scheduling, this variant is used as a baseline for any threaded overlap. The dynamically threaded approach is explored here since it permits the communications threaded to fall back into compute work upon the completion of any communications, as oppossed to the fixed $n-1$ threads allocated to compute for

a static approach.

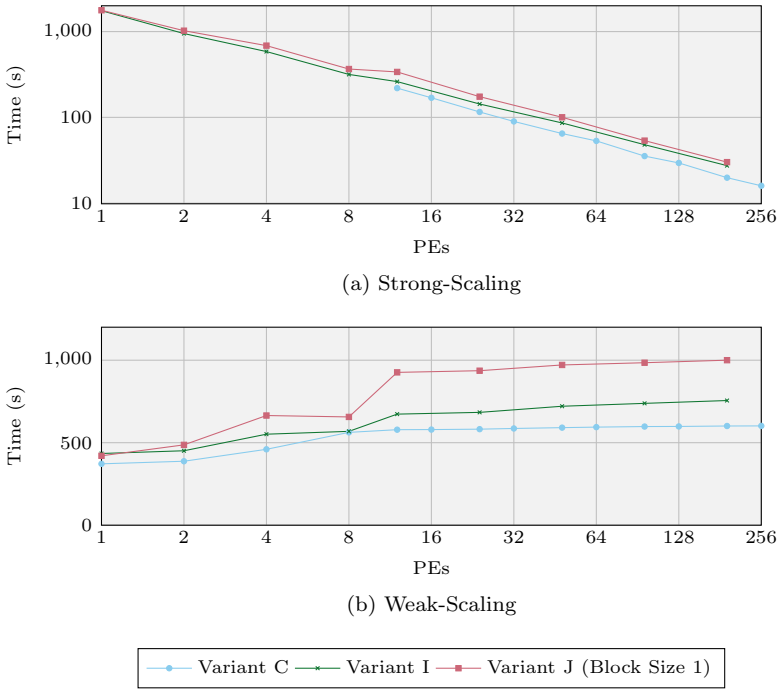


Figure 6.8: MPI Overlap Performance — OpenMP Variants

From the walltimes it can be seen that the use of a static threading schedule has introduced some additional overhead to an execution of Hydra. These overheads can be ascribed to two factors.

First the memory allocation/deallocation functions are assigned workloads tied to the number of processes, not threads. Due to the zeroing of these memory locations, a lack of threading on these shared memory block allocations leads to an effective increase in their cost over the non-threaded variants. This factor is a mere oversight that can be fixed with appropriate threading of the zeroing process.

Second, the use of OpenMP does see an increase in the compute costs of select kernels. Since the use of threads is implemented through the use of OpenMP pragmas, such as parallel for, the kernels themselves are largely un-

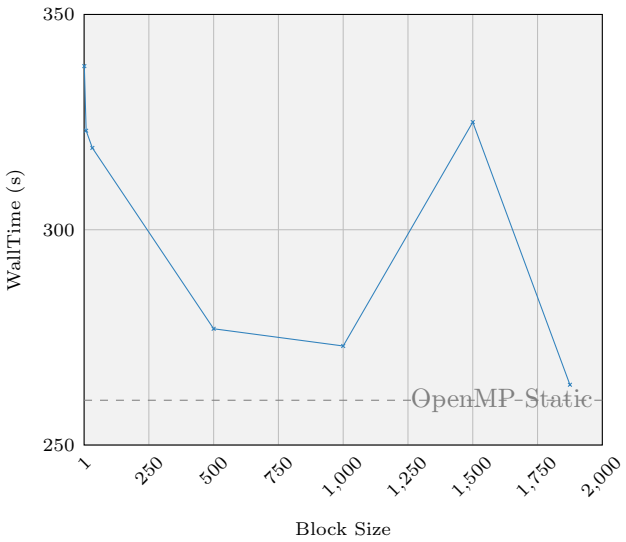


Figure 6.9: OpenMP Dynamic Schedule (Variant J), 150^3, 12 Threads

touched from previous variants, suggesting that that some impact of introducing OpenMP, such as threading overhead or some unforeseen interaction rather than the threading configuration is at fault (each thread was validated to be bound to a unique physical core). This would imply that some component of the thread-parallel process is responsible for the increase in cost, such as either threading overheads or memory access patterns. This may be in part due to the use of collapsed 2D-loops.

The use of a dynamic schedule exacerbates this problem — the significant increase in walltime over static scheduling is almost entirely attributable to an increase in the compute costs of Hydra. This appears to be a factor of selecting a suitable block size for the dynamic approach. In Figure 6.9 the block size is varied across a range of values, approaching that of the block sizes selected by a static approach that distributes the entire workload evenly with minimal allocations.

For the investigation of the dynamic scheduling performance, 2 MPI processes with 6 threads per process are used for a $150 \times 150 \times 150$ problem size. The 2D collapsed dynamic loop results in a work allocation of approximately $((150 \times 75)/blocksize)$ work blocks per MPI process, with each work block pro-

cessing a single inner loop iteration of roughly 150 cells (depending on the kernel). This results in an approximate maximum number of 11250 blocks of work if using the default block size of 1. When spread across 6 threads, this allocates 1875 blocks per thread (assuming an even work load per block). As a consequence, as the block size tends towards 1875, the number of times a block of work must be assigned to a thread tends towards 1, similar to that of the static work allocation.

In Figure 6.9 the impact of this block size upon the dynamic OpenMP variant is apparent. With the exception of a block size of 1500, the performance of the dynamic variant approaches that of the static implementation, suggesting that the principle cause of the disparity in walltimes of the original experiments was due to the very fine-grained work allocation, either due to OpenMP overhead or potential other factors such as cache performance. However the improvement is most significant at very small block sizes, as they get larger the potential gains are reduced. In scenerios where the use of dynamic work allocation offers a benefit to work-load balance, it may overcome this penalty using one of the larger block-sizes.

The OpenMP variants as a whole do not offer direct performance improvements over the initial data-parallel implementation of Hydra. However, these underperforming aspects appear to be primarily tied to the implementation/block size configuration issues, factors that can be refined using the existing code as a basis for improvement. In addition, while not initially offering a direct improvement, the use of dynamic work allocation permits us to explore other factors of interest — most notably that of compute/communications overlap. By freeing the MPI processes from fixed sequences of tasks (beyond that of enforcing data coherency), the use of threads allows the use of under-utilised resources such as CPU compute power during communication steps that are primarily bottlenecked by the interconnect. Such an optimisation could prove to eliminate a significant bottleneck created by the distributed nature of much HPC supercomputing hardware.

Listing 6.8: Variant L

```
1 Thread Creation
2     Master -> MPI Exchange Stage
3            -> Remaining Compute
4
5     Other Threads -> Compute Only
6 Thread Destruction
```

Figure 6.10: MPI Communication/Computation Overlap - Non-Blocking, Threaded Approach

6.4.3 MPI Threaded Overlap

Variant K

Extending Variant J, the compute kernels are separated into dependent and independent compute as per Variant F in the non-threaded version. As before, this is intended to capture the performance overhead of this change, but in a threaded context.

Variant L (Figure 6.8)

Extending Variant K, all handling of MPI communications is allocated to a master thread, while the remaining threads begin to process any independent compute. If any independent compute remains upon the completion of all outstanding MPI comms, the master thread moves onto this compute via the use of a dynamic schedule.

The disadvantage of a threaded overlap approach is primarily the removal of a resource that could be used for compute, by allocating a thread (and associated core) to primarily communications. In an environment with few threads, this leads to a substantial increase in the compute cost of the remaining threads, due to an increased workload — in an environment with only two threads, this would lead to a doubling of the compute time, likely mitigating any savings to be had from overlap (or even damaging overall performance). However, there are techniques to mitigate the impact of such a scenario.

In a multi-threaded environment, only one thread is required for communication management. While not explored here, the use of Hyper-threading may

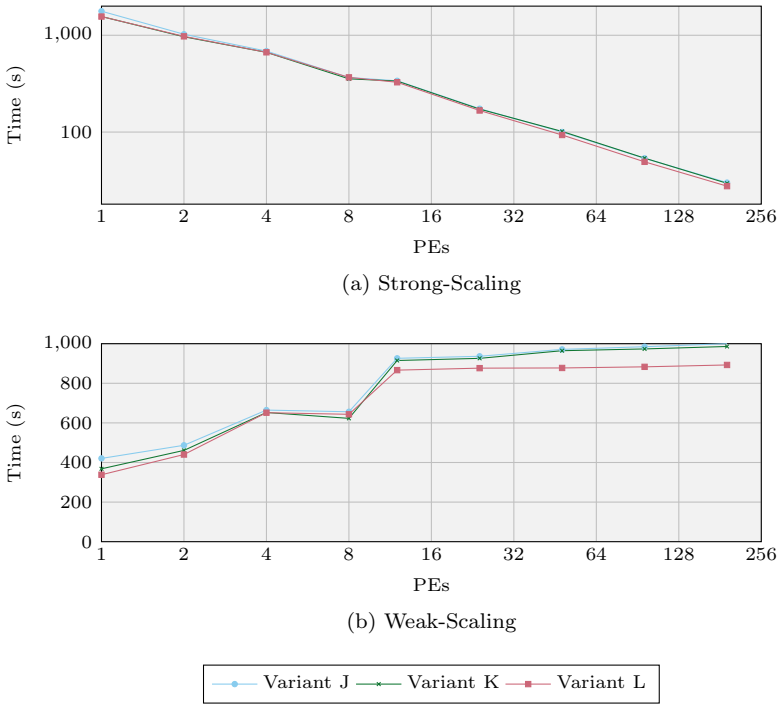
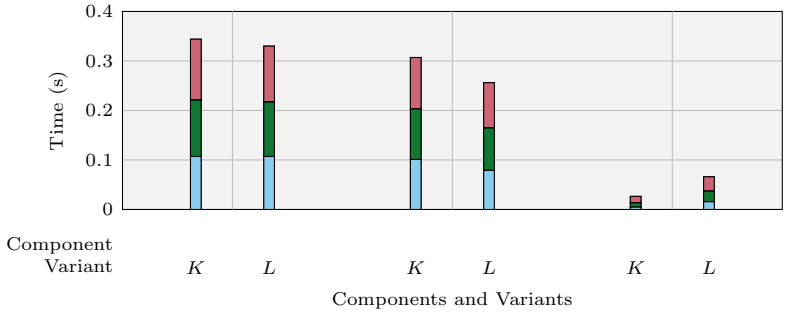


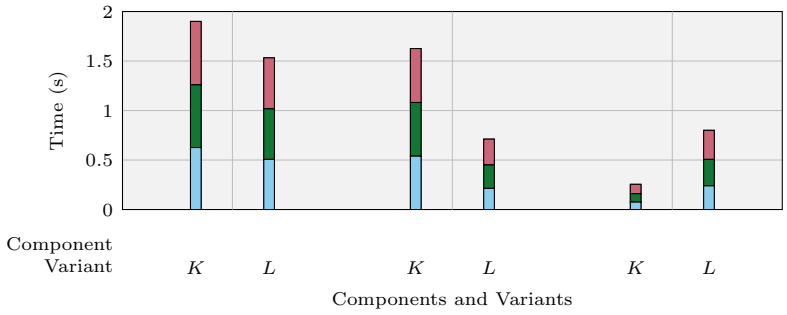
Figure 6.11: MPI Overlap Performance — Threaded Overlap Variants

enable the independant progression of communication without removing a core from the compute pool. In addition, the greater the pool of compute threads, the lesser the impact of a single threads removal, distributing its workload evenly across all remaining threads. In the Minerva experiments, where there are six threads per MPI process, it would be expected that the compute workload of the other threads would increase by no more than $1/5th$. As long as the total time for communication is less than this increase, it would suggest that overlap is worthwhile. Further to this, the use of the dynamic rather than static scheduler avoids the scenario where the communication thread sits idle after all MPI messages have been sent, resuming compute work. This limits the loss of a compute thread to only the time taken to perform communications, and should theoretically result in a scenario that is no worse than the non-overlap variant.

From the min/mean/max process walltimes in Figures 6.11(a) and 6.11(b), it can be seen that the re-ordering of operations to facilitate overlap has had



(a) Lartvis Compute and Communications — Variants K vs L



(b) Madvmx Compute and Communications — Variants K vs L

Figure 6.12: Communication and Compute Overlap — Madvmx and Lartvis

little impact upon the overall walltime. Further, in Variant L the introduction of overlap has a reasonable speedup over that of the non-overlap threaded variant. Examining closer, Figures 6.12(a) and 6.12(b) demonstrate two of the compute-communication stages, in the forms of functions Lartvis and Madvmx. The resulting times are those measured on the master thread of each process, the sole thread responsible for managing any MPI communications. As such, the measurement of any compute times on this thread is the compute that remains *after* the communications are complete - i.e. if overlap is occuring it should be expected that some compute has already occured, resulting in a smaller compute time. In an optimal scenario, this communication time plus any remaining compute time should be less than the non-overlap variant.

As can be seen, the impact upon the Lartvis function is minimal, seeing only a slight improvement in the overall function time. Given the minimal time spent in communication, it is likely that is little scope for overlap within the function. However, in a more substantial operation such as the Madvmx function, a reduction of almost a quarter the runtime can be seen.

An interesting outcome of these sub-components is that it can be seen that the assumption of a fixed communication time is invalid — in both cases the time spent in the communication stage increases for all three statistical metrics. This is likely due to an increased strain on the memory subsystem — both compute and communications accessing data simultaneously for different purposes. However, this extra communication time is still overlapped with the compute, at the cost of an increased period of time where the computer work pool is lacking the master thread. As long as this increase in compute time is worth the savings from communication overlap, the process is still worthwhile.

6.5 Node Core-Count

In the course of this chapter a number of instances have been identified where the impact of memory performance has become a primary bottleneck in the performance of various compute kernels. It is likely this is also the predominant cause of the difference in the Wg timings for Chapter 5, where the fewer active cores per socket, the better the kernel performance per loop iteration. The Hydra model enables the exploration of alternate scenarios, such as the impact of reducing the number of cores per socket in use upon the overall walltime. Table 6.6 presents the outcome of a weak-scaled 150^3 problem at 256 PEs for a variety of different core-counts per socket in use.

However this constitutes an unfair comparison. To maintain the same number of PEs with a reduced core count per socket requires an increased number of nodes over the original empirical experiments. This introduces additional resources to the overall system, including more memory/effective bandwidth,

	Cores Per Socket			
PEs	**Model(1)**	**Model(2)**	**Model(4)**	**Empirical(6)**
16	421.93	469.16	570.55	671.10
32	426.31	475.80	579.50	678.20
64	427.36	479.99	585.80	689.28
128	427.37	480.00	585.81	708.31
256	427.39	480.02	585.83	700.35

Table 6.6: Model Timings — Cores Per Socket, Minerva, Weak Scaled 150^3

and more network interconnects. While the resulting amount of inter-node communication is increased, the additional network hardware helps to alleviate this. However, performance is not the only concern when constructing a HPC system. Such models also enables the comparison of smaller-scale, lower performance machines such that the trade-off between a reduced cost and reduced computation can be calculated.

6.6 Summary

This chapter has demonstrated that a multitude of factors related to both software and hardware ultimately influence the overall performance of an application. A focus upon the memory patterns has demonstrated the impact of memory performance upon an HPC code, showing both how improving the use of cache and reducing memory access can improve the performance, while also demonstrating that the benefits of compute optimisation techniques can be restricted by these memory-bound kernels. Further, the use of computation/communciation overlap techniques can mask such costs, although the use of solely non-blocking MPI functions proves to be insufficient to rely on this approach, given its potential dependance upon the implementation or hardware support. While the overlap demonstrated here does not provide an overall speedup, this is due to an underlying performance problem in the threaded implementation (as can be seen from the control variants) rather than the overlap approach. The approach itself is demonstrably viable and future efforts to identify and migitate the threading overhead could result in a significant optimisation.

CHAPTER 7

Application to Linear Solvers

Previous chapters have focused upon the application of analytical modelling to a single application, Hydra. However, for a method or technique to be viable to the field at large, it must be demonstrated that other applications of interest are also amenable to the same processes. Portable, Extensible Toolkit for Scientific Computing (PETSc) is a software library that provides an API to either first or third-party linear solver solutions, and is designed for the use in other software applications across multiple domains. This chapter focuses upon the performance characteristics of Conjugate Gradient (CG), one such linear solver algorithm implemented within PETSc, demonstrating how it can not only be broken down in a similar manner to Hydra, but also how it shares similar performance characteristics in its communication patterns, despite being a distinct piece of software in its own right.

Specifically, this chapter addresses the following:

- The CG algorithm of PETSc is introduced, including its function breakdown;
- The function breakdown is further separated into the compute and communication components, and it is shown how the sum of various minimum and maximum contributors is equivalent to $\approx 1\%$ of the total runtime, similar to how a critical path was achieved for Hydra;
- Given one basis for modelling is to establish behaviour of codes at scale, a brief comparison is made between the original CG algorithm and a "coalesced" algorithm already implemented in PETSc that utilises fewer collective operations in exchange for an additional compute component. It

is shown that at the scale of 16384 Processing Elements (PEs) the contribution of the collective component appears to have minimal impact, yet the additional cost of compute in the coalesced algorithm is measureable in its impact.

7.1 Introduction to Linear Solvers

A linear solver is an algorithm that solves a system of linear equations to find vector x in the equation:

$$\mathbf{Ax} = \mathbf{b} \tag{7.1}$$

In the equation, A is the matrix of coefficients for all of the linear equations, x is the vector of the unknown components in the system of linear equations and b is the vector of the results of the right hand side of the equation. An example is provided in Figure 7.1.

$$2x_1 + 4x_2 + 10x_3 + 12x_4 = 126 \tag{7.2}$$

$$4x_1 + 1x_2 + 8x_3 + 5x_4 = 67 \tag{7.3}$$

$$3x_1 + 6x_2 + 4x_3 + 8x_4 = 97 \tag{7.4}$$

$$6x_1 + 3x_2 + 7x_3 + 13x_4 = 135 \tag{7.5}$$

$$5x_1 + 4x_2 + 9x_3 + 11x_4 = 126 \tag{7.6}$$

$$A = \begin{bmatrix} 2 & 4 & 10 & 12 \\ 4 & 1 & 8 & 5 \\ 3 & 6 & 4 & 8 \\ 6 & 3 & 7 & 13 \\ 5 & 4 & 9 & 11 \end{bmatrix} \quad x = \begin{bmatrix} x1 \\ x2 \\ x3 \\ x4 \end{bmatrix} \quad b = \begin{bmatrix} 126 \\ 67 \\ 97 \\ 135 \\ 126 \end{bmatrix}$$

Figure 7.1: Linear Solver Components

While simple in presentation, deriving the solution involves a significant amount of computation and data movement. This problem becomes even more pronounced when the matrix and vector are scaled up in size. In addition, the ma-

trix can have many properties such as diagonal symmetry or density which influence the nature of the problem and, by extension, the approach to be taken. For example, a dense matrix has many spatially local accesses in memory, whereas a sparse matrix has relatively few local memory accesses.

To take advantage of current High Performance Computing (HPC) systems, a linear solver approach must be scalable in order to achieve an appropriate degree of machine efficiency. As the HPC community focuses its efforts towards Exascale capable machines, this only becomes more critical. It is often the case that some characteristics with little impact at small scale (such as global collective overheads) emerge to become significant bottlenecks when greater numbers of processing elements are introduced. Understanding these computation and communication characteristics becomes of great importance, and exploring new approaches such as communication-avoiding algorithms will be significant in future strategies. This introduces additional considerations beyond the compute and memory overheads of any algorithm, as they must now ensure that data is distributed and communicated in an efficient manner. These communications overheads are obviously an impediment to achieving scalable behaviour, and must be kept to a minimum.

This need for a portable, correct and efficient parallel implementation of such algorithms has given rise to a number of third-party libraries targetted at the scientific community. Linear systems can form the crux of a multitude of scientific problems, including those used by the Atomic Weapons Establishment (AWE), and thus efforts have focused upon enabling an application developer to integrate a single, stable implementation across multiple applications rather than requiring a reimplementation everytime. This has the benefit of saving a significant amount of development time, and removing a potential avenue of error by passing the complex nature of a parallel implementation to a single, more maintainable, source. In doing so an application developer, whose particular scientific domain may be unrelated to linear solver algorithms, can instead rely on a library maintainer who is likely to be well versed in such algorithms (in

both correctness and efficiency).
Examples of such libraries include:

- The PETSc [15, 16, 17]
- The Parallel High Performance Preconditioners (HYPRE) library [53]
- The MultiLevel Preconditioning Package (ML) [63]
- The Sparse Object Oriented Linear Equations Solver (SPOOLES) library [129, 10]

However, there are dangers to using "blackbox" solutions. The library developer is entrusted with ensuring that the solution is found efficiently, restricting optimisation efforts for those not involved in the library's maintenance. This can prevent a true understanding of how the solver performs at scale, since it is difficult to make associations between how the library performs and the underlying hardware. Such libraries are often highly configurable, with a wide range of options that can be overwhelming to a developer who is not familiar with them. PETSc is especially notable for this, providing a significant selection of solvers and preconditioners (some accessing further third-party libraries). Each of these in turn can provide their own set of options. Thus, the performance considerations of an algorithm must be well understood. For an iterative solver, this total walltime can be broken down into two factors – the total number of iterations required for convergence and the time taken for each individual iteration to occur.

The total number of iterations (i.e. the rate of convergence) can be attributed to the following factors:

- Algorithmic – the amount the error is reduced per iteration;
- Matrix condition number – the difficulty of solving the matrix. Typically the lower the condition number, the faster the rate of convergence;

- Preconditioning – An algorithmic approach that applies a transformation to the problem to obtain a lower condition number and thus an easier problem to solve;
- Convergence threshold – the required accuracy to determine if convergence has been acheived (the higher the accuracy, the more iterations that are required).

The time per iteration can be attributed to a different set of factors:

- Algorithmic – e.g. decomposition, communication patterns, number of matrix/vector operations per iteration, etc;
- Preconditioning – The cost of the extra step involved in preconditioning the problem;
- Machine hardware – e.g. CPU, memory bandwidth/latency, network interconnect, etc;
- Implementation optimisation – e.g. vectorisation, efficient cache usage, optimised math libraries, etc.

This chapter focuses upon the relationship between the time per iteration and the machine hardware; specifically, the performance behaviour of the PETSc implemented CG linear solver algorithm and how it relates to its parallel environment, resulting in the following goals:

- Produce a fine-grained performance breakdown of the CG solver for a sparse-matrix system of problems, with a particular focus upon collective operations at scale.
- Highlight how the techniques used for capturing and modelling Hydra could be applied to PETSc.
- Contrast the performance of PETSc's base CG solver and its communication-avoiding variant.

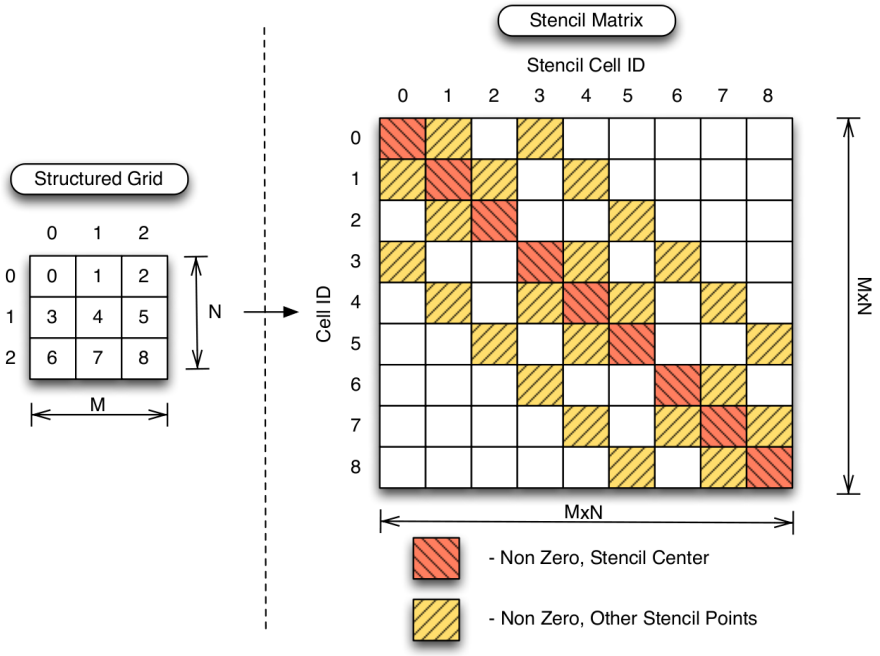


Figure 7.2: Structured Grid with 5-Point Stencil to Matrix

7.1.1 PETSc Descomposition Behaviour

For any linear system the properties of the matrix, as well as its condition number, can be highly influential upon the performance and validity of the solver. Using a linear solver for a system of stencil computations, a single row in the matrix contains all the values that pertain to a single cell's stencil – i.e. for a 5-point stencil there will be at most 5 non-zero entries in any arbitrary matrix row; which stencil is used for a row is determined by the row ID. A row in the matrix maps to the global ID of a single cell in a structured grid, meaning that for an $M \times N$ structured grid, the number of rows in the matrix is also $M \times N$. The values contained within said row consist of the stencil values for that cell, e.g., in Figure 7.2 cell 4 has a stencil that consists of values from itself and from cells 1, 3, 5 and 7. Thus row 4 of the matrix contains only the non-zero data for these respective cells. This non-zero stencil data is arranged in the matrix row in such a fashion that each piece of data has a column ID that maps to the global ID of the cell it belongs to. As a consequence, this

means that the number of columns in the matrix is also $M \times N$, resulting in a total matrix size of $M \times N \times M \times N$. This size, in conjunction with the limited number of entries per row, is what gives the matrix its sparse nature. When applied to a 3D problem, it is readily apparent that this problem of sparsity will only be exacerbated further. For a grid of $Nx \times Ny \times Nz$ there will be a maximum of 7 points per matrix row, but a global matrix size of $(Nx \times Ny \times Nz)^2$, highlighting the need for efficient handling of non-zero data in both computation and storage. The underlying PETSc library is capable of employing efficient data storage structures to overcome the sparsity of such data – Compressed Sparse Row (CSR) [15] techniques for matrix storage allow the storage of only the non-zero elements, their column indexes and the location of new rows. PETSc is capable of taking these CSR formats and populating its own internal matrix data structures.

The work-decomposition behaviour of any parallel application is pivotal in governing the frequency and size of any MPI messages required to fulfill data dependencies for parallel computation. When using PETSc's Distributed Array interface to handle grid decomposition, the structured grid is typically spread evenly across all available processes, minimising the surface area of any internal grid boundary such that each local grid is as cubic as possible. In turn, the PETSc parallel data structures are decomposed in a similar manner, with whichever process that owns a particular grid cell also owning its matching matrix row and vector elements. The benchmark in use for this chapter explores a structured 3D 7-point stencil problem rather than a 2D 5-point stencil, but the same principles apply to its construction.

The nature of any stencil based computation is such that some degree of communication must occur to ensure that local copies of remote data (i.e. the ghost cells/halo data) are up-to-date and accurate. The two most typical communication patterns found during the course of our investigations are collective operations, such as those found in global reductions for Vector Dot-Products, and near-neighbour communications, such as those found in Matrix-Vector Mul-

tiplication.

The collective operations are largely performance independent of the problem decomposition, with more influence being attributed to the number of processes and their mapping to the underlying hardware. Near-neighbour communications however are heavily reliant not only upon the network hardware but also upon the decomposition patterns, as it is this that governs the size and number of MPI message exchanges that must occur to resolve the data dependencies that result from stencil-based computation.

With the decomposition patterns employed in PETSc the processes with which communication must occur can be identified. Since the PETSc decomposition will never split a matrix row, the only remote data required for a Matrix-Vector multiply on a single row is the corresponding vector entries that match the columns of the matrix row entries. Since any matrix row consists solely of data belonging to a cell and its associated stencil locations, the required vector entries belong to the processes that also own the corresponding stencil cells. With a 7-point stencil, these can only be neighbouring processes in any of the three dimensions, resulting in a maximum of 6 remote processes which may require message exchanges. This describes a near-neighbour exchange pattern, where all required data is packed into a single message on each process and exchanged with its corresponding neighbouring processes.

7.2 Conjugate Gradient Performance Analysis

As the trend towards Exascale and large-scale multi/many core architectures continues, the emergence of new bottlenecks and a shift in the cost of operations that were previously trivial at small scale is likely to inhibit future efforts to optimise parallel applications [20, 155]. The notion of scalability becomes an ever greater concern and will necessitate a renewed focus upon the role of collective operations (amongst others), with the cost of such communications coming under greater scrutiny.

As a case study, the Conjugate Gradient solver is selected with no preconditioner, constructing a PETSc Distributed Array benchmark based on the Orthrus benchmark provided by AWE as the basis for our investigations. The Orthrus benchmark is responsible for the initial data constructs, with interactions with the PETSc benchmark performed via the use of the DMDA Application Program Interface (API) functions. The algorithm employs both near-neighbour communications and collectives, making it a prime candidate for investigating the behaviour of its various potential bottlenecks. The presence of similar near-neighbour communication patterns to Hydra also presents an interesting opportunity for comparison, allowing the potential application of similar techniques to those of Chapter 4.

In order to obtain a more refined instrumented breakdown of the PETSc CG implementation, a Performance and Modelling Timing Interface (PMTM) instrumented version of PETSc is used to obtain the results in this chapter. While PETSc can provide logging and timing functionality, these results are restricted in both detail (capturing only library calls rather than their internal components) and quantity (providing a more limited set of timing metrics than PMTM). Instead, the pair are used in conjunction with one another; PMTM timings provide computation/communication timings while PETSc logging features provide both validation and additional metrics such as frequency and size of MPI messages.

7.2.1 CG Breakdown

PETSc provides a range of typical Matrix/Vector operations via a set of library function calls and the use of PETSc data constructs. During the course of a single CG iteration, the following operations are used:

- One Matrix-Vector Multiply;
- One Vector Norm Computation;
- Two Vector Dot-Products;

- Two Vector AXPY computations – $y = \alpha x + y$;
- One Vector AYPX computation – $y = x + \alpha y$;
- PCApply – Application of the preconditioner to a vector.

From this function list, a rough model of the callpath can be derived via the summation of all library function calls. Of these functions, three contain some form of network communication – the Matrix-Vector Multiply (MatMult), the Vector-Norm (VecNorm) and the Vector Dot-Product (VecTDot). Delving further, it is revealed the Matrix-Vector multiply contains a near-neighbour communication exchange, while the remaining two functions both contain AllReduce global collectives. The PCApply method can contain communication, depending upon the preconditioner, but only unpreconditioned scenarios are explored in the following experiments and so PCApply is a simple local vector copy.

The nature of communication overheads means that any timings may contain not only network communication overheads, but also the synchronisation costs of ensuring that both sender and receiver are ready. One approach is to use the average time across all processes for each function call, but this may give a misleading impression of the collective costs (where load-balancing issues in other functions may be the true cause). As an alternate approach the minimum time of any library function that contains a collective operation is taken (which has an implicit barrier synchronisation), aiming to capture the minimum synchronisation cost involved with these functions. The maximum time for pure compute/memory functions is used (i.e. AXPY, AYPX and PCApply), as these timings lack any synchronisation points. In this manner, a similar approach to that taken for capturing the various components of Hydra in Chapter 4 is adopted.

The Matrix-Vector Multiply function is more complex. While not containing any explicit global synchronisation, there is a degree of synchronisation with a process's nearest neighbours, which can cause load-imbalance to propogate through the system. In addition, the computation in Matrix-Vector Multiply

P	Total Iterations	CG Solve Time (s)	Function Sum (s)	Error (%)
1	326	1.51	1.50	−1.14
2	451	3.52	3.49	−0.96
4	533	5.33	5.28	−0.82
8	605	10.50	10.40	−0.90
16	891	15.63	15.51	−0.72
32	1078	19.06	18.96	−0.54
64	1236	22.27	22.17	−0.47
128	1819	33.02	32.94	−0.24
256	2222	41.28	41.17	−0.26
512	2561	47.80	47.77	−0.07
1024	3721	71.05	70.99	−0.09
2048	4556	85.45	85.33	−0.14
4096	5256	100.64	100.96	0.32
8192	7573	149.78	150.94	0.78
16 384	9282	195.50	195.53	0.01

Table 7.1: CG Function Sum Validation, CG/No Preconditioner, HECToR, PGI-12.10/MPICH-5.6.1, Weak Scaling (50^3)

can form some of the most computationally expensive parts of the CG iteration, and thus has the greatest prospect of introducing imbalance (e.g. contention on the memory subsystem or other shared resources). In conjunction with empirical timings it is determined that the maximum timing for this function appears to be the most representative.

This results in Equation 7.7, a rough approximation of the function cost of a single CG iteration.

$$\begin{aligned} CGSolve_{iteration} = &\text{MatMult (Max)} + \text{VecNorm (Min)} + \\ &(\text{VecTDot (Min)} * 2) + (\text{VecAXPY (Max)} * 2) + \\ &\text{VecAYPX (Min)} + \text{PCApply (Max)} \end{aligned} \tag{7.7}$$

The goal is to validate this approach for a 50^3 weak-scaled problem up to 16384 cores for 20 timesteps (i.e. 20 CG solver executions). *CGSolver (Max)* is defined as the overall time measured for the CG solver across all iterations, while *Sum* is the outcome of applying Equation 7.7. From Table 7.1 it can be seen that this results in a time that is at most $\approx 1\%$ away from the actual total solve time, demonstrating that our breakdown sufficiently captures the CG iteration behaviour. When broken down by percentage in Figure 7.3, past four cores

(the point at which a single socket is fully populated) the breakdown is mostly consistent with a small shift away from compute-only functions such as $AXPY$ towards functions with communication operations.

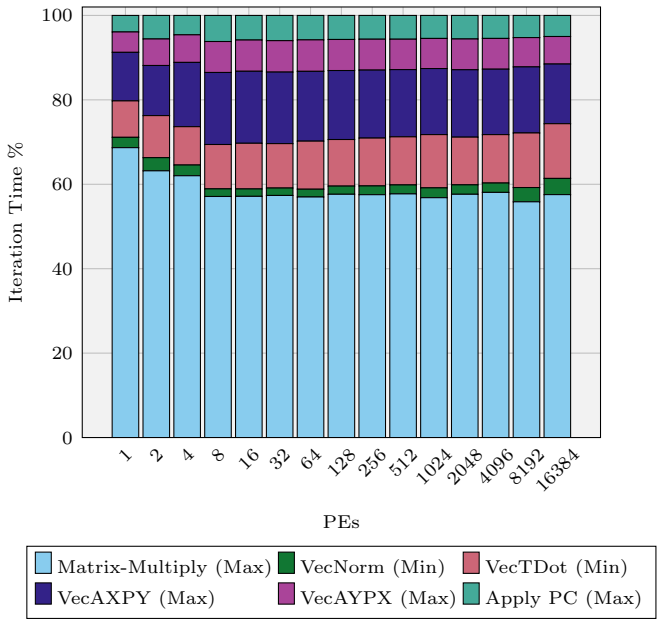


Figure 7.3: Breakdown by Percentage of CG Function Sum Time, CG/No Preconditioner, HECToR, Weak-Scaled, 50^3

A Matrix-Vector multiplication, $\mathbf{Mx} = \mathbf{r}$, for an m row by n column matrix is achieved via the following:

$$\forall_j \in \{0, 1...m-2, m-1\} : r_j = \sum_{i=0}^{n-1} M_{ij} * x_j \tag{7.8}$$

where r_j is a single element in the resulting vector. For each non-zero element in a matrix row, element A_{ij}, a matrix-vector multiply requires a local copy of the corresponding vector element x_j. The sparse nature of the matrix ensures only a small selection of the vector's elements will be required for each row. However, due to the manner of PETSc's decomposition, while a whole matrix row can be guaranteed to be on the same process, it may not possess up-to-date copies of all required vector elements. Thus, the process is divided into three distinct components: (a) compute on local components; (b) a near-neighbour ex-

change to resolve data dependencies; and, (c) all remaining compute using halo data from the near-neighbour exchange. These three steps match the Multiply Compute, VecScatterBegin/VecScatterEnd and Multiply-Add Compute stages respectively.

The compute stages are split into two due to the data-dependencies involved. In the first "local" compute stage the data-dependencies are already resolved, with the required data being up-to-date on the local process. This is the case for any Matrix-Vector element pair where the PETSc decomposition has placed both the matrix row element and the corresponding vector element on the same process. This constitutes the Multiply Compute stage.

In the second, "remote" compute stage, there exist matrix row elements on the local process for which the matching vector element is on a remote process. Thus before computation can go ahead a communication stage must occur to retrieve and refresh the local halo data with a copy of these values. These computations are grouped up and performed as part of the Multiply-Add Compute stage where, upon completing all product calculations, the final summation can occur to get the value for the result vector. This stage does not begin until all communication is complete.

The remaining two stages, VecScatterBegin and VecScatterEnd are two separate functions responsible for overseeing the completion of the near-neighbour exchange stage. To minimise the latency overheads, all halo data required for the remote compute stage on a neighbouring process is identified and gathered into a single packed buffer. This data is then communicated using non-blocking MPI functions within a single message, and a reciprocating message received from the neighbour process to populate the local process's halo data. This step is repeated for all other neighbouring processes till all data-dependencies are resolved. The exchange is split into two functions to faciliate communication overlap. Overlap is not guaranteed due to the complex nature of communication overlap and its dependence on overlap techniques, MPI implementation and network hardware [30, 97, 170, 171]. However it does enable the potential ex-

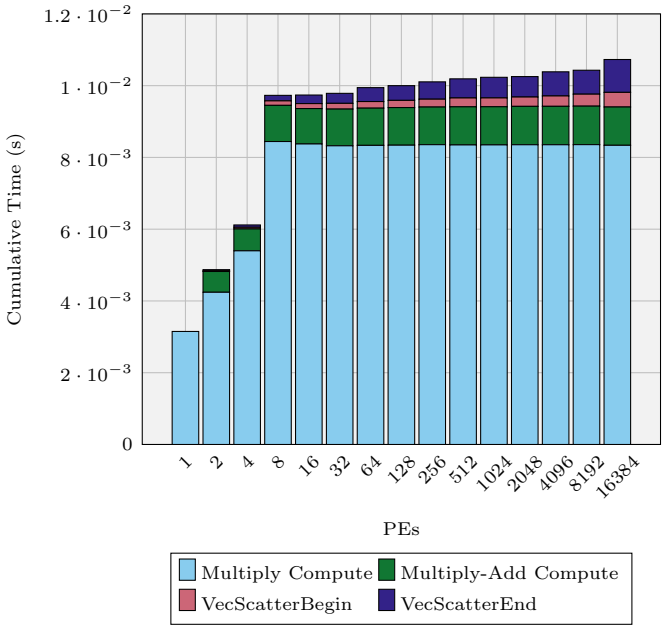


Figure 7.4: Single Matrix-Multiply Call (Mean) Breakdown by Function in CG, CG/No Preconditioner, HECToR, Weak-Scaled, 50^3

ecution of communication exchanges while simultaneously performing the non-dependent local compute step, which would result in an overall speedup; Hence the check for completion of communication (VecScatterEnd) is not performed until after the completion of Multiply Compute, despite the communication being started before the local compute stage.

Figure 7.4 presents the average time spent in each of these components, with a summed cumulative time equal to the average time spent in the Matrix-Vector Multiply library function. It is evident that as the process count is scaled, the average time spent in both of the compute functions remains relatively consistent, with the exception of between one and eight cores. In constrast, there is an increasing trend in the time spent in the near-neighour exchange components, despite the message sizes remaining reasonably consistent (due to the weak-scaling nature of the problem). These increases could be attributed to a number of factors, including synchronisation, contention or process affinity. An increase in process count can impact the network utilisation and load balancing

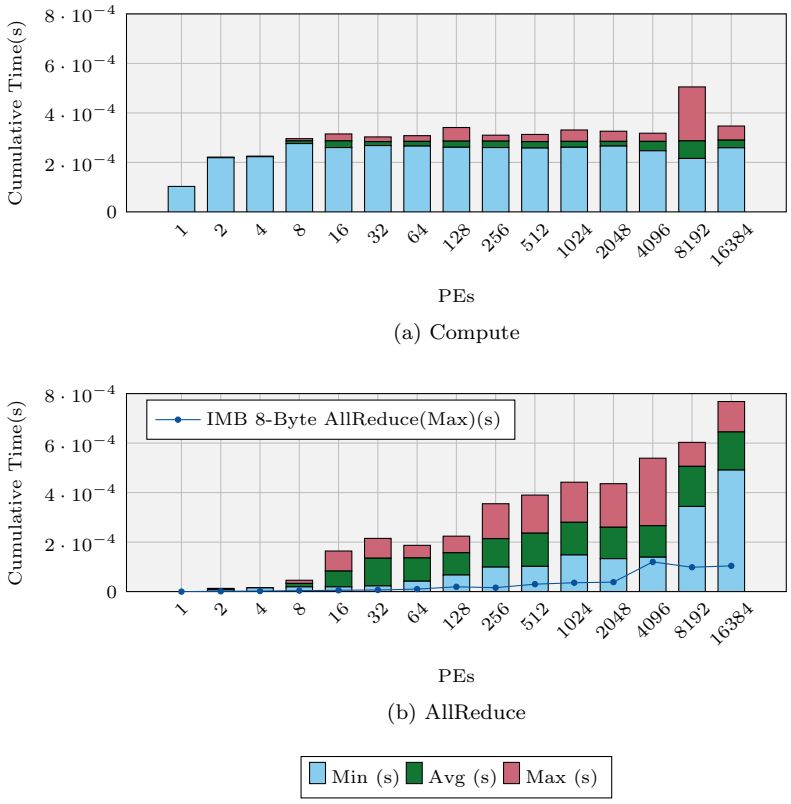


Figure 7.5: VecNorm Components, CG/No Preconditioner, HECToR, Weak-Scaling 50^3

of the problem, with minor variances in the compute time being propogated throughout the system, leading to a more significant impact as the process counts increase. In addition, with more processes, the process ID between neighbours becomes greater (e.g. a Y dimension neighbour's ID is separated from the ID of the current process by the length of the X decomposition). Depending upon the network architecture and the manner of process-to-core allocation, this could lead to more physically distant communications with all the additional overheads this entails.

The Vector-Norm library function does not contain any near-neighbour collectives. It does however include a blocking MPI AllReduce global collective alongside its local computation step, with the accompanying synchronisation

step this requires. Figure 7.5 highlights the disparity that exists between the aggregate minimum, average and maximum timings across the different processes. It is apparent that, with the exception of a few anomolies, the compute timings are relatively stable. The AllReduce collective function however demonstrates a significant amount of variance, with both the minimum and maximum times taken dominating that of the compute portions of the function. This runs contrary to our understanding of the network performance of HECToR. For reference, the 8 byte AllReduce performance originally reported in Figure 3.4(b) is overlaid on top. It is apparent that even if the focus is only upon the minimum time taken, the benchmarked AllReduce time is significantly less than that reported by the Vector-Norm function. Since the original IMB benchmark is unlikely to capture characteristics such as load-imbalance, this time could potentially be attributed to synchronisation costs born out of variances in the compute or other library functions. Nonetheless, this breakdown reveals that such collectives are making a contribution to the costs of these functions, albeit it is unclear if it as the result of actual communication costs or due to synchronisation issues.

Extending this same process to the Vec Dot-Product functions (VecTDot), it can be established that a similar behaviour is occuring. There is a little more variance between the various statistical aggregates of the compute, but these are relatively marginal in comparison to the variance of the AllReduce MPI calls. Curiously, since both the Vector Dot-Product and the Vector Norm functions operate on a single double (8 bytes) per process, it would be expected that they demonstrate similar timings. This does not appear to be the case however, lending more credence to the theory that this variance could be attributed to other factors such as synchronisation costs due to imbalance. This could be potentially problematic for any attempt at a communication-avoiding algorithm at this scale, as the advantage of such approaches is eliminating the cost of communication. They would not however eliminate synchronisation costs attributable to load-imbalance, as these would merely be shifted to the next synchronisation

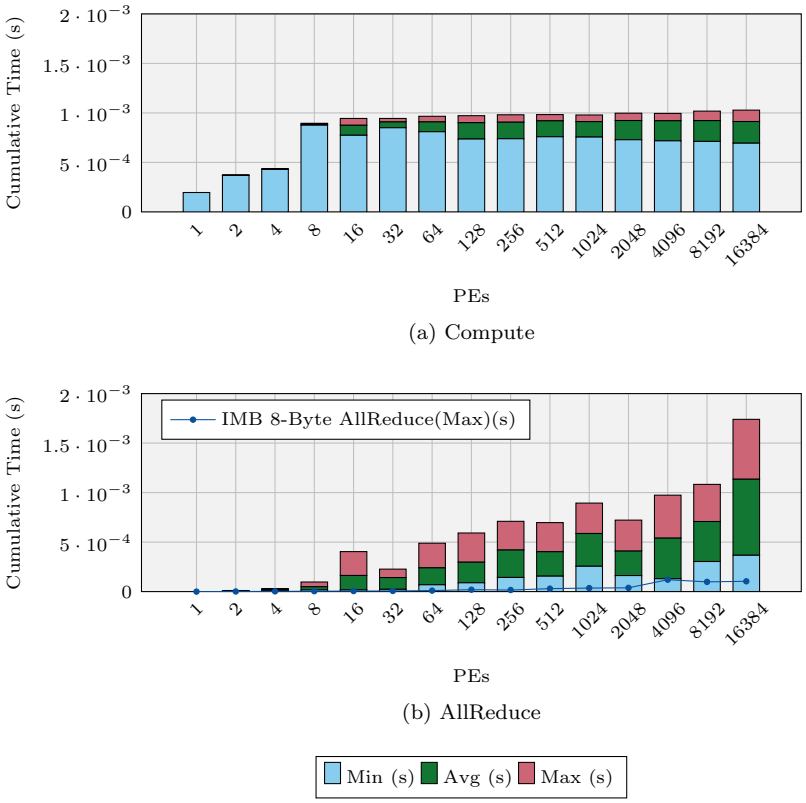


Figure 7.6: VecTDot Components, CG/No Preconditioner, HECToR, Weak-Scaling 50^3

point (assuming that the load-imbalance is not corrected before this point is reached). To fully explore this possibility, the next section will explore the use of such a communication-avoiding algorithm built into PETSc to combine two Vector Dot-Products into one per CG iteration.

However, before exploring the application of optimisations at large-scale, it is first necessary to address the poor scaling behaviours at a small number of cores in these experiments. If all the functions are examined for general trends, it is apparent that the compute contributions all exhibit the same increasing trend at very small scale, before stabilising at approximately eight cores, reminiscent of prior observed behaviour elsewhere. When examining that initial set of results, it is speculated that the poor scaling performance could have been a consequence

of either memory-bottlenecks or an increasing communication cost due to a rising number of messages. However, when viewed in conjunction with the breakdown measurements contained in this section, it is clear that the most significant increases in walltime when scaling on a single node are a result of increasing compute costs, not communication costs.

Since the poor performance is isolated to compute components only, it is a likely conclusion that either memory or CPU performance is responsible for the erratic behaviour. Documentation of the underlying CPU architecture reveals that the Interlagos chip pairs cores together into modules. [127] Within these modules a number of resources are shared, most notably the Floating Point Execution Units. It is therefore possible for some degree of contention to occur within these modules for select scenarios. However, this is raised as a possibility, there is no manifestation of poor scaling behaviour past eight cores. Such behaviour is expected to be apparent for up to a full node, up to 32 cores on HECToR, if the CPU structure was responsible. Since this does not appear to be the case, it is likely not the cause of the observed behaviour, therefore we would conclude that the primary factor lies elsewhere. Considering the STREAM benchmarks of 3.3.2, and how Minerva exhibits similar behaviours without the Interlagos chip structure being a factor, this would appear to reinforce the conclusion that not only is some degree of resource contention occuring. but that it is likely memory contention and is significant enough to be resposible for the poor single-node performance of these linear solvers.

7.2.2 Coalesced CG

In PETSc version 3.3 there exists an alternate version of CG intended to minimise the overall frequency of global collective AllReduce calls. Accessed via the PETSc command line argument "-ksp_cg_single_reduction", this option uses an approach where the number of AllReduces from Dot-Products is halved, instead trading it for a roughly equivalent increase in the number of calls to AYPX, a communication independent function. The frequency of different library calls

Function	CG (Original)	CG (Coalesced)	Comms.
MatMult	i	i+2	Neighbour Exchange
VecNorm	i+1	i+1	AllReduce
VecTDot	2i	3	AllReduce
VecMDot	0	i-1	AllReduce
AXPY	2i	2i	None
AYPX	i-1	2(i-1)	None
PCApply	i+1	i+1	Situation Dependent

Table 7.2: CG Function Call Frequency across i Iterations

between the original and coalesced version of CG is detailed in Table 7.2, obtained via instrumentation call counters and source-code inspection.

It is apparent from this table that the number of AYPX function calls has doubled, while the two calls to VecTDot have been replaced by a single call to VecMDot. The VecTDot function is a PETSc library function that computes the Vector Dot-Product of a single vector. VecMDot takes multiple vectors as input, computing the Dot-Products of all vectors involved but combining the final AllReduce step of each Dot-Product into a single global collective call (with an extra double in the send and recieve buffers for each vector involved). In this scenario VecMDot operates upon two vectors. Thus, while the number of calls to VecMDot is halved, the number of Vector Dot-Products involved overall is the same – the primary reduction comes from combining two AllReduces into one, minimising the impact of latency costs.

Since a Vector Dot-Product has both compute and a global AllReduce, and AYPX has only compute, theoretically this code has greater scalability by eliminiating the overhead of the collective communication. However for a general improvement this would require the assumption that the compute overheads of AYPX are equivalent to, or less than, the savings made by combining two AllReduces into a single AllReduce. The minor variances in other functions (such as the two extra Matrix-Vector Multiplies) is discounted, since they should have little overall impact with a large iteration count. When the cost of an AllReduce is not significant, such as at small-scale, then any variance in time taken would likely be attributed to the difference in compute/memory costs between the two functions.

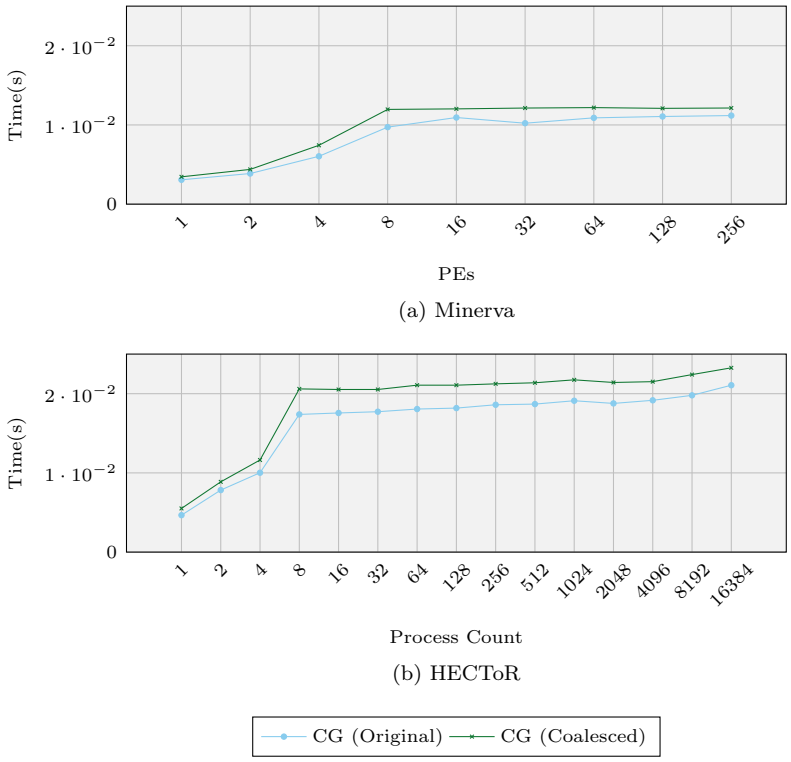


Figure 7.7: Solve Time per Iteration, CG/No Preconditioner, Weak-Scaled, 50^3

To investigate the effectiveness of a coalesced CG implemented in such a manner, the earlier scaling investigations for CG with no preconditioner are repeated. All experiments were conducted using the PMTM instrumented version of PETSc. To ensure a fair comparison, the following results for both the original CG and coalesced CG are run using this built-from-source version of PETSc (as opposed to reusing earlier results for the base implementation of CG).

Figure 7.7 presents the outcome of the weak scaling experiment. For an arbitrary core count, the number of iterations for both the original and coalesced CG algorithms was the same, thus the focus is upon the time spent per iteration by each. A simple comparison reveals that for strong and weak scaling, on both Minerva and HECToR, the coalesced algorithm appears to be slightly worse than that of the original algorithm. The cause is likely to fall into one of two categories – either the saving was not significant enough to overcome the trade-

off, or the performance of the collectives operations includes a synchronisation cost that is not eliminated by the use of a coalesced algorithm (merely moved elsewhere).

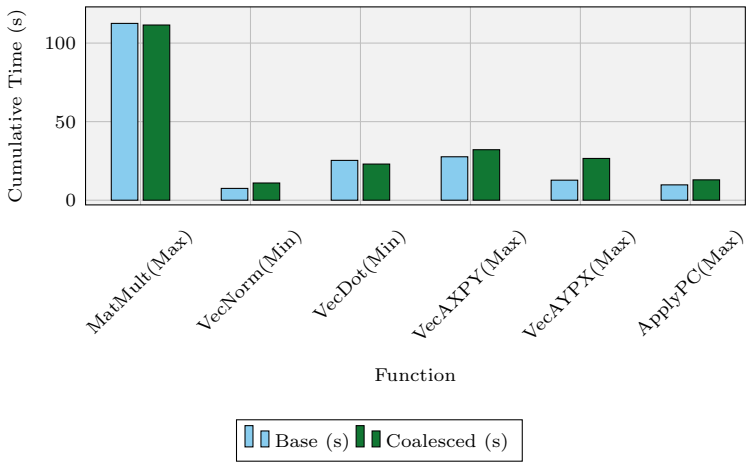


Figure 7.8: Base vs Coalesced CG Function Breakdown, CG/No Preconditioner, HECToR, Weak-Scaled, 50^3, 16384 Cores

A similar breakdown process to Section 7.2.1 is now applied to the coalesced reduction algorithm, presented in Figure 7.8. For reference a side-by-side comparison is provided between the original algorithm and the coalesced approach presented within this section.

The side-by-side comparison reveals a number of expected outcomes. Most notable is the disparity in the VecAYPX results. This is the expected tradeoff for adopting this approach – doubling the number of AYPX calls has resulted in roughly double the time spent within this function. A counter to this trade-off is also seen – a relatively minor saving in the time spent performing VecDot functions (VecDot here refers to VecTDot in the base algorithm and VecMDot in the coalesced approach). These savings however would suggest that even at 16,384 cores, the overall saving is not sufficient to overcome the increased compute trade-off – the cost of the AllReduce function would need to be more significant, potentially at a greater number of cores, before improvements in performance could be expected. However, there also exist a number of unexpected

discrepancies that could also impact upon the overall walltime.

Of the other functions that are expected to remain static in the context of performance, only the Matrix-Vector Multiply appears to have done so. The compute only functions ApplyPC and VecAXPY functions have seen a small increase in their overall cost. In addition, the VecNorm function has also seen an increase in its walltime.

While unexpected, the VecAXPY and PCApply (in a communication-free preconditioning context) can be overcome with scaling, or could be the consequence of some otherwise unidentified factor. However the prospect of an increased Vector-Norm function is disconcerting, due to it also incorporating an AllReduce as part of its function. Thus further investigations, presented in Figures 7.9a and 7.9b, seek to decompose the behaviour into compute and collective components.

From these results a simple conclusion can be taken away – the cause of the increased Vector-Norm cost is potentially attributable to both compute *and* the collective components, with both having maximums that are significantly higher for the coalesced approach than for the base CG algorithm. In the case of the compute, it would appear that the compute costs are approximately double for the coalesced algorithm than for the base CG algorithm. At the time of the writing however it is was not apparent as to what the source of this increased compute cost is, and would likely have to be the focus of further investigation. Nonetheless, it is readily apparent that the compute is at least in part, if not primarily, responsible for the increase in Vector-Norm walltime.

The impact of the AllReduce is more complex. While the overall maximums have increased, the minimums are still on a par with one another for both algorithmic approaches. Since there is not an expected increase in the cost of the Vector-Norm AllReduce, it could potentially be a consequence of different synchronisation costs due to a modification in the order of functions called. Such costs may not actually be an increase, but costs that were previously attributed to, or the consequence of, other functions. For example, the disparity in the

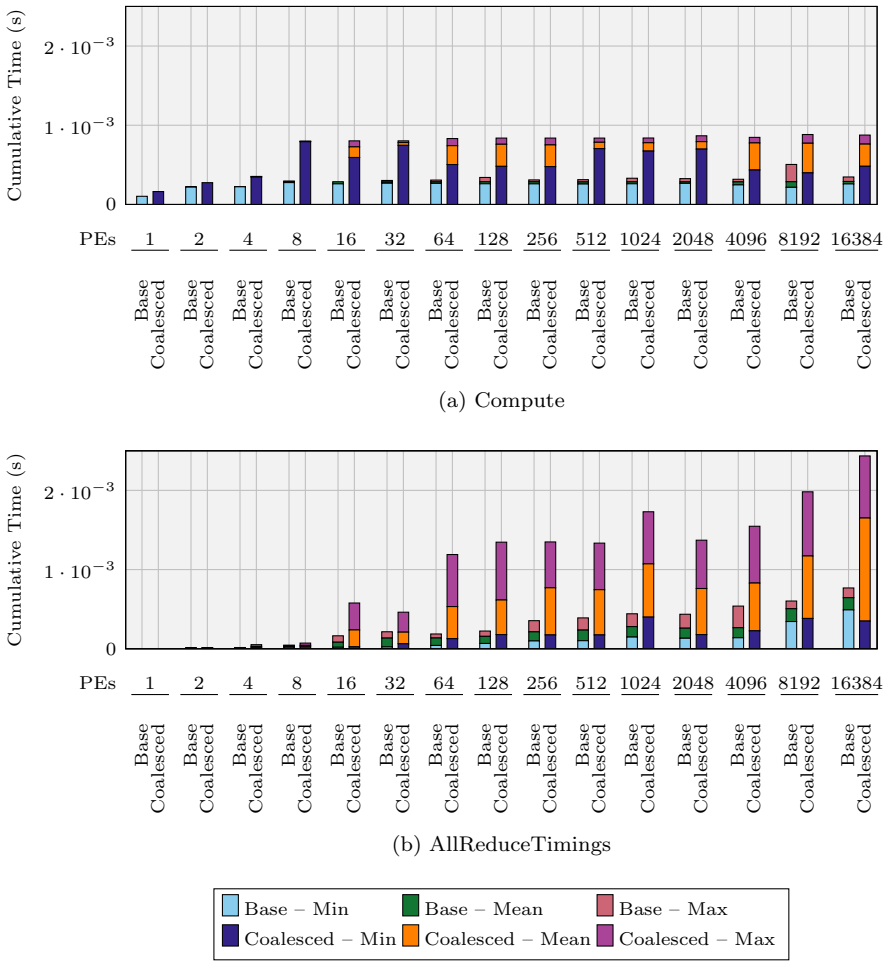


Figure 7.9: Base vs Coalesced CG, VecNorm, CG/No Preconditioner, HECToR, Weak-Scaled, 50^3

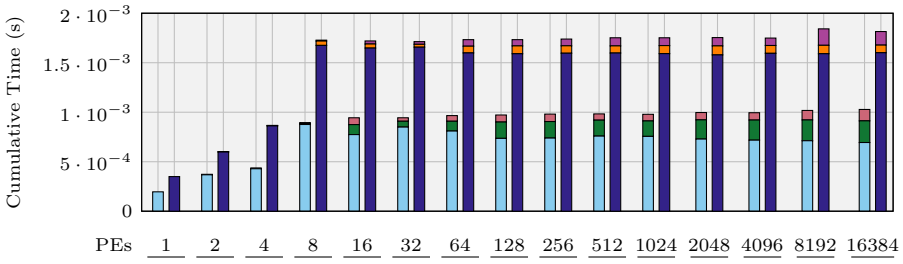
minimum and maximum times spent in the compute portion of this function would likely factor as part of the time spent synchronising in the maximum AllReduce time.

Finally, a direct comparison between the cost of the compute and collective components of the Dot-Product functions is shown. For the base algorithm VecDot here refers to VecTDot, while for the coalesced algorithm this refers to VecMDot.

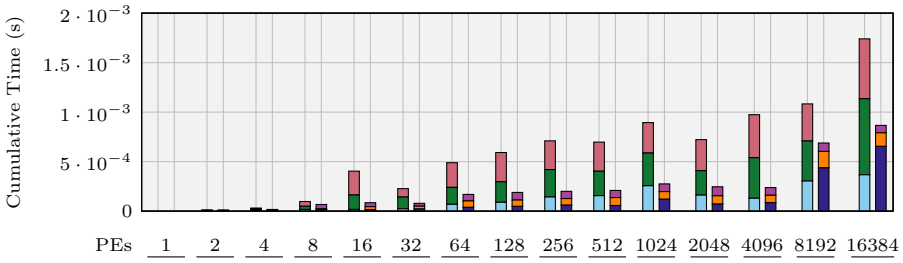
As expected, the compute portion of the VecMDot is roughly equivalent to

twice that of the VecTDot function in the base algorithm – a consequence of VecMDot performing twice as much work for half the number of calls. Thus the overall time spent by each performing compute is similar across an iteration.

For the AllReduce component, there is a general trend of a decrease for the coalesced algorithm. Since aggregate time spent in a single function call is used as opposed to overall time spent, it may have been expected that the two algorithms spend a similar time in the collective operations. Thus a hypothesis is that this is similar to the Vector-Norm collective differences, where synchronisation costs may play some role in the variance between the two sets of results, either due to variance in compute or due to the order in which functions are called resulting in a shift where synchronisation costs are attributed to a different function.



(a) Compute



(b) AllReduce

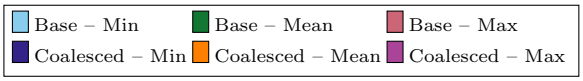


Figure 7.10: Base vs Coalesced CG, VecDot, CG/No Preconditioner, HECToR, Weak-Scaled, 50^3

7.3 Summary

This section has presented a more in-depth investigation of the Conjugate Gradient algorithm, revealing insights into not only compute, but also point-to-point and collective communication components of the algorithm. In particular, the consequences of an increasing number of cores upon the AllReduce collectives were explored, identifying the subsequent increase in communication costs. It is also shown that while these costs are likely to become ever more significant at scale, they are not significant enough at up to 16384 cores to overcome trade-offs in compute in order to exploit communication-avoiding approaches such as the coalesced AllReduce algorithm. This is in part due to the additional number of AYPX calls, but also in part due to an as yet unidentified general increase in compute across all functions.

CHAPTER 8

Conclusions

The state of the High Performance Computing (HPC) field is constantly in flux. Given that HPC provides the foundation for enabling many modern advances in both science and industry, pushing the boundaries of what it is capable of has remained a constant goal for those in this domain. Recent years have seen the arrival of a multitude of different hardware configurations, including highly-parallel large-scale clusters, co-processor based computing and further specialised hardware; as a consequence, enabling the efficient use of these systems has been, and will likely continue to be, an interesting challenge of note – especially with the goal of Exascale computing on the horizon. This work has explored the use of performance modelling and analysis, demonstrating how they can be applied to aid science and industry in their endeavours.

In Chapter 4 a case-study of a Hydrodynamics benchmark was introduced. In doing so, it was demonstrated how such codes can show interesting performance characteristics, as well as unexpected behaviours that warrant further investigation. This provides justification for the motivation of this work, using performance modelling and analysis to identify, characterise and potentially optimise codebases such as this for use in the highly parallel environment of HPC. Serving as a useful case-study, it formed the basis for the work that followed.

Chapter 5 constructs the aforementioned analytical model of this application, demonstrating a repeatable, step-wise approach to breaking down and sub-modelling the various contributors to the overall walltime of the benchmark. A notable characteristic was that the maximum of the compute and minimum of the communication times, along with the sync time, reasonably captures the overall performance of the application, as was posited by the work

of Adve [2]. In turn, this characteristic can be used to construct sub-models of the communication patterns/message times, as well as deconstructing the compute behaviour into unit-time blocks based on an iteration of an internal loop for each individual kernel. These times, represented as Wg values, abstract the compute/memory cost performance into a single value per kernel, which can easily be tied to the input parameters via a derivation of the total number of loop iterations and then extrapolated into a total walltime per kernel. When combined, these sub-models allow for the prediction of an overall walltime, as well as showing up any discrepancies such as the disjoint between the performance of the Madvmz and Madvmx, despite their similarity in functionality. Such models can be used to analyse performance not only on existing machines, as was validated for three different architectures in this work, but potentially also for future architectures. Even in cases where full large-scale examples of such machines do not yet exist, only a single node (for seeding Wg values) and a network benchmark on a small-scale can allow for predictions at larger, unknown configurations.

Chapter 6 uses the knowledge from Chapters 4 and 5 to explore potential optimisations for Hydra, focusing upon the memory, compute and network characteristics. While the compute optimisations are minimal in their impact at best, due to the memory-bound behaviour of the code as has been discovered through both Performance Application Programming Interface (PAPI) and the performance characteristics in Chapter 4, the discrepancies identified for the Madvmz_1 kernel lead to potential optimisation in the memory behaviour, leading to an overall speedup of approximately 1.3× to 1.4×. In addition, the use of OpenMP threads highlights the lack of obverlapping behaviour from the Message Passing Interface (MPI) non-blocking implementation, and while the poor performance of select compute kernels with OpenMP threads leads to an overall slowdown, it shows that such a behaviour is technically feasible in Hydra, and can lead to an overall speedup over a non-overlapping variant if the problem with underperforming OpenMP kernels is addressed.

Chapter 7 serves to demonstrate how these processes are not restricted to just a single piece of software, introducing and highlighting Portable, Extensible Toolkit for Scientific Computing (PETSc), a library that is available for use across within many scientific or industrial applications. While serving a different purpose to Hydra, it is shown how the techniques in this work can be used to construct breakdowns of other codes, and a simple model of the critical path using a max/min approach once again gets within 1% of the overall runtime. Further breakdowns also show similar behaviours to Hydra in regards to the compute and communication characteristics — compute times are relatively fixed once a problem size is fixed past a minimum Processing Element (PE) count (due to memory bandwidth behaviours), and the communication patterns are striking similar due to both codes adopting a near-neighbour based approach to data-transfer. While not conducted in this work, the derivation of Wg values and prediction of message times would seem to follow a strikingly similar approach to that of the work in earlier chapters, highlighting how such techniques can be transferable across codebases.

8.1 Thesis Limitations

The use of analytical performance modelling within this work has proven viable for Hydra, the benchmark application of interest, the core focus of this work. However, while this work has demonstrated the similarity in behaviour of the PETSc library, it should be highlighted that the use of such analytical models is on an application-by-application basis, and applying the technique to another code necessitates a time investment by a developer who is familiar with the code in question. This does not limit the theoretical application of such a technique but the approach within this work does not tackle other issues associated with the practical application of the approach, such as the overheads and costs, in both money and time, required to produce a model.

Additionally, this work has focused upon the performance prediction of this

application, but has not addressed additional factors such as power/energy costs. Further, for reducing complexity in the initial construction of the model, more nebulous variable costs such as I/O input have been discounted from the contributing costs, yet in a real-world scenario they remain a factor in the overall performance of running the application. Additional factors, such as the use of mixed cell benchmarks, also remain an additional concern and would require a further extension of the model.

Finally, while the model itself has validated satisfactorily, it is difficult to validate for the most extreme of scales such as that which might be seen by Petascale machines before such machines are developed and become more widely available. Nevertheless, this does not reduce the usefulness of the model for smaller scale and existing machines, nor does it mean that it is incapable of predicting at higher scale, rather that the model may benefit from access to larger scale machines for further validation.

8.2 Future Work

There exist a number of areas of interest for future work on this subject. Tackling some of the limitations of this thesis, focus upon the use of automated tools for the implementation of such analytical models would prove of great interest for speeding up the process of their development. These techniques can be applied in one of two ways:

1. The application of automated instrumentation tools. Existing tools already exist such as source-to-source compilers or dynamic library handling that enable the insertion of timing code. By identifying suitable locations in an automated manner, codes can have their critical paths automated processed into an overarching analytical model.

2. The use of automation can remove some of the necessary domain knowledge by employing automated experimentation techniques to tie the behaviours of instrumented blocks of code to a pre-defined set of input pa-

rameters. A number of existing works in the field already exist for the purposes of automatically applying such an approach to experiments in general. Statistical techniques can then be applied in such a manner that relationships could be drawn without the need for intimate knowledge of the code-base.

Finally, given the tendancy towards more unusual architectures that will likely arise as Petascale machines are closer to reality, incorporating more heterogenous architectures within modelling efforts will likely become a must, such as the Intel Xeon Phi or GPU-based computing. In particular, capturing the behaviour of the data transfer onto accelerator devices, as well as generating performance metrics for kernels on said devices, can enable the construction of models that could balance workloads between an accelerator and the host device's Central Processing Unit (CPU) in order to more effectively execute code in hybrid hardware environments.

8.3 Final Words

While the physical and technical limitations of Moore's Law may have been more keenly felt in recent years, the field of HPC shows no sign of slowing with regards to its continual pursuit of ever greater performance.

The Top 500 has demonstrated how the goal of Exascale computing is ever closer, with the rise of co-processor/hybrid computing and increasingly more performant parallel systems enabling a greater rate of scientific throughput and processing than has ever been available in history.

With such a rapid rate of advance in the field this author looks forward with great excitment to what HPC will have to offer in the future.

Bibliography

[1] N. R. Adiga, G. Almási, Y. Aridor, R. Barik, D. K. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. A. Blumrich, A. A. Bright, J. R. Brunheroto, C. Cascaval, J. G. Castaños, W. Chan, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. L. Chiu, T. M. Cipolla, P. Crumley, K. M. Desai, A. Deutsch, T. Domany, M. B. Dombrowa, W. E. Donath, M. Eleftheriou, C. C. Erway, J. Esch, B. G. Fitch, J. Gagliano, A. Gara, R. Garg, R. S. Germain, M. Giampapa, B. Gopalsamy, J. A. Gunnels, M. Gupta, F. G. Gustavson, S. Hall, R. A. Haring, D. F. Heidel, P. Heidelberger, L. Herger, D. Hoenicke, R. D. Jackson, T. Jamal-Eddine, G. V. Kopcsay, E. Krevat, M. P. Kurhekar, A. P. Lanzetta, D. Lieber, L. K. Liu, M. Lu, M. P. Mendell, A. Misra, Y. Moatti, L. S. Mok, J. E. Moreira, B. J. Nathanson, M. Newton, M. Ohmacht, A. J. Oliner, V. Pandit, R. B. Pudota, R. A. Rand, R. D. Regan, B. Rubin, A. E. Ruehli, S. Rus, R. K. Sahoo, A. Sanomiya, E. Schenfeld, M. Sharma, E. Shmueli, S. Singh, P. Song, V. Srinivasan, B. D. Steinmacher-Burow, K. Strauss, C. W. Surovic, R. A. Swetz, T. Takken, R. B. Tremaine, M. Tsao, A. R. Umamaheshwaran, P. Verma, P. Vranas, T. J. C. Ward, M. E. Wazlowski, W. Barrett, C. Engel, B. Drehmel, B. Hilgart, D. Hill, F. Kasemkhani, D. J. Krolak, C. Li, T. A. Liebsch, J. A. Marcella, A. Muff, A. Okomo, M. Rouse, A. Schram, M. Tubbs, G. Ulsh, C. D. Wait, J. Wittrup, M. Bae, K. A. Dockser, L. Kissel, M. K. Seager, J. S. Vetter, and K. Yates. An Overview of the BlueGene/L Supercomputer. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, SC'02, pages 7:1–7:22, Baltimore, Maryland, USA, November 2002. IEEE Computer Society, Los Alaminos, CA, USA, IEEE.

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

[3] V. S. Adve, R. L. Bagrodia, J. C. Browne, E. Deelman, A. Dube, E. N. Houstis, J. R. Rice, R. Sakellariou, D. J. Sundaram-Stukel, P. J. Teller, and M. K. Vernon. POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems. *IEEE Transactions on Software Engineering*, 26(11):1027–1048, November 2000.

[4] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model–One Step Closer Towards a Realistic Model for Parallel Computation. In *Proceedings of the*

Seventh Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA'95, pages 95–105, Santa Barbara, CA, USA, July 1995. ACM, New York, NY, USA.

[5] G. Almási, C. Archer, J. G. Castaños, C. C. Erway, P. Heidelberger, X. Martorell, J. E. Moreira, K. Pinnow, J. Ratterman, N. Smeds, B. Steinmacher-burow, W. Gropp, and B. Toonen. Implementing MPI on the BlueGene/L Supercomputer. In *Euro-Par 2004 Parallel Processing*, volume 3149 of *Lecture Notes in Computer Science*, pages 833–845. Springer, Berlin/Heidelberg, Germany, 2004.

[6] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the American Federation of Information Processing Societies Spring Joint Computer Conference*, AFIPS'67 (Spring), pages 483–485, Atlantic City, New Jersey, USA, April 1967. ACM, New York, NY, USA.

[7] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and Computation Transformations for Multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP'95, pages 166–178, Santa Barabara, California, July 1995. ACM, New York, NY, USA.

[8] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Achieving High Sustained Performance in an Unstructured Mesh CFD Application. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC'99, Portland, Oregon, USA, November 1999. ACM, New York, NY, USA.

[9] ARM. Cortex-A9 NEON Media Processing Engine Technical Reference Manual — Revision:r4p1. Technical report, ARM, June 2012.

[10] C. Ashcraft and R. G. Grimes. SPOOLES: An Object-Oriented Sparse Matrix Library. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, PPSC'99), San Antonio, Texas, USA, March 1999. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[11] I. T. Association et al. *Infiniband Architecture Specification: Release 1.0*. Infiniband Trade Association, 2000.

[12] W. Augustin and T. Worsch. Usefulness and Usage of SKaMPI-Bench. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2840 of *Lecture Notes in Computer Science*, pages 63–70. Springer, Berlin/Heidelberg, Germany, 2003.

[13] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, SC'91, pages 158–165, Albuquerque, NM, USA, November 1991. ACM, New York, NY, USA.

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

[15] S. Balay, J. Brown, , K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Users Manual. Technical Report ANL-95/11 - Revision 3.4, Argonne National Laboratory, 2013.

[16] S. Balay, J. Brown, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page. `http://www.mcs.anl.gov/petsc`, 2013.

[17] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. *Efficient Management of Parallelism in Object Oriented Numerical Software Libraries*, pages 163–202. Birkhäuser Boston, Cambridge, MA, USA, 1997.

[18] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Entering the Petaflop Era: The Architecture and Performance of Roadrunner. In *Proceedings of the ACM/IEEE 2008 Conference on Supercomputing*, SC'08, pages 1:1–1:11, Austin, Texas, USA, November 2008. IEEE Press Piscataway, NJ, USA.

[19] G. D. Benson, C.-W. Chu, Q. Huang, and S. G. Caglar. A Comparison of MPICH Allgather Algorithms on Switched Networks. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2840:335–343, 2003.

[20] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. therine Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. D. neau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. y Williams, and K. Yelick. Exascale

computing study: Technology challenges in achieving exascale systems, 2008.

[21] A. Bérut, A. Arakelyan, A. Petrosyan, S. Ciliberto, R. Dillenschneider, and E. Lutz. Experimental verification of landauer/'s principle linking information and thermodynamics. *Nature*, 483(7388):187–189, March 2012.

[22] L. N. Bhuyan and D. P. Agrawal. Generalized Hypercube and Hyperbus Structures for a Computer Network. *IEEE Transactions on Computers*, C-33(4):323–333, April 1984.

[23] M. Blumrich, D. Chen, P. Coteus, A. Gara, M. Giampapa, P. Heidelberger, S. Singh, B. Steinmacher-Buros, T. Takken, and P. Vranas. Design and Analysis of the BlueGene/L Torus Interconnection Network. Technical Report Report RC23025 (W0312-022), IBM Research Division, 2003.

[24] W. Bolosky and M. Fitzgerald, R .and Scott. Simple but Effective Techniques for NUMA Memory Management. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, SOSP'89, pages 19–31, Litchfield Park, AZ, USA, December 1989. ACM, New York, NY, USA.

[25] S. Borkar and A. A. Chien. The Future of Microprocessors. *Communications of the ACM*, 54(5):67–77, May 2011.

[26] N. E. Bowler, A. Arribas, K. R. Mylne, S. E. Robertson, and S. E. Beare. The MOGREPS Short-Range Ensemble Prediction System. *Quarterly Journal of the Royal Meteorological Society*, 134(632):703–722, April 2008.

[27] J. T. Bradley. *Towards reliable modelling with stochastic process algebras.* PhD thesis, University of Bristol, 1999.

[28] R. Brightwell, B. W. Barrett, K. S. Hemmert, and K. D. Underwood. Challenges for High-Performance Networking for Exascale Computing. In *Proceedings of the 19th International Conference on Computer Communications and Networks*, ICCCN'10, pages 1–6, Zurich, Switzerland, August 2010. IEEE Press, Piscataway, NJ, USA.

[29] R. Brightwell, D. Doerfler, and K. D. Underwood. A Comparison of 4X InfiniBand and Quadrics Elan-4 Technologies. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, CLUSTER'04, pages 193–204, San Deigo, California, USA, September 2004. IEEE Computer Society,Los Alaminos, CA, USA, IEEE.

[30] R. Brightwell and K. D. Underwood. An Analysis of the Impact of MPI Overlap and Independent Progress. In *Proceedings of the 18th Annual International Conference on Supercomputing*, ICS'04, pages 298–305, Malo, France, June–July 2004. ACM, New York, NY, USA.

[31] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, August 2000.

[32] R. Bunt, S. Pennycook, S. Jarvis, L. Lapworth, and Y. Ho. Model-Led Optimisation of a Geometric Multigrid Application. In *Proceedings of the 15th IEEE International Conference on High Performance Computing and Communications and The 11th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, HPCC-EUC'13, pages 742–753, Zhangjiajie, Hunan Province, China, November 2013. IEEE, IEEE Computer Society, Los Alaminos, CA, USA.

[33] D. Burger, J. R. Goodman, and A. Kägi. Memory Bandwidth Limitations of Future Microprocessors. *SIGARCH Computer Architecture News*, 24(2):78–89, May 1996.

[34] D. Callahan, J. Cocke, and K. Kennedy. Estimating Interlock and Improving Balance for Pipelined Architectures. *Journal of Parallel and Distributed Computing*, 5(4):334–358, August 1988.

[35] J. Cao, D. J. Kerbyson, E. Papaefstathiou, and G. R. Nudd. Performance Modeling of Parallel and Distributed Computing using PACE. In *Proceedings of the 19th IEEE International Performance, Computing and Communications Conference*, IPCCC'00, pages 485–492, Phoenix, Arizona, USA, February 2000. IEEE Press, Piscataway, NJ, USA.

[36] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[37] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova. Evaluation of the Intel® Core™ i7 Turbo Boost Feature. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, IISWC'09, pages 188–197, Austin, TX, USA, October 2009. IEEE Computer Society, Los Alamitos, CA, USA, IEEE.

[38] G. Chrysos. Intel® xeon phi coprocessor — the architecture. `https://software.intel.com/en-us/articles/`

intel-xeon-phi-coprocessor-codename-knights-corner, November 2012.

[39] U. Consortium et al. UPC Language Specifications v1.2. Technical report, Lawerence Berkley National Laboratory, 2005.

[40] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, S. Ramesh, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP'93, pages 1–12, San Diego, California, USA, 1993. ACM, New York, NY, USA.

[41] 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.

[42] L. Dagum and R. Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, January–March 1998.

[43] F. Darema, D. George, V. Norton, and G. Pfister. A Single-Program-Multiple-Data Computational Model for EPEX/FORTRAN. *Parallel Computing*, 7(1):11–24, April 1988.

[44] J. Davis, G. Mudalige, S. Hammond, J. A. Herdman, I. Miller, and S. Jarvis. Predictive Analysis of a Hydrodynamics Application on Large-Scale CMP Clusters. *Computer Science - Research and Development*, 26(3–4):175–185, June 2011.

[45] 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, October 1974.

[46] K. Diefendorff, P. Dubey, R. Hochsprung, and H. Scale. Altivec Extension to PowerPC Accelerates Media Processing. *IEEE Micro*, 20(2):85–95, March/April 2000.

[47] M. Djurfeldt, M. Lundqvist, C. Johansson, M. Rehn, O. Ekeberg, and A. Lansner. Brain-Scale Simulation of the Neocortex on the IBM Blue Gene/L Supercomputer. *IBM Journal of Research and Development*, 52(1.2):31–41, January 2008.

[48] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Z. Jin, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. van der Steen, J. Vetter, P. Williams, R. Wisniewski, , and K. Yelick. The International Exascale Software Project Roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, February 2011.

[49] J. Dongarra and M. A. Heroux. Toward a New Metric for Ranking High Performance Computing Systems. Technical Report SAND2013-4744, Sandia National Laboratories, 2013.

[50] 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, August 2003.

[51] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. UPC Language Specifications v1.1.1. Technical report, Lawerence Berkley National Laboratory, 2003.

[52] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. *IEEE Micro*, 32(3):122–134, March–April 2012.

[53] R. D. Falgout and U. M. Yang. hypre: a Library of High Performance Preconditioners. In *Computational Science - ICCS 2002 Part III*, volume 2331 of *Lecture Notes in Computer Science*, pages 632–641. Springer-Verlag, Berling/Heidelberg, Germany, 2002.

[54] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo. Intel AVX, New Frontiers in Performance Improvements and Energy Efficiency. Technical report, Intel Corporation, 2008.

[55] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948 –960, September 1972.

[56] A. Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. `www.agner.org/optimize/instruction_tables.pdf`, Accessed April 2017.

[57] P. for Advanced Computing in Europe (PRACE). Prace research infrastructure. `http://www.prace-ri.eu/`, Accessed August 2016.

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

[59] M. Forum. Message passing interface (mpi) forum home page. `http://www.mpi-forum.org/`, Accessed June, 2014.

[60] M. I. Frank, A. Agarwal, and M. K. Vernon. LoPC: Modeling Contention in Parallel Algorithms. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP'97, pages 276–287, Las Vegas, Nevada, USA, June 1997. ACM, New York, NY, USA.

[61] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *Lecture Notes in Computer Science*, pages 97–104. Springer, Berlin/Heidelberg, Germany, 2004.

[62] D. Gannon, W. Jalby, and K. Gallivan. Strategies for Cache and Local Memory Management by Global Program Transformation. In *Supercomputing, 1st International Conference Proceedings*, volume 297 of *Lecture Notes in Computer Science*, pages 229–254. Springer, Berlin, Heidelberg, Germany, 1988.

[63] M. W. Gee, C. M. Siefert, J. J. Hu, R. S. Tuminaro, and M. G. Sala. ML 5.0 Smoothed Aggregation User's Guide. Technical Report SAND2006-2649, Sandia National Laboratories, 2006.

[64] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, April 2010.

[65] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press, Cambridge, Massachusetts, USA, 1994.

[66] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN'82, pages 120–126, Boston, Massachusetts, USA, June 1982. ACM, New York, NY, USA.

[67] W. Gropp and E. Lusk. Reproducible Measurements of MPI Performance Characteristics. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1697 of *Lecture Notes in Computer Science*, pages 11–18. Springer, Berlin/Heidelberg, Germany, 1999.

[68] W. Gropp, E. Lusk, N. Doss, and A. Skejellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.

[69] Gropp, William D. and Kaushik, Dinesh K. and Keyes, David E. and Smith, Barry F. Performance Modeling and Tuning of an Unstructured Mesh CFD Application. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC'00, page 34, Dallas, Texas, USA, November 2000. IEEE Computer Society, Los Alaminos, CA, USA, IEEE.

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

[71] S. D. Hammond, G. R. Mudalige, J. A. Smith, S. A. Jarvis, J. A. 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*, SimuTools'09, pages 19:1–19:10, Rome, Italy, March 2009. ICST, Brussels, Belgium.

[72] S. D. Hammond, J. A. Smith, G. R. Mudalige, and S. A. Jarvis. Predictive Simulation of HPC Applications. In *Proceedings of the IEEE 23rd International Conference on Advanced Information Networking and Applications*, AINA'09, pages 33–40, Bradford, UK, May 2009. IEEE Computer Society, Los Alaminos, CA, USA, IEEE.

[73] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short-vector simd architectures. In *Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*, pages 225–245. Springer, Berlin/Heidelberg, Germany, 2011.

[74] 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 the 2006 ACM/IEEE Conference on Supercomputing*, SC'06, pages

3:1–3:10, Tampa, FL, USA, November 2006. IEEE Computer Society, Los Alaminos, CA, USA, IEEE.

[75] A. Hoisie, O. Lubeck, and H. Wasserman. Performance and Scalability Analysis of Teraflop-Scale Parallel Architectures Using Multidimensional Wavefront Applications. *International Journal of High Performance Computing Applications*, 14(4):330–346, November 2000.

[76] A. Hoisie, O. Lubeck, H. J. Wasserman, F. Petrini, and H. Alme. A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs. In *Proceedings of the 2000 International Conference on Parallel Processing*, ICPP'00, pages 219–228, Toronto, Canada, August 2000. IEEE Computer Society, Los Alaminos, CA, USA, IEEE.

[77] P. D. Hovland and L. C. McInnes. Parallel simulation of compressible flow using automatic differentiation and petsc. *Parallel Computing*, 27(4):503–519, March 2001.

[78] C. Iancu, P. Husbands, and P. Hargrove. HUNTing the Overlap. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, PACT'05, pages 279–290, St. Louis, MO, USA, September 2005. IEEE Computer Society, Los Alaminos, CA, USA, IEEE.

[79] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2C:Instruction Set Reference. Technical report, Intel Corporation, 2016.

[80] Intel Corporation. ARK — Your Source for Intel® Product Specifications. `http://ark.intel.com/`, Accessed August 2016.

[81] Intel Corporation. Getting Started with Intel MPI Benchmarks 2017. `https://software.intel.com/en-us/articles/intel-mpi-benchmarks`, Accessed August 2016.

[82] Intel Corporation. Intel MPI Benchmarks. `http://software.intel.com/en-us/articles/intel-mpi-benchmarks`, Accessed July, 2013.

[83] Intel Corporation. Intel MPI Benchmarks 3.2.4 User's Guide. `http://software.intel.com/sites/products/documentation/hpc/ics/imb/32/IMB_Users_Guide/IMB_Users_Guide.pdf`, Accessed July, 2013.

[84] H. W. M. J. Dongarra and E. Strohmaier. TOP500 Supercomputer Sites. *Supercomputer*, 13(2–3):89–111, June 1997.

[85] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, 49(4.5):589–604, July 2005.

[86] R. F. Katz and M. G. Worster. Simulation of Directional Solidification, Thermochemical Convection, and Chimney Formation in a Hele-Shaw Cell. *Journal of Computational Physics*, 227:9823–9840, December 2008.

[87] R. A. Kendall, E. Aprà, D. E. Bernholdt, E. J. Bylaska, M. Dupuis, G. I. Fann, R. J. Harrison, J. Ju, J. A. Nichols, J. Nieplocha, T. Straatsma, T. L. Windus, and A. T. Wong. High Performance Computational Chemistry: An Overview of NWChem, a Distributed Parallel Application. *Computer Physics Communications*, 128(1):260–283, June 2000.

[88] D. J. Kerbyson, A. Hoisie, and H. Wasserman. Modelling the Performance of Large-Scale Systems. *IEE Proceedings — Software*, 150(4):214–221, August 2003.

[89] 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.

[90] D. J. Kerbyson, A. H. J., A. Hoisie, F. Petrini, H. J. Wasserman, and M. L. Gittings. Predictive Performance and Scalability Modeling of a Large-Scale Application. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC'01, pages 37:1–37:12, Denver, CO, USA, November 2001. ACM.

[91] D. E. Keyes, L. C. McInnes, C. Woodward, W. Gropp, E. Myra, M. Pernice, J. Bell, J. Brown, A. Clo, J. Connors, E. Constantinescu, D. Estep, K. Evans, C. Farhat, A. Hakim, G. Hammond, G. Hansen, J. Hill, T. Isaac, X. Jiao, K. Jordan, D. Kaushik, E. Kaxiras, A. Koniges, K. Lee, A. Lott, Q. Lu, J. Magerlein, R. Maxwell, M. McCourt, M. Mehl, R. Pawlowski, A. P. Randles, D. Reynolds, B. Rivire, U. Rde, T. Scheibe, J. Shadid, B. Sheehan, M. Shephard, A. Siegel, B. Smith, X. Tang, C. Wilson, and B. Wohlmuth. Multiphysics Simulations: Challenges and Opportunities. *International Journal of High Performance Computing Applications*, 27(1):4–83, February 2013.

[92] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Mller, and W. E. Nagel. *The Vampir Performance Analysis Tool-Set*, pages 139–155. Springer Berlin, Heidelburg, Germany, July 2008.

[93] D. Koufaty and D. T. Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, 23(2):56–65, March–April 2003.

[94] J. Labarta, S. Girona, and T. Cortes. Analyzing Scheduling Policies Using Dimemas. *Parallel Computing*, 23(1–2):23–34, April 1997.

[95] L. L. N. Laboratory. GitHub – LLNL/IOR Parallel Filesystem I/O Benchmark. `https://github.com/LLNL/ior`, Accessed July, 2016.

[96] R. Landauer. Irreversibility and Heat Generation in the Computing Process. *IBM Journal of Research and Development*, 5(3):183–191, July 1961.

[97] W. Lawry, C. Wilson, A. B. Maccabe, and R. Brightwell. COMB: A Portable Benchmark Suite for Assessing MPI Overlap. In *Proceedings of the 2002 IEEE International Conference on Cluster Computing*, CLUSTER'02, pages 472–475, Chicago, IL, USA, September 2002. IEEE Computer Society, Los Alaminos, CA, USA, IEEE.

[98] C. E. Leiserson. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, October 1985.

[99] B. Lewis, D. Berg, et al. *Multithreaded Programming with Pthreads*. Prentice-Hall Inc, Upper Saddle River, NJ, USA, 1998.

[100] likwid Blog. likwid: Intel Sandybridge and Counting the FLOPs. `likwid-tools.blogspot.co.uk/2012/02/intel-sandybridge-and-counting-flops.html`, Accessed August 2016.

[101] H. Lin, P. Balaji, R. Poole, C. Sosa, X. Ma, and W.-c. Feng. Massively Parallel Genomic Sequence Search on the Blue Gene/P Architecture. In *Procedings of the 2008 ACM/IEEE Conference on High Performance Computing, Networking, Storage and Analysis*, SC'08, pages 33:1–33:11, Austin, Texas, USA, November 2008. IEEE Computer Society,Los Alaminos, CA, USA, IEEE.

[102] C. Lindemann. Performance Modelling with Deterministic and Stochastic Petri Nets. *SIGMETRICS Performance Evaluation Review*, 26(2):3–3, August 1998.

[103] A. Logg. Automating the Finite Element Method. *Archives of Computational Methods in Engineering*, 14(2):93–138, June 2007.

[104] P. Luszczek, D. Bailey, J. Dongarra, J. Kepner, R. Lucas, R. Rabenseifner, and D. Takahashi. The hpc challenge (hpcc) benchmark suite. In *Proceedings of the 2006 ACM/IEEE Conference on High Performance Computing, Networking, Storage and Analysis*, SC'06, page 213, Tampa, Florida, USA, November 2006. ACM, New York, NY, USA.

[105] P. Luszczek, J. J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi. Introduction to the HPC Challenge Benchmark Suite. Technical report, Lawrence Berkley National Laboratory, 2005.

[106] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1):4–15, February 2002.

[107] M. M. Mathis, N. M. Amato, and M. L. Adams. A General Performance Model for Parallel Sweeps on Orthogonal Grids for Particle Transport Calculations. In *Proceedings of the 14th International Conference on Supercomputing*, ICS'00, pages 255–263, Santa Fe, NM, USA, May 2000. ACM.

[108] 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.

[109] M. M. Mathis and D. K. Kerbyson. Performance Modeling of Unstructured Mesh Particle Transport Computations. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, IPDPS'04, pages 245–252, Santa Fe, New Mexico USA, April 2004. IEEE, IEEE Computer Society, Los Alamitos, CA, USA.

[110] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Commitee on Computer Architecture (TCCA) NewsLetter*, pages 19–25, December 1995.

[111] J. D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. `http://www.cs.virginia.edu/stream/`, 1995.

[112] J. D. McCalpin. Preventing FP Overcounts for AVX Instructions on Sandy Bridge. `https://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring/topic/564455`, Accessed August 2016.

[113] S. A. McKee. Reflections on the Memory Wall. In *Proceedings of the First Conference on Computing Frontiers*, CF'04, pages 162–, Ischia, Italy, April 2004. ACM, New York, NY, USA.

[114] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM transactions on Programming Languages and Systems (TOPLAS)*, 18(4):424–453, July 1996.

[115] V. E. McZgee and W. T. Carleton. Piecewise Regression. *Journal of the American Statistical Association*, 65(331):1109–1124, September 1970.

[116] R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[117] M. K. Molloy. Performance Analysis Using Stochastic Petri Nets. *IEEE transactions on Computers*, C-31(9):913–917, September 1982.

[118] G. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114 ff., April 1965.

[119] C. Moritz and M. Frank. Logpg: Modeling network contention in message-passing programs. *IEEE Transactions on Parallel and Distributed Systems*, 12(4):404–415, April 2001.

[120] P. J. Mucci. Llcbench - low level architectural characterization benchmark suite. `http://icl.cs.utk.edu/llcbench`, Accessed August 2016.

[121] P. J. Mucci, K. London, and J. Thurman. The cachebench report, November 1998.

[122] G. R. Mudalige, M. B. Giles, C. Bertolli, and P. H. Kelly. Predictive Modeling and Analysis of OP2 on Distributed Memory GPU Clusters. 40(2):61–67, September 2012.

[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. In *Proceedings of the 2006 IEEE International Conference on Cluster Computing*, CLUSTER'06, pages 1–12, Barcelona, Spain, September 2006. IEEE Computer Society, Los Alaminos, CA, USA, IEEE.

[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 IEEE International Symposium on Parallel and Dis-*

tributed Processing, IPDPS'08, pages 1–14, Miami, Florida, USA, April 2008. IEEE Computer Society, Los Alaminos, CA, USA, IEEE.

[125] A. Munshi. The OpenCL Specification. In *Proceedings of the 2009 IEEE Hot Chips 21 Symposium*, HCS'09, pages 1–314, Stanford, CA, USA, August 2009. IEEE Press, Piscataway, NJ, USA.

[126] R. C. Murphy and P. M. Kogge. On the Memory Access Patterns of Supercomputer Applications: Benchmark Selection and Its Implications. *IEEE Transactions on Computers*, 56(7):937–945, July 2007.

[127] N. A. G. (NAG). How to Make Best Use of the AMD Interlagos Processor. `https://www.nag.co.uk/market/dcse_reports/interlagos_whitepaper.pdf`, Accessed 2016.

[128] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12(1):69–80, January 1996.

[129] Netlib. SPOOLES 2.2 : SParse Object Oriented Linear Equations Solver. `http://www.netlib.org/linalg/spooles/spooles.2.2.html`, Accessed July, 2013.

[130] B. Nichols, D. Buttlar, and J. Farrell. *Pthreads programming: A POSIX standard for better multiprocessing.* O'Reilly Media, Inc., Sebastopol, CA, USA, 1996.

[131] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, March 2008.

[132] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox. Pace - a toolset for the performance prediction of parallel and distributed systems. *International Journal of High Performance Computing Applications*, 14(3):228–251, August 2000.

[133] R. W. Numrich and J. Reid. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[134] NVIDIA. CUDA Parallel Computing — What is CUDA? — NVIDIA UK. `http://www.nvidia.co.uk/object/cuda-parallel-computing-uk.html`, Accessed September, 2016.

[135] U. of Edinburgh HPcx Ltd. What is HECToR and why is it special? `http://www.hector.ac.uk/abouthector/hectorbasics`, Accessed July, 2013.

[136] K. Olukotun and L. Hammond. The Future of Microprocessors. *Queue*, 3(7):26–29, September 2005.

[137] OpenACC. Openacc home. `www.openacc-standard.org`, Accessed August 2016.

[138] PAPI. Counting floating point operations on intel sandy bridge and ivy bridge. `http://icl.cs.utk.edu/projects/wiki/PAPITopics:SAndyFlops`, Accessed August 2016.

[139] 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 IEEE International Parallel & Distributed Processing Symposium*, IPDPS'13, pages 1085–1097. IEEE Computer Society, Los Alaminos, CA, USA, IEEE, May 2013.

[140] F. Petrini, G. Fossum, J. Fernandez, A. L. Varbanescu, M. Kistler, and M.Perrone. Multicore Surprises: Lessons Learned from Optimizing Sweep3D on the Cell Broadband Engine. In *Proceedigns of the 2007 IEEE International Parallel and Distributed Processing Symposium*, IPDPS'07, pages 1–10, California, USA, March 2007. IEEE Press, Piscataway, NJ, USA.

[141] 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 the 2003 ACM/IEEE Conference on Supercomputing*, SC'03, pages 55:1–55:17, Phoenix, AZ, USA, November 2003. ACM, New York, NY, USA.

[142] V. Pillet, J. Labarta, T. Cortes, and S. Girona. Paraver: A Tool to Visualize and Analyze Parallel Code. In *Proceedings of WoTUG-18: Transputer and occam Developments*, WoTUG-18, pages 17–31, Manchester, UK, March 1995. IOs Press, Amsterdam, Netherlands.

[143] S. Potluri, P. Lai, K. A. Tomko, S. Sur, Y. Cui, M. Tatineni, K. W. Schulz, W. L. Barth, A. Majumdar, and D. K. Panda. Quantifying Performance Benefits of Overlap Using MPI-2 in a Seismic Modeling Application. In *Proceedings of the 24th International Conference on Supercomputing (ICS)*, ICS'10, pages 17–25, Tsukuba, Ibaraki, Japan, June 2010. ACM, New York, NY, USA.

[144] S. Prakash and R. L. Bagrodia. MPI-SIM: Using Parallel Simulation to Evaluate MPI Programs. In *Proceedings of the 30th Conference on Winter*

Simulation, WSC'98, pages 467–474, Washington, D.C, USA, December 1998. IEEE, IEEE Computer Society, Los Alamitos, CA, USA.

[145] S. Pronk, S. Páll, R. Schulz, P. Larsson, P. Bjelkmar, R. Apostolov, M. R. Shirts, J. C. Smith, P. M. Kasson, D. van der Spoel, B. Hess, and E. Lindahl. GROMACS 4.5: A High-Throughput and Highly Parallel Open Source Molecular Simulation Toolkit. *Bioinformatics*, 29(7):845–854, April 2013.

[146] M. J. Puckelwartz, L. L. Pesce, V. Nelakuditi, L. Dellefave-Castillo, J. R. Golbus, S. M. Day, T. P. Cappola, G. W. Dorn II, I. T. Foster, and E. M. McNally. Supercomputing for the Parallelization of Whole Genome Analysis. *Bioinformatics*, 30(11):1508–1513, June 2014.

[147] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, PDP'09, pages 427–436, Weimar, Germany, February 2009. IEEE Computer Society, Los Alamitos, CA, USA, IEEE.

[148] C. V. Ramamoorthy and G. S. Ho. Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets. *IEEE Transactions on Software Engineering*, SE-6(5):440–449, September 1980.

[149] M. J. Rashti and A. Afsahi. Assessing the Ability of Computation/Communication Overlap and Communication Progress in Modern Interconnects. In *Proceedings of the 15th Annual IEEE Symposium on High-Performance Interconnects*, HOTI'07, pages 117–124, Stanford, CA, USA, August 2007. IEEE, Computer Society, Los Alaminos, CA, USA, IEEE.

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

[151] 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/Heidelberg, Germany, 1998.

[152] Y. Saad and M. H. Schultz. Topological Properties of Hypercubes. *IEEE Transactions on Computers*, 37(7):867–872, July 1988.

[153] J. C. Sancho, K. J. Barker, D. J. Kerbyson, and K. Davis. Quantifying the Potential Benefit of Overlapping Communication and Computation

in Large-Scale Scientific Applications. In *Proceedings of the ACM/IEEE 2006 Conference on High Performance Networking and Computing*, SC'06, page 17, Tampa, Florida, USA, November 2006. IEEE Computer Society, Los Alaminos, CA, USA, IEEE.

[154] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. volume 27, pages 18:1–18:15. ACM, New York, NY, USA, August 2008.

[155] J. Shalf, S. Dosanjh, and J. Morrison. Exascale Computing Technology Challenges. In *High Performance Computing for Computational Science VECPAR 2010*, volume 6449 of *Lecture Notes in Computer Science*, pages 1–25. Springer, Berlin/Heidelberg, Germany, 2011.

[156] H. Shan and J. Shalf. Using IOR to Analyze the I/O Performance for HPC Platforms. Technical report, Lawrence Berkley National Laboratory, 2007.

[157] 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.

[158] D. Spooner, S. Jarvis, J. Cao, S. Saini, and G. Nudd. Local Grid Scheduling Techniques using Performance Prediction. *IEE Proceedings — Computers and Digital Techniques*, 150(2):87–96, March 2003.

[159] T. L. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A Parallel Workstation For Scientific Computation. In *Proceedings of the 24th International Conference on Parallel Processing*, ICPP'95, pages 11–14, Urbana-Champain, Illinois, USA, August 1995. CRC Press, Boca Raton, FL, USA.

[160] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66–73, May–June 2010.

[161] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer. Top500 supercomputer sites. `https://www.top500.org/`, Accessed August 2016.

[162] V. Subotic, J. C. Sancho, J. Labarta, and M. Valero. The Impact of Application's Micro-Imbalance on the Communication-Computation Overlap. In *Proceedings of the 19th International Euromicro Conference on*

Parallel, Distributed and Network-Based Processing, PDP'11, pages 191–198, Ayia Napa, Cyprus, February 2011. IEEE Computer Society,Los Alaminos, CA, USA, IEEE.

[163] D. Sundaram-Stukel and M. K. Vernon. Predictive Analysis of a Wavefront Application using LogGP. In *Proceedings of the 1999 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP'99, pages 141–150, Atlanta, Georgia, USA, May 1999. ACM, New York, NY, USA.

[164] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, March 2005.

[165] H. Sutter and J. Larus. Software and the Concurrency Revolution. *Queue*, 3(7):54–62, September 2005.

[166] V. Taylor, X. Wu, and R. Stevens. Prophesy: An Infrastructure for Performance Analysis and Modeling of Parallel and Grid Applications. *ACM SIGMETRICS Performance Evaluation Review*, 30(4):13–18, March 2003.

[167] S. Team. Skampi webpage. `http://liinwww.ira.uka.de/~skampi/`, Accessed August 2016.

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

[169] J. Weinberg, M. O. McCracken, E. Strohmaier, and A. Snavely. Quantifying Locality In The Memory Access Patterns of HPC Applications. In *Proceedings of the 2005 ACM/IEEE Conference on High Performance Networking and Computing*, SC'05, pages 50–, Seattle, WA, USA, November 2005. IEEE Computer Society, Los Alaminos, CA, USA, IEEE.

[170] J. White III and S. Bova. Where's the overlap? an analysis of popular mpi implementations. Number CEWES MSRC/PET TR/99-09, 1999.

[171] J. B. White III and J. J. Dongarra. Overlapping Computation and Communication for Advection on Hybrid Parallel Computers. In *Proceedings of the 25th IEEE International Symposium on Parallel and Distributed Processing*, IPDPS'11, pages 59–67, Anchorage, Alaska, May 2011. IEEE Computer Society, Los Alaminos, CA, USA, IEEE.

[172] M. V. Wilkes. The Memory Wall and the CMOS End-Point. *SIGARCH Computer Architecture News*, 23(4):4–6, September 1995.

[173] 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.

[174] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford University, 1992.

[175] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining Loop Transformations Considering Caches and Scheduling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitectures*, MICRO 29, pages 274–286, Paris, France, December 1996. IEEE, IEEE Computer Society, Los Alamitos, CA, USA.

[176] S. A. Wright, S. D. Hammond, S. J. Pennycook, I. Miller, J. A. Herdman, and S. A. Jarvis. LDPLFS: Improving I/O Performance without Application Modification. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, IPDPSW'12, pages 1352–1359, Shanghai, China, May 2012. IEEE Computer Society, Los Alamitos, CA, USA, IEEE.

[177] S. A. Wright and S. A. Jarvis. Quantifying the Effects of Contention on Parallel File Systems. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, IPDPSW'15, pages 932–940, Hyderabad, India, May 2015. IEEE Computer Society, Los Alamitos, CA, USA, IEEE.

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

APPENDIX A

Figure Data

Figure	Machine	Compiler	MPI	Data Tables
1.1	N/A	N/A	N/A	A.3
3.1(a)	Minerva	Intel v12.0	N/A	A.7
3.1(b)	Minerva	Intel v12.0	N/A	A.4
3.2(a)	Minerva	Intel v12.0	OpenMPI v1.4.3	A.6
3.2(b)	Minerva	Intel v12.0	OpenMPI v1.4.3	A.5
3.3	HECToR	PGI v12.10	Cray MPI v5.6.1	A.8
3.4(a)	HECToR	PGI v12.10	Cray MPI v5.6.1	A.10
3.4(b)	HECToR	PGI v12.10	Cray MPI v5.6.1	A.9
3.5	DawnDev	IBM XL v11.0	IBM BlueGene MPI	A.12
3.6(a)	Hera	PGI v8.0	OpenMPI v1.3.2	A.11
3.6(b)	Hera	PGI v8.0	OpenMPI v1.3.2	A.11
4.3	Minerva	Intel v12.0	OpenMPI v1.4.3	A.13
4.4	Minerva	Intel v12.0	OpenMPI v1.4.3	A.14
4.5	Minerva	Intel v12.0	OpenMPI v1.4.3	A.15–A.17
4.6	Minerva	Intel v12.0	OpenMPI v1.4.3	A.18
4.7(a)	DawnDev	IBM XL v11.0	IBM BlueGene MPI	A.19
4.7(b)	Hera	PGI v8.0	OpenMPI v1.3.2	A.19
4.8	Minerva	Intel v12.0	OpenMPI v1.4.3	A.20
4.9	Minerva	Intel v12.0	OpenMPI v1.4.3	A.21–A.23
5.1	Minerva	Intel v12.0	OpenMPI v1.4.3	—
5.2	Minerva	Intel v12.0	OpenMPI v1.4.4	A.24
5.3	Minerva	Intel v12.0	OpenMPI v1.4.4	A.24
5.4	Minerva	Intel v12.0	OpenMPI v1.4.3	—
5.5	Minerva	Intel v12.0	OpenMPI v1.4.3	—
5.6	Minerva	Intel v12.0	OpenMPI v1.4.3	—
5.7	Minerva	Intel v12.0	OpenMPI v1.4.3	—
6.1	Minerva/ Intel X3430	Intel v12.0	N/A	A.27–A.32
6.2	Minerva/ Intel X3430	Intel v12.0	N/A	A.27–A.32
6.3	Minerva/ Intel X3430	Intel v12.0	OpenMPI v1.4.3	A.27–A.32, A.39–A.44, A.45–A.50
6.4	Intel X3430 Minerva	Intel v12.0	OpenMPI v1.4.3	A.51–A.52
6.5	Minerva/ Intel X3430	Intel v12.0	N/A	A.53–A.59
6.6	Minerva	Intel v12.0	OpenMPI v1.4.3	A.52, A.60
6.7	Minerva	Intel v12.0	OpenMPI v1.4.3	A.61, A.62
6.8	Minerva	Intel v12.0	OpenMPI-1.4.4	A.63
6.9	Minerva	Intel v12.0	OpenMPI-1.4.4	A.64
6.11	Minerva	Intel v12.0	OpenMPI-1.4.4	A.65, A.66
6.12	Minerva	Intel v12.0	OpenMPI-1.4.4	A.67–A.72
7.3	HECToR	PGI v12.10	Cray MPI v5.6.1	A.73
7.4	HECToR	PGI v12.10	Cray MPI v5.6.1	A.74
7.5	HECToR	PGI v12.10	Cray MPI v5.6.1	A.75
7.6	HECToR	PGI v12.10	Cray MPI v5.6.1	A.76
7.7(a)	HECToR	PGI v12.10	Cray MPI v5.6.1	A.77
7.7(b)	HECToR	PGI v12.10	Cray MPI v5.6.1	A.77
7.8	HECToR	PGI v12.10	Cray MPI v5.6.1	A.78
7.9	HECToR	PGI v12.10	Cray MPI v5.6.1	A.79, A.80
7.10	HECToR	PGI v12.10	Cray MPI v5.6.1	A.81, A.82

Table A.1: Experimental Parameters by Figure

Figure	Machine	Compiler	MPI
3.2	Minerva	Intel v12.0	OpenMPI v1.4.3
3.4	DawnDev	IBM XL v11.0	IBM BlueGene MPI
3.5	Hera	PGI v8.0	OpenMPI v1.3.2
4.3	Minerva	Intel v12.0	N/A
4.4	Minerva	Intel v12.0	OpenMPI v1.4.3
4.6	DawnDev	IBM XL v11.0	IBM BlueGene MPI
4.6	Hera	PGI v8.0	OpenMPI v1.3.2
4.8	Minerva	Intel v12.0	OpenMPI v1.4.3
4.8	Minerva	Intel v12.0	OpenMPI v1.4.3
5.7	Minerva	Intel v12.0	OpenMPI v1.4.3
5.9	Minerva	Intel v12.0	OpenMPI v1.4.3
5.10	DawnDev	IBM XL v11.0	IBM BlueGene MPI
5.10	Hera	PGI v8.0	OpenMPI v1.3.2
5.11	DawnDev	IBM XL v11.0	IBM BlueGene MPI
5.11	Hera	PGI v8.0	OpenMPI v1.3.2
5.12	Minerva	Intel v12.0	OpenMPI v1.4.3
5.13	Minerva	Intel v12.0	OpenMPI v1.4.3
5.14	Minerva	Intel v12.0	OpenMPI v1.4.3
6.4	Minerva	Intel v12.0	N/A
6.5	X3430	Intel v12.0	N/A
6.6	Minerva	Intel v12.0	OpenMPI v1.4.3
7.1	HECToR	PGI v12.10	Cray MPI v5.6.1
7.2	HECToR	PGI v12.10	Cray MPI v5.6.1
7.1	HECToR	PGI v12.10	Cray MPI v5.6.1
7.2	HECToR	PGI v12.10	Cray MPI v5.6.1

Table A.2: Experimental Parameters by Table

Date	Cores	Performance (GFLOP/s)		Efficiency(%)
		Linpack	Theoretical	
1993-06-01	1024	0.06	0.13	45.57
1993-11-01	140	0.12	0.24	52.59
1994-06-01	3680	0.14	0.18	77.93
1994-11-01	140	0.17	0.24	72.09
1995-06-01	140	0.17	0.24	72.09
1995-11-01	140	0.17	0.24	72.09
1996-06-01	1024	0.22	0.31	71.74
1996-11-01	2048	0.37	0.61	59.93
1997-06-01	7264	1.07	1.45	73.50
1997-11-01	9152	1.34	1.83	73.10
1998-06-01	9152	1.34	1.83	73.10
1998-11-01	9152	1.34	1.83	73.10
1999-06-01	9472	2.12	3.15	67.25
1999-11-01	9632	2.38	3.21	74.18
2000-06-01	9632	2.38	3.21	74.18
2000-11-01	8192	4.94	12.29	40.19
2001-06-01	8192	7.23	12.29	58.81
2001-11-01	8192	7.23	12.29	58.81
2002-06-01	5120	35.86	40.96	87.55
2002-11-01	5120	35.86	40.96	87.55
2003-06-01	5120	35.86	40.96	87.55
2003-11-01	5120	35.86	40.96	87.55
2004-06-01	5120	35.86	40.96	87.55
2004-11-01	32 768	70.72	91.75	77.08
2005-06-01	65 536	136.80	183.50	74.55
2005-11-01	131 072	280.60	367.00	76.46
2006-06-01	131 072	280.60	367.00	76.46
2006-11-01	131 072	280.60	367.00	76.46
2007-06-01	131 072	280.60	367.00	76.46
2007-11-01	212 992	478.20	596.40	80.18
2008-06-01	122 400	1026.00	1375.80	74.57
2008-11-01	129 600	1105.00	1456.70	75.86
2009-06-01	129 600	1105.00	1456.70	75.86
2009-11-01	224 162	1759.00	2331.00	75.46
2010-06-01	224 162	1759.00	2331.00	75.46
2010-11-01	186 368	2566.00	4701.00	54.58
2011-06-01	548 352	8162.00	8773.60	93.03
2011-11-01	705 024	10 510.00	11 280.40	93.17
2012-06-01	1 572 864	16 324.80	20 132.70	81.09
2012-11-01	560 640	17 590.00	27 112.50	64.88
2013-06-01	3 120 000	33 862.70	54 902.40	61.68
2013-11-01	3 120 000	33 862.70	54 902.40	61.68
2014-06-01	3 120 000	33 862.70	54 902.40	61.68
2014-11-01	3 120 000	33 862.70	54 902.40	61.68
2015-06-01	3 120 000	33 862.70	54 902.40	61.68
2015-11-01	3 120 000	33 862.70	54 902.40	61.68
2016-06-01	10 649 600	93 014.60	125 435.90	74.15

Table A.3: Top 500 Max/Peak Performance, June 1993 - June 2016 - Data for Figure 1.1

PEs	Bandwidth (GB/s)			
	Copy	Scale	Add	Triad
1	10.05	10.55	11.81	11.99
2	15.28	15.70	18.18	18.52
3	23.41	22.47	25.92	26.31
4	31.59	28.54	37.46	32.10
5	28.01	27.18	32.19	31.76
6	24.62	26.32	28.92	31.00
7	29.93	30.63	34.90	35.03
8	33.85	31.00	39.82	34.96
9	30.13	27.41	34.98	33.63
10	29.18	29.45	35.35	34.49
11	31.85	27.20	34.52	36.04
12	31.63	27.78	29.70	30.15

Table A.4: STREAM – Data for Figure 3.1(b)

PEs	Time (s)
2	9.80E-7
4	1.94E-6
8	3.96E-6
16	5.90E-6
32	9.22E-6
128	1.52E-5

Table A.5: IMB AllReduce Time, 4 Bytes – Data for Figure 3.2(b)

Message Size (Bytes)	Time (s)		Message Size (Bytes)	Time (s)	
	Intra-Node	Inter-Node		Intra-Node	Inter-Node
1	4.70E-07	1.76E-06	241664	3.29E-05	1.12E-04
2	4.80E-07	1.74E-06	245760	3.37E-05	1.13E-04
4	4.70E-07	1.78E-06	249856	3.40E-05	1.13E-04
8	4.80E-07	1.76E-06	253952	3.47E-05	1.14E-04
16	4.60E-07	2.01E-06	258048	3.53E-05	1.14E-04
32	4.20E-07	2.01E-06	262144	3.55E-05	1.16E-04
64	6.20E-07	2.01E-06	266240	3.64E-05	1.18E-04
128	6.50E-07	2.04E-06	270336	3.66E-05	1.20E-04
256	6.40E-07	2.15E-06	274432	3.72E-05	1.21E-04
512	6.60E-07	2.48E-06	278528	3.76E-05	1.22E-04
1024	7.90E-07	2.82E-06	282624	3.81E-05	1.23E-04
2048	1.01E-06	3.64E-06	286720	3.87E-05	1.25E-04
4096	1.33E-06	4.68E-06	290816	3.92E-05	1.25E-04
8192	2.02E-06	6.69E-06	294912	3.96E-05	1.27E-04
12 288	2.66E-06	8.52E-06	299008	4.01E-05	1.28E-04
16 384	4.49E-06	1.04E-05	303104	4.08E-05	1.30E-04
20 480	5.07E-06	1.21E-05	307200	4.11E-05	1.31E-04
24 576	5.50E-06	1.39E-05	311296	4.14E-05	1.33E-04
28 672	5.91E-06	1.58E-05	315392	4.20E-05	1.34E-04
32 768	6.45E-06	1.75E-05	319488	4.25E-05	1.34E-04
36 864	6.75E-06	2.71E-05	323584	4.30E-05	1.36E-04
40 960	7.30E-06	2.94E-05	327680	4.38E-05	1.43E-04
45 056	7.72E-06	3.16E-05	331776	4.43E-05	1.42E-04
49 152	8.16E-06	3.39E-05	335872	4.47E-05	1.45E-04
53 248	8.67E-06	3.60E-05	339968	4.51E-05	1.45E-04
57 344	9.15E-06	3.83E-05	344064	4.59E-05	1.46E-04
61 440	9.42E-06	4.07E-05	348160	4.53E-05	1.46E-04
65 536	9.98E-06	4.96E-05	352256	4.28E-05	1.47E-04
69 632	1.05E-05	5.15E-05	356352	4.31E-05	1.47E-04
73 728	1.08E-05	5.39E-05	360448	4.37E-05	1.48E-04
77 824	1.14E-05	5.57E-05	364544	4.42E-05	1.48E-04
81 920	1.20E-05	5.79E-05	368640	4.48E-05	1.51E-04
86 016	1.23E-05	6.01E-05	372736	4.48E-05	1.52E-04
90 112	1.30E-05	6.00E-05	376832	4.55E-05	1.54E-04
94 208	1.34E-05	6.18E-05	380928	4.55E-05	1.54E-04
98 304	1.39E-05	6.36E-05	385024	4.63E-05	1.56E-04
102 400	1.45E-05	6.56E-05	389120	4.67E-05	1.56E-04
106 496	1.51E-05	6.75E-05	393216	4.73E-05	1.57E-04
110 592	1.53E-05	6.95E-05	397312	4.78E-05	1.58E-04
114 688	1.60E-05	7.14E-05	401408	4.81E-05	1.59E-04
118 784	1.66E-05	7.35E-05	405504	4.85E-05	1.60E-04
122 880	1.71E-05	7.60E-05	409600	4.90E-05	1.61E-04
126 976	1.79E-05	7.78E-05	413696	4.94E-05	1.62E-04
131 072	1.83E-05	7.83E-05	417792	5.00E-05	1.64E-04
135 168	1.92E-05	7.73E-05	421888	5.03E-05	1.65E-04
139 264	2.00E-05	7.85E-05	425984	5.09E-05	1.66E-04
143 360	2.03E-05	7.97E-05	430080	5.12E-05	1.67E-04
147 456	2.10E-05	8.09E-05	434176	5.16E-05	1.69E-04
151 552	2.12E-05	8.21E-05	438272	5.22E-05	1.70E-04
155 648	2.18E-05	8.33E-05	442368	5.26E-05	1.71E-04
159 744	2.23E-05	8.45E-05	446464	5.30E-05	1.72E-04
163 840	2.30E-05	8.60E-05	450560	5.35E-05	1.74E-04
167 936	2.32E-05	8.74E-05	454656	5.38E-05	1.75E-04
172 032	2.42E-05	8.82E-05	458752	5.44E-05	1.86E-04
176 128	2.44E-05	8.98E-05	462848	5.46E-05	1.87E-04
180 224	2.54E-05	9.10E-05	466944	5.53E-05	1.88E-04
184 320	2.54E-05	9.21E-05	471040	5.56E-05	1.89E-04
188 416	2.64E-05	9.35E-05	475136	5.62E-05	1.90E-04
192 512	2.64E-05	9.48E-05	479232	5.66E-05	1.91E-04
196 608	2.76E-05	1.03E-04	483328	5.71E-05	1.92E-04
200 704	2.76E-05	1.04E-04	487424	5.75E-05	1.92E-04
204 800	2.84E-05	1.05E-04	491520	5.79E-05	1.92E-04
208 896	2.87E-05	1.05E-04	495616	5.83E-05	1.94E-04
212 992	2.93E-05	1.06E-04	499712	5.87E-05	1.94E-04
217 088	2.97E-05	1.06E-04	503808	5.91E-05	1.97E-04
221 184	3.03E-05	1.06E-04	507904	5.97E-05	1.98E-04
225 280	3.07E-05	1.06E-04	512000	6.00E-05	1.99E-04
229 376	3.16E-05	1.06E-04	516096	6.04E-05	2.02E-04
233 472	3.19E-05	1.09E-04	520192	6.09E-05	2.02E-04
237 568	3.26E-05	1.11E-04	524288	6.15E-05	2.02E-04

Table A.6: IMB PingPong Intra/Inter-Node — Figure 3.2(a)

Bytes	Bandwidth (GB/s)				
	Read	**Write**	**RWM**	**MemSet**	**MemCopy**
256	20.90	35.52	54.65	19.73	46.04
336	20.87	40.00	54.85	25.94	54.85
424	19.86	36.71	49.44	31.06	62.12
512	22.50	45.71	63.60	33.25	63.72
680	22.33	41.33	56.31	34.69	71.96
848	22.64	44.05	64.53	37.56	74.55
1024	22.66	45.71	68.03	37.51	72.69
1360	22.72	44.66	67.56	40.06	80.12
1704	22.75	44.64	65.34	40.57	82.52
2048	22.76	45.71	70.40	41.21	82.69
2728	22.79	45.04	67.51	42.36	85.65
3408	22.80	45.08	70.81	42.92	86.55
4096	22.81	45.71	71.79	42.79	86.83
5456	22.33	43.42	59.04	44.16	88.32
6824	22.36	43.42	58.89	44.31	89.03
8192	22.50	44.16	59.70	44.20	88.68
10 920	22.54	44.25	59.64	43.39	87.02
13 648	22.64	44.77	60.17	43.91	65.84
16 384	22.68	44.92	60.31	44.03	60.99
21 840	22.72	45.11	62.24	44.57	34.78
27 304	22.73	45.11	60.96	44.75	34.41
32 768	22.23	40.73	60.51	34.47	34.34
43 688	15.61	27.82	43.50	27.89	34.60
54 608	15.64	27.56	43.41	27.74	35.18
65 536	15.64	27.58	43.42	27.62	34.44
87 376	15.67	27.61	43.48	27.10	35.27
109 224	15.65	27.41	43.48	27.11	34.49
131 072	15.71	27.30	43.22	27.06	31.91
174 760	15.66	26.97	39.25	27.06	28.38
218 448	15.18	26.61	36.16	26.73	23.33
262 144	14.96	24.44	34.08	25.28	22.18
349 520	14.58	20.98	30.33	22.31	22.08
436 904	14.49	18.94	28.67	18.96	22.15
524 288	14.43	17.77	27.26	17.23	22.17
699 048	14.43	16.83	26.63	16.76	22.06
873 808	14.45	16.77	26.66	16.77	22.15
1 048 576	14.47	16.77	26.66	16.77	22.19
1 398 096	14.50	16.78	26.68	16.77	22.17
1 747 624	14.51	16.78	26.68	16.78	22.19
2 097 152	14.53	16.78	26.69	16.78	22.19
2 796 200	14.54	16.77	26.70	16.78	22.20
3 495 248	14.55	16.78	26.71	16.78	22.18
4 194 304	14.56	16.78	26.72	16.78	22.20
5 592 400	14.56	16.78	26.72	16.78	20.70
6 990 504	14.57	16.78	26.72	16.78	13.40
8 388 608	14.57	16.78	26.69	16.78	10.30
11 184 808	13.87	14.86	24.25	14.68	9.15
13 981 008	11.68	9.98	18.44	7.30	8.80
16 777 216	10.45	8.17	16.48	7.30	8.94
22 369 616	10.10	7.61	15.01	7.30	8.89
27 962 024	9.94	7.48	14.63	7.31	8.70
33 554 432	9.96	7.32	14.59	7.30	8.77
44 739 240	9.96	7.16	14.64	7.23	8.73
55 924 048	9.96	7.26	14.42	7.30	8.83
67 108 864	9.96	7.27	14.45	7.30	8.84
89 478 480	9.96	7.27	14.35	7.28	8.78
111 848 104	9.96	7.26	14.34	7.30	8.79
134 217 728	9.96	7.27	14.36	7.30	8.76
178 956 968	9.96	7.26	14.36	7.31	8.77
223 696 208	9.96	7.27	14.19	7.31	8.80
268 435 456	9.96	7.30	14.21	7.31	8.79
357 913 936	9.96	7.30	14.13	7.30	8.77
447 392 424	9.96	7.26	14.09	7.30	8.74
536 870 912	9.96	7.23	14.05	7.30	8.75

Table A.7: CacheBench – Data for Figure 3.1(a)

PEs	Bandwidth (GB/s)			
	Copy	Scale	Add	Triad
1	11.80	5.56	6.16	5.79
2	12.27	6.79	7.55	7.45
3	14.45	8.04	8.89	8.92
4	16.16	9.06	9.87	9.94
5	15.92	9.40	9.81	9.86
6	16.27	9.94	10.04	10.09
7	16.42	10.03	9.94	10.09
8	16.52	9.91	9.77	9.91
9	18.87	11.37	11.07	11.36
10	21.29	12.73	12.18	12.51
11	23.64	14.17	13.47	13.93
12	26.06	15.47	14.32	14.87
13	29.01	17.26	15.54	16.19
14	32.24	19.55	17.06	17.35
15	34.95	21.78	18.57	19.04
16	36.89	24.47	20.14	20.44
17	40.01	27.56	21.52	21.48
18	42.06	31.07	23.25	21.99
19	44.57	35.36	25.37	22.94
20	47.49	40.32	27.94	25.71
21	49.52	45.96	31.46	28.32
22	52.35	51.06	34.88	31.97
23	52.89	59.20	40.93	38.77
24	53.41	66.20	49.13	48.27
25	53.24	72.18	59.27	64.04
26	53.15	79.53	71.57	72.90
27	54.25	87.56	82.50	85.86
28	52.87	92.96	93.40	99.95
29	54.84	98.92	105.59	108.03
30	56.52	104.86	113.25	116.89
31	56.72	110.42	120.77	122.73
32	58.75	113.98	130.20	129.52

Table A.8: STREAM – Data for Figure 3.3

PEs	Time (s)
2	1.20E-06
4	2.29E-06
8	3.89E-06
16	5.26E-06
32	6.65E-06
64	1.01E-05
128	1.94E-05
256	1.59E-05
512	3.00E-05
1024	3.58E-05
2048	3.83E-05
4096	1.20E-04
8192	9.84E-05
16 384	1.04E-04
32 768	7.10E-05
65 336	9.48E-05

Table A.9: IMB AllReduce, 8 Bytes – Data for Figure 3.4(b)

Message Size (Bytes)	Time (s)		Message Size (Bytes)	Time (s)	
	Intra-Node	Inter-Node		Intra-Node	Inter-Node
1	2.80E-07	1.64E-06	241664	3.66E-05	5.24E-05
2	2.90E-07	1.68E-06	245760	3.72E-05	5.27E-05
4	2.90E-07	1.70E-06	249856	3.75E-05	5.39E-05
8	3.40E-07	1.67E-06	253952	3.84E-05	5.41E-05
16	3.50E-07	1.69E-06	258048	3.90E-05	5.54E-05
32	3.40E-07	1.69E-06	262144	3.93E-05	5.59E-05
64	3.50E-07	1.72E-06	266240	4.03E-05	5.74E-05
128	3.60E-07	1.80E-06	270336	4.64E-05	5.77E-05
256	4.10E-07	1.81E-06	274432	4.12E-05	5.88E-05
512	5.40E-07	1.97E-06	278528	4.22E-05	5.87E-05
1024	6.30E-07	2.20E-06	282624	4.28E-05	6.02E-05
2048	8.30E-07	2.65E-06	286720	4.31E-05	6.07E-05
4096	1.27E-06	3.44E-06	290816	4.39E-05	6.18E-05
8192	1.46E-06	8.67E-06	294912	4.44E-05	8.56E-05
12 288	2.19E-06	1.03E-05	299008	4.47E-05	6.39E-05
16 384	2.78E-06	1.06E-05	303104	4.59E-05	6.98E-05
20 480	3.42E-06	1.20E-05	307200	4.63E-05	8.11E-05
24 576	4.07E-06	1.23E-05	311296	4.64E-05	8.21E-05
28 672	4.66E-06	1.33E-05	315392	4.76E-05	6.72E-05
32 768	5.34E-06	1.55E-05	319488	4.83E-05	6.63E-05
36 864	5.89E-06	1.53E-05	323584	4.90E-05	7.02E-05
40 960	6.46E-06	1.62E-05	327680	4.93E-05	6.77E-05
45 056	7.15E-06	2.21E-05	331776	4.98E-05	6.88E-05
49 152	7.67E-06	1.92E-05	335872	5.19E-05	6.90E-05
53 248	8.31E-06	1.90E-05	339968	5.12E-05	1.11E-04
57 344	8.96E-06	2.20E-05	344064	5.19E-05	9.18E-05
61 440	9.47E-06	2.18E-05	348160	5.24E-05	7.37E-05
65 536	1.02E-05	2.17E-05	352256	5.38E-05	7.49E-05
69 632	1.09E-05	2.82E-05	356352	5.43E-05	7.75E-05
73 728	1.15E-05	2.75E-05	360448	5.45E-05	7.51E-05
77 824	1.21E-05	2.14E-05	364544	5.57E-05	7.61E-05
81 920	1.26E-05	2.94E-05	368640	5.58E-05	7.54E-05
86 016	1.33E-05	3.12E-05	372736	5.65E-05	7.79E-05
90 112	1.37E-05	2.41E-05	376832	5.71E-05	7.83E-05
94 208	1.45E-05	3.29E-05	380928	5.76E-05	9.40E-05
98 304	1.50E-05	3.42E-05	385024	5.89E-05	7.83E-05
102 400	1.57E-05	2.59E-05	389120	5.91E-05	8.00E-05
106 496	1.63E-05	3.77E-05	393216	6.15E-05	9.22E-05
110 592	1.69E-05	3.81E-05	397312	6.08E-05	1.35E-04
114 688	1.75E-05	2.83E-05	401408	6.25E-05	8.15E-05
118 784	1.81E-05	4.14E-05	405504	6.26E-05	8.27E-05
122 880	1.88E-05	5.17E-05	409600	6.29E-05	1.08E-04
126 976	2.04E-05	3.08E-05	413696	7.26E-05	8.36E-05
131 072	1.97E-05	4.32E-05	417792	6.55E-05	1.10E-04
135 168	2.07E-05	3.23E-05	421888	6.64E-05	8.43E-05
139 264	2.12E-05	3.28E-05	425984	7.65E-05	8.73E-05
143 360	2.17E-05	3.40E-05	430080	6.69E-05	8.71E-05
147 456	2.25E-05	3.52E-05	434176	6.94E-05	8.98E-05
151 552	2.29E-05	3.63E-05	438272	6.93E-05	8.65E-05
155 648	2.37E-05	3.61E-05	442368	7.02E-05	8.93E-05
159 744	2.44E-05	3.70E-05	446464	7.11E-05	8.85E-05
163 840	2.47E-05	3.76E-05	450560	7.01E-05	8.99E-05
167 936	2.54E-05	3.85E-05	454656	7.25E-05	8.92E-05
172 032	2.62E-05	3.88E-05	458752	7.27E-05	9.09E-05
176 128	2.67E-05	3.96E-05	462848	7.50E-05	9.12E-05
180 224	2.71E-05	4.05E-05	466944	8.01E-05	9.32E-05
184 320	2.80E-05	4.16E-05	471040	7.61E-05	9.17E-05
188 416	2.86E-05	4.20E-05	475136	7.50E-05	9.40E-05
192 512	2.88E-05	4.28E-05	479232	7.79E-05	9.19E-05
196 608	2.98E-05	4.34E-05	483328	7.96E-05	9.44E-05
200 704	3.05E-05	4.77E-05	487424	8.28E-05	9.38E-05
204 800	3.09E-05	8.70E-05	491520	8.31E-05	9.83E-05
208 896	3.13E-05	6.24E-05	495616	8.38E-05	9.80E-05
212 992	4.60E-05	5.04E-05	499712	8.49E-05	1.52E-04
217 088	3.25E-05	4.73E-05	503808	8.84E-05	9.78E-05
221 184	3.44E-05	4.93E-05	507904	8.78E-05	1.32E-04
225 280	3.40E-05	4.90E-05	512000	8.76E-05	1.27E-04
229 376	3.48E-05	4.98E-05	516096	9.07E-05	1.00E-04
233 472	3.51E-05	5.30E-05	520192	9.36E-05	1.21E-04
237 568	3.55E-05	5.18E-05	524288	9.46E-05	1.06E-04

Table A.10: IMB PingPong – Data for Figure 3.4(a)

Message Size (Bytes)	IMB (s)		SKaMPI (s)	Message Size (Bytes)	SKaMPI (s)
	Intra-Node	Inter-Node	Inter-Node		Inter-Node
16	1.41E-06	2.60E-06	9.10E-06	249 856	3.48E-04
32	1.52E-06	2.44E-06	9.10E-06	253 952	3.53E-04
64	1.54E-06	2.59E-06	9.10E-06	258 048	3.59E-04
128	1.62E-06	4.59E-06	1.20E-05	262 144	3.64E-04
256	1.83E-06	5.01E-06	1.24E-05	266 240	3.69E-04
512	2.27E-06	6.37E-06	1.41E-05	270 336	3.75E-04
1024	3.06E-06	1.23E-05	1.61E-05	274 432	3.80E-04
2048	4.54E-06	1.61E-05	2.16E-05	278 528	3.85E-04
3648	7.09E-06	1.61E-05	2.55E-05	280 576	3.87E-04
4096	7.07E-06	1.31E-05	2.68E-05	282 624	3.91E-04
8192	1.03E-05	1.94E-05	4.04E-05	286 720	3.96E-04
12 288	1.45E-05	3.26E-05	4.09E-05	290 816	4.01E-04
16 384	1.79E-05	2.97E-05	4.57E-05	294 912	4.07E-04
20 480	—	—	5.10E-05	299 008	4.12E-04
24 576	—	—	5.65E-05	303 104	4.17E-04
28 672	—	—	6.23E-05	307 200	4.22E-04
32 768	—	—	6.72E-05	311 296	4.28E-04
36 864	—	—	7.26E-05	315 392	4.33E-04
40 960	—	—	7.79E-05	319 488	4.38E-04
45 056	—	—	8.76E-05	323 584	4.43E-04
49 152	—	—	8.85E-05	327 680	4.49E-04
53 248	—	—	9.42E-05	331 776	4.54E-04
57 344	—	—	9.90E-05	335 872	4.70E-04
61 440	—	—	1.04E-04	339 968	4.65E-04
65 536	—	—	1.10E-04	344 064	4.75E-04
69 632	—	—	1.15E-04	348 160	4.76E-04
73 728	—	—	1.20E-04	352 256	4.82E-04
77 824	—	—	1.48E-04	356 352	4.86E-04
81 920	—	—	1.31E-04	360 448	4.91E-04
86 016	—	—	1.36E-04	364 544	4.99E-04
90 112	—	—	1.41E-04	368 640	5.02E-04
94 208	—	—	1.47E-04	372 736	5.08E-04
98 304	—	—	1.52E-04	376 832	5.15E-04
102 400	—	—	1.57E-04	380 928	5.18E-04
106 496	—	—	1.63E-04	385 024	5.25E-04
110 592	—	—	1.68E-04	389 120	5.28E-04
114 688	—	—	1.73E-04	393 216	5.34E-04
118 784	—	—	1.79E-04	397 312	5.39E-04
122 880	—	—	1.84E-04	401 408	5.44E-04
126 976	—	—	1.89E-04	405 504	5.50E-04
131 072	—	—	1.94E-04	409 600	5.56E-04
135 168	—	—	2.00E-04	413 696	5.65E-04
139 264	—	—	2.05E-04	417 792	5.66E-04
143 360	—	—	2.10E-04	421 888	5.71E-04
147 456	—	—	2.16E-04	425 984	5.76E-04
151 552	—	—	2.21E-04	430 080	5.82E-04
155 648	—	—	2.26E-04	434 176	5.87E-04
159 744	—	—	2.32E-04	438 272	5.92E-04
163 840	—	—	2.37E-04	442 368	5.98E-04
167 936	—	—	2.42E-04	446 464	6.03E-04
172 032	—	—	2.47E-04	450 560	6.08E-04
176 128	—	—	2.53E-04	454 656	6.13E-04
180 224	—	—	2.58E-04	458 752	6.18E-04
184 320	—	—	2.63E-04	462 848	6.24E-04
188 416	—	—	2.68E-04	466 944	6.29E-04
192 512	—	—	2.74E-04	471 040	6.34E-04
196 608	—	—	2.79E-04	475 136	6.40E-04
200 704	—	—	2.84E-04	479 232	6.45E-04
204 800	—	—	2.90E-04	483 328	6.51E-04
208 896	—	—	2.95E-04	487 424	6.58E-04
212 992	—	—	3.00E-04	491 520	6.61E-04
217 088	—	—	3.06E-04	495 616	6.66E-04
221 184	—	—	3.11E-04	499 712	6.72E-04
225 280	—	—	3.16E-04	503 808	6.77E-04
229 376	—	—	3.22E-04	507 904	6.82E-04
233 472	—	—	3.27E-04	512 000	6.88E-04
237 568	—	—	3.32E-04	516 096	6.93E-04
241 664	—	—	3.38E-04	520 192	6.98E-04
245 760	—	—	3.43E-04	524 288	7.03E-04

Table A.11: Hera IMB Timings, SkaMPI Full Send-Recv – Data for Figure 3.6

Message Size (Bytes)	Time (s)					
	Intra-Node		Inter-Node			
	Single Pair	Two-Pair	Single Pair	64 Pair	126 Pair	256 Pair
16	2.70E-06	2.76E-06	2.98E-06	4.10E-06	4.09E-06	4.03E-06
32	2.73E-06	2.79E-06	3.02E-06	4.85E-06	4.95E-06	4.09E-06
64	2.89E-06	2.91E-06	3.21E-06	5.00E-06	5.44E-06	4.26E-06
128	2.97E-06	3.01E-06	3.45E-06	5.36E-06	4.86E-06	4.49E-06
256	4.06E-06	4.14E-06	4.90E-06	6.27E-06	6.28E-06	5.96E-06
512	4.32E-06	4.42E-06	5.72E-06	8.19E-06	8.13E-06	8.17E-06
1024	4.91E-06	5.23E-06	7.17E-06	1.33E-05	1.32E-05	1.33E-05
2048	7.03E-06	8.00E-06	1.26E-05	3.08E-05	2.70E-05	2.48E-05
3072	7.50E-06	8.80E-06	1.49E-05	4.23E-05	3.78E-05	3.51E-05
4096	7.98E-06	9.90E-06	1.81E-05	5.62E-05	4.93E-05	4.79E-05
8192	9.67E-06	1.33E-05	2.92E-05	1.04E-04	9.17E-05	9.14E-05
9216	1.00E-05	1.42E-05	3.17E-05	1.14E-04	1.02E-04	1.02E-04
10240	1.05E-05	1.51E-05	3.42E-05	1.25E-04	1.12E-04	1.12E-04
11264	1.10E-05	1.59E-05	3.68E-05	1.36E-04	1.25E-04	1.22E-04
12288	1.15E-05	1.69E-05	4.00E-05	1.50E-04	1.35E-04	1.35E-04
13312	1.20E-05	1.77E-05	4.26E-05	1.61E-04	1.45E-04	1.45E-04
14336	1.21E-05	1.86E-05	4.50E-05	1.73E-04	1.58E-04	1.55E-04
15360	1.26E-05	1.93E-05	4.77E-05	1.82E-04	1.68E-04	1.66E-04
16384	1.30E-05	2.03E-05	5.09E-05	1.95E-04	1.79E-04	1.78E-04
17408	1.35E-05	2.12E-05	5.36E-05	2.06E-04	1.91E-04	1.89E-04
18432	1.40E-05	2.19E-05	5.59E-05	2.15E-04	2.02E-04	1.99E-04
19456	1.44E-05	2.31E-05	5.91E-05	2.29E-04	2.12E-04	2.12E-04
20480	1.48E-05	2.37E-05	6.17E-05	2.40E-04	2.22E-04	2.22E-04
21504	1.53E-05	2.47E-05	6.45E-05	2.48E-04	2.35E-04	2.32E-04
22528	1.58E-05	2.54E-05	6.70E-05	2.58E-04	2.45E-04	2.42E-04
23552	1.62E-05	2.65E-05	7.01E-05	2.71E-04	2.55E-04	2.55E-04
24576	1.67E-05	2.74E-05	7.26E-05	2.81E-04	2.68E-04	2.65E-04
25600	1.71E-05	2.83E-05	7.53E-05	2.90E-04	2.78E-04	2.76E-04
26624	1.75E-05	2.91E-05	7.79E-05	3.00E-04	2.89E-04	2.86E-04
27648	1.80E-05	3.01E-05	8.10E-05	3.11E-04	3.01E-04	2.99E-04
28672	1.84E-05	3.09E-05	8.36E-05	3.22E-04	3.12E-04	3.09E-04
29696	1.89E-05	3.18E-05	8.61E-05	3.32E-04	3.22E-04	3.19E-04
30720	1.93E-05	3.25E-05	8.87E-05	3.40E-04	3.32E-04	3.29E-04
31744	1.98E-05	3.37E-05	9.19E-05	3.54E-04	3.45E-04	3.42E-04

Table A.12: IMB Ping-Pong – Data Subset for Figure 3.5

Problem	Iterations		Time (s)										
	Total	Mlagh	MDT	$\sigma_{\overline{x}}$	Mlagh	$\sigma_{\overline{x}}$	Madv	$\sigma_{\overline{x}}$	Shortprint	$\sigma_{\overline{x}}$	Memory	$\sigma_{\overline{x}}$	Mlagh Per Inner Loop
30^3	209	209	5.70E-03	2.00E-05	5.39E-03	9.82E-06	3.73E-02	9.88E-05	1.98E-03	1.72E-05	3.09E-03	2.33E-05	5.39E-03
50^3	209	225	2.62E-02	1.71E-05	2.74E-02	3.27E-05	1.66E-01	5.67E-04	8.75E-03	1.35E-05	1.53E-02	1.43E-04	2.54E-02
80^3	210	289	1.04E-01	2.94E-04	1.50E-01	4.55E-05	6.41E-01	1.25E-03	3.42E-02	6.05E-06	4.80E-02	3.16E-04	1.09E-01
100^3	217	351	2.06E-01	1.55E-04	3.57E-01	1.86E-04	1.22E+00	8.95E-04	6.82E-02	5.93E-05	8.07E-02	5.62E-04	2.21E-01
120^3	229	428	3.51E-01	9.92E-04	7.20E-01	3.16E-04	2.22E+00	2.68E-03	1.15E-01	2.93E-05	1.29E-01	3.16E-04	3.85E-01
150^3	258	578	6.86E-01	2.87E-04	1.73E+00	3.04E-04	4.65E+00	3.10E-03	2.26E-01	1.88E-04	2.35E-01	2.66E-04	7.73E-01

Table A.13: Hydra – Function Serial Scaling – Time Per Iteration – Intel-12.0/OpenMPI-1.4.3 – Data for Figure 4.3

PEs	Compute (Max) (s)	$\sigma_{\overline{x}}$	Collectives (Min) (s)	$\sigma_{\overline{x}}$	Update Bounds (Max) (s)	$\sigma_{\overline{x}}$	Point-To-Point (Min) (s)	$\sigma_{\overline{x}}$	Memory Management (Max) (s)	$\sigma_{\overline{x}}$
1	382.08	0.24	0.03	0.00	18.77	0.02	0.04	0.00	17.52	0.11
2	425.68	1.15	0.20	0.08	18.78	0.02	4.96	0.19	18.46	0.07
4	500.08	0.49	0.30	0.01	23.99	0.09	13.80	0.44	26.16	0.06
8	548.96	0.10	0.43	0.29	23.60	0.20	41.88	0.88	40.82	0.04
12	549.31	0.36	0.58	0.14	24.72	0.13	50.77	1.51	44.00	0.09
16	550.19	0.25	0.58	0.08	24.46	0.20	53.99	0.88	44.17	0.14
24	553.40	2.32	0.82	0.18	24.47	0.03	56.22	2.09	44.11	0.10
32	550.36	1.05	1.34	0.38	22.29	0.24	61.91	1.38	44.16	0.16
48	556.93	1.26	1.08	0.17	21.85	0.43	61.40	2.04	43.96	0.09
64	549.99	0.79	1.61	0.16	20.21	0.43	76.39	1.29	44.03	0.04
96	563.30	1.30	1.27	0.21	19.89	0.21	71.79	6.60	44.04	0.08
128	565.66	7.62	4.09	2.02	21.34	0.94	76.08	0.59	47.15	1.50
192	564.74	0.99	2.58	0.76	19.08	0.23	75.60	5.20	44.13	0.07
256	553.72	0.83	4.00	0.70	18.06	0.20	82.83	2.35	45.43	0.45

Table A.14: Hydra, Minerva, Walltime Breakdown by Component (Min/Max) – Data for Figure 4.4

PEs	Compute (s)							Point-to-Point Exchange (s)						
	Min	Q_1	Q_2	μ	Q_3	Max	σ	Min	Q_1	Q_2	μ	Q_3	Max	σ
1	381.32	381.32	381.32	381.32	381.32	381.32	—	0.04	0.04	0.04	0.04	0.04	0.04	—
2	422.22	422.78	423.35	423.35	423.91	424.48	1.60	4.77	5.41	6.04	6.04	6.68	7.32	1.80
4	494.10	496.32	497.44	497.12	498.23	499.48	2.25	14.02	14.68	15.58	15.86	16.76	18.26	1.85
8	420.20	511.23	542.39	513.18	544.10	548.77	57.08	41.00	45.10	47.18	76.45	77.99	169.77	57.42
12	542.25	545.38	546.55	546.38	547.49	548.89	1.93	49.03	51.27	53.50	53.24	55.10	57.04	2.52
16	491.82	532.03	545.53	533.33	546.59	549.70	23.42	52.98	56.47	58.20	71.62	73.00	118.02	26.44
24	542.27	544.43	546.05	545.90	546.97	550.93	2.13	57.50	60.33	61.11	61.50	61.66	66.22	2.27
32	421.99	543.34	545.22	537.81	546.51	549.70	30.24	63.50	66.00	68.38	76.12	70.94	191.58	30.32
48	542.26	544.69	545.59	545.90	547.17	554.89	2.26	59.54	67.75	69.24	69.34	71.18	75.93	2.91
64	490.69	542.82	544.28	541.33	545.55	551.46	12.76	77.40	82.48	87.95	90.32	92.29	145.40	13.93
96	539.61	543.02	544.18	544.57	545.75	560.74	2.82	68.27	84.50	88.34	88.69	93.62	98.40	5.76
128	416.85	542.77	543.99	542.27	545.69	552.79	16.00	76.66	87.65	92.69	94.10	97.24	214.59	15.92
192	538.14	542.72	543.99	544.46	545.12	565.13	3.40	77.58	88.92	92.53	93.20	98.20	103.83	5.91
256	490.51	542.40	543.58	542.94	544.87	553.68	6.70	85.43	91.19	96.26	96.65	100.94	147.97	7.97

Table A.15: Hydra, Minerva, Process Timing Range, Compute and Exchange – Data for Figure 4.5(a)

PEs	Collectives (s)							Update Bounds (s)						
	Min	Q_1	Q_2	μ	Q_3	**Max**	σ	**Min**	Q_1	Q_2	μ	Q_3	**Max**	σ
1	0.03	0.03	0.03	0.03	0.03	0.03	—	18.80	18.80	18.80	18.80	18.80	18.80	—
2	0.26	0.32	0.37	0.37	0.42	0.47	0.15	18.69	18.72	18.76	18.76	18.79	18.83	0.10
4	0.30	0.62	0.74	0.83	0.95	1.52	0.51	23.43	23.67	23.76	23.73	23.81	23.97	0.22
8	0.76	0.90	1.66	9.34	9.92	33.53	14.85	11.80	19.76	22.83	20.19	23.17	23.32	5.14
12	0.71	1.25	1.83	1.69	2.11	2.56	0.61	18.78	19.39	23.58	22.36	24.28	24.46	2.46
16	0.60	1.29	1.75	5.91	6.27	19.56	7.90	14.27	18.03	19.66	19.51	21.09	24.60	3.36
24	0.75	2.06	2.56	2.43	2.96	3.70	0.81	13.08	18.74	19.76	19.92	23.35	24.54	3.38
32	1.32	2.18	2.43	4.45	2.82	35.42	7.97	9.75	15.02	18.65	17.49	20.09	21.99	3.29
48	1.11	3.26	3.59	3.59	4.14	4.81	0.66	12.12	14.96	17.68	17.28	19.43	22.23	2.49
64	1.43	2.80	3.74	4.60	4.45	21.85	4.28	0.01	3.44	10.25	9.68	16.02	19.35	6.80
96	1.05	5.05	6.05	5.88	6.65	7.73	1.06	0.01	3.29	9.45	9.43	16.13	20.27	7.05
128	2.71	4.20	5.29	5.67	6.01	39.13	4.33	0.01	3.31	8.24	8.77	15.25	20.26	6.61
192	2.43	6.21	7.07	7.08	8.12	10.03	1.41	0.01	2.01	8.69	8.37	14.58	19.03	6.75
256	3.24	5.20	6.17	6.59	7.47	26.85	2.87	0.01	0.01	8.20	7.77	13.70	17.89	6.48

Table A.16: Hydra, Minerva, Process Timing Range, Collectives and Update Bounds – Data for Figure 4.5(c)

	Time (s)						
PEs	**Min**	Q_1	Q_2	μ	Q_3	**Max**	σ
1	17.29	17.29	17.29	17.29	17.29	17.29	—
2	18.26	18.29	18.31	18.31	18.34	18.36	0.07
4	25.86	25.97	26.00	25.98	26.02	26.06	0.09
8	19.06	34.83	40.36	35.11	40.58	40.74	9.90
12	42.80	43.29	43.63	43.54	43.84	44.03	0.41
16	25.16	38.23	43.38	39.02	43.72	44.00	7.98
24	42.78	43.41	43.56	43.57	43.89	44.13	0.39
32	18.77	42.25	43.31	41.33	43.47	43.91	5.99
48	42.53	43.10	43.42	43.38	43.62	44.13	0.39
64	25.36	43.05	43.38	42.24	43.63	43.99	4.38
96	41.81	43.01	43.32	43.23	43.51	43.92	0.41
128	18.64	43.07	43.30	42.71	43.52	44.21	3.17
192	41.17	42.82	43.18	43.10	43.43	43.98	0.46
256	25.36	42.88	43.20	42.90	43.47	44.55	2.25

Table A.17: Hydra, Minerva, Process Timing Range, Memory Management – Data for Figure 4.5(e)

—	Total Walltime (s)						Compute (s)					
PEs	**None**	$\sigma_{\overline{x}}$	**Node**	$\sigma_{\overline{x}}$	**Socket**	$\sigma_{\overline{x}}$	**None**	$\sigma_{\overline{x}}$	**Node**	$\sigma_{\overline{x}}$	**Socket**	$\sigma_{\overline{x}}$
8	655.61	0.44	657.63	0.17	581.63	0.31	548.96	0.1	547.52	0.81	500.09	0.46
16	671.10	0.49	658.91	0.34	584.44	0.28	550.19	0.25	547.98	0.60	549.78	0.73
32	678.20	0.13	673.92	0.57	673.57	0.34	550.36	1.05	547.28	0.29	547.70	0.16
64	689.28	0.30	682.88	0.70	683.99	0.29	549.99	0.79	548.45	0.34	547.50	0.08
128	708.31	8.88	694.81	0.30	693.26	0.46	565.66	7.62	548.25	0.13	565.39	17.52
256	700.35	1.38	702.09	0.66	716.66	15.37	553.72	0.83	548.25	0.13	565.39	17.52

Table A.18: Hydra, Minerva, Weak-Scaling - Node/Socket Load-Balancing – Data for Figures 4.6(a), 4.6(b)

—	DawnDev Time (s)				Hera Time (s)			
PEs	**Total)**	**Max Compute)**	**Min Point-To-Point**	**Min Collectives**	**Total)**	**Max Compute)**	**Min Point-To-Point**	**Min Collectives**
32	—	—	—	—	806.95	703.71	94.43	5.56
64	1665.85	1573.35	92.56	3.40	902.04	705.53	175.30	19.70
128	1702.68	1571.53	129.55	3.82	905.10	720.79	146.96	25.89
256	1718.81	1569.75	139.64	4.65	975.13	713.22	187.45	41.52
512	1707.87	1571.23	120.76	5.65	1002.74	712.95	209.95	49.02
1024	1729.29	1570.17	125.58	7.20	1039.72	730.14	208.42	64.34
2048	1780.04	1569.47	153.47	12.25	1172.40	721.20	239.81	105.64

Table A.19: Hydra, DawnDev/Hera, Weak-Scaling - Walltime Breakdown – Data for Figure 4.7

PEs	Time (s)									
	Compute (Max)	$\sigma_{\overline{x}}$	Collectives (Min)	$\sigma_{\overline{x}}$	Update Bounds (Max)	$\sigma_{\overline{x}}$	Point-To-Point (Min)	$\sigma_{\overline{x}}$	Memory Management (Max)	$\sigma_{\overline{x}}$
1	1828.79	0.64	0.04	0.00	51.24	0.03	0.05	0.00	60.72	0.04
2	956.97	1.04	0.46	0.18	28.99	0.10	13.62	0.72	33.92	0.04
4	534.81	0.60	0.37	0.10	20.48	0.05	18.73	0.42	26.46	0.07
8	302.54	0.80	0.63	0.06	15.93	0.21	28.85	0.46	18.97	0.10
12	206.79	0.17	0.46	0.04	11.96	0.03	25.02	0.27	13.03	0.07
16	145.41	0.40	0.48	0.12	9.81	0.00	21.84	0.39	9.88	0.03
24	97.96	0.10	0.56	0.02	6.95	0.06	16.52	0.16	6.62	0.01
32	72.48	0.38	0.48	0.03	5.00	0.07	13.92	0.16	5.12	0.01
48	46.04	0.14	0.62	0.03	3.62	0.01	14.36	0.11	3.47	0.01
64	35.56	0.09	0.72	0.10	3.51	0.05	12.01	0.11	2.70	0.01
96	23.24	0.06	0.88	0.02	2.57	0.05	9.01	0.16	1.83	0.01
128	17.62	0.12	1.00	0.07	1.85	0.02	9.55	0.13	1.54	0.13
192	11.44	0.03	1.09	0.02	1.29	0.01	6.60	0.02	0.95	0.00
256	10.42	1.70	1.45	0.04	1.35	0.29	5.57	0.39	0.71	0.10

Table A.20: Hydra, Minerva, Walltime Breakdown by Function (Min/Max) – Data for Figure 4.8

PEs	Compute (s)							Point-to-Point Exchange (s)						
	Min	Q_1	Q_2	μ	Q_3	Max	σ	Min	Q_1	Q_2	μ	Q_3	Max	σ
1	1827.56	1827.56	1827.56	1827.56	1827.56	1827.56	—	0.05	0.05	0.05	0.05	0.05	0.05	—
2	947.32	949.90	952.47	952.47	955.04	957.61	7.27	12.21	14.02	15.82	15.82	17.63	19.44	5.11
4	528.68	529.44	530.03	530.64	531.23	533.83	2.24	19.55	20.65	21.68	21.47	22.49	22.95	1.51
8	220.62	278.70	299.09	280.05	300.41	301.73	36.63	29.19	30.90	31.42	48.59	49.59	101.66	32.59
12	202.48	203.37	203.60	203.86	204.18	206.16	1.04	24.82	25.90	26.84	27.44	29.75	30.07	2.01
16	122.75	135.79	142.04	138.15	144.40	145.33	8.65	21.53	23.96	25.80	28.57	31.07	41.03	6.95
24	92.65	93.25	94.42	94.81	96.40	97.81	1.75	16.25	18.37	19.48	19.61	20.58	22.82	1.77
32	53.53	67.08	67.77	67.41	69.31	72.44	4.00	13.63	16.75	17.73	18.29	18.77	28.17	3.02
48	41.78	42.76	43.17	43.24	43.64	45.79	0.91	14.57	16.12	16.94	16.96	17.65	19.27	1.04
64	28.63	32.36	32.78	32.64	33.27	35.63	1.22	11.94	14.37	15.35	15.31	16.24	17.92	1.23
96	20.89	21.35	21.63	21.67	21.98	23.29	0.44	8.79	10.37	10.78	10.85	11.49	12.23	0.77
128	12.57	15.76	16.20	15.98	16.35	17.61	0.65	9.55	11.00	11.41	11.41	11.80	13.37	0.58
192	9.81	10.38	10.52	10.49	10.74	11.48	0.31	6.56	7.57	7.77	7.82	8.17	8.78	0.40
256	6.56	7.55	7.84	7.77	8.00	8.86	0.32	5.76	6.98	7.17	7.16	7.42	7.90	0.33

Table A.21: Hydra, Minerva, Process Timing Range, Compute and Exchange – Data for Figure 4.9(a)

PEs	Collectives (s)							Update Bounds (s)						
	Min	Q_1	Q_2	μ	Q_3	Max	σ	Min	Q_1	Q_2	μ	Q_3	Max	σ
1	0.04	0.04	0.04	0.04	0.04	0.04	—	51.21	51.21	51.21	51.21	51.21	51.21	—
2	0.16	1.06	1.96	1.96	2.86	3.75	2.54	28.78	28.86	28.94	28.94	29.02	29.10	0.23
4	0.48	1.80	2.41	1.98	2.59	2.63	1.02	20.11	20.24	20.30	20.27	20.33	20.37	0.11
8	0.62	1.36	1.77	7.22	7.50	24.71	10.76	8.51	13.76	15.57	13.85	15.63	15.69	3.23
12	0.61	1.44	1.69	1.66	1.88	2.34	0.46	8.23	8.52	11.59	10.60	11.74	11.92	1.63
16	0.71	1.14	2.43	3.64	4.49	9.50	3.45	6.09	6.45	6.74	7.30	7.58	9.80	1.39
24	0.53	1.46	2.28	2.21	2.91	3.42	0.81	2.81	4.36	4.81	4.98	6.16	7.05	1.33
32	0.41	1.66	2.25	2.61	2.83	9.24	1.88	2.02	2.21	3.63	3.36	3.85	5.10	0.98
48	0.68	1.81	2.01	2.06	2.45	3.02	0.50	0.02	1.46	2.27	2.07	2.81	3.60	1.03
64	0.53	1.92	2.14	2.25	2.48	4.55	0.70	0.01	0.76	1.89	1.77	2.83	3.61	1.15
96	0.89	1.67	1.81	1.83	2.00	2.37	0.27	0.01	0.41	1.30	1.18	1.85	2.55	0.81
128	0.85	1.48	1.65	1.74	1.91	3.64	0.40	0.01	0.44	1.01	0.83	1.29	1.85	0.55
192	1.09	1.69	1.81	1.81	1.93	2.26	0.19	0.01	0.27	0.70	0.56	0.82	1.30	0.38
256	1.42	1.74	1.87	1.94	2.09	3.02	0.27	0.01	0.01	0.48	0.41	0.59	1.05	0.29

Table A.22: Hydra, Minerva, Process Timing Range, Collectives and Update Bounds – Data for Figure 4.9(c)

	Time (s)						
PEs	**Min**	Q_1	Q_2	μ	Q_3	**Max**	σ
1	60.75	60.75	60.75	60.75	60.75	60.75	—
2	33.82	33.84	33.86	33.86	33.89	33.91	0.06
4	26.08	26.19	26.27	26.23	26.31	26.31	0.11
8	11.37	16.79	18.71	16.92	18.77	19.02	3.42
12	12.60	12.72	12.83	12.79	12.84	12.92	0.10
16	7.54	8.87	9.60	9.15	9.73	9.91	0.91
24	6.30	6.32	6.41	6.42	6.50	6.61	0.10
32	3.50	4.79	4.90	4.81	4.99	5.13	0.36
48	3.16	3.25	3.30	3.30	3.37	3.49	0.08
64	1.97	2.47	2.51	2.50	2.57	2.68	0.15
96	1.65	1.69	1.71	1.72	1.76	1.85	0.04
128	0.88	1.29	1.32	1.31	1.35	1.43	0.07
192	0.78	0.88	0.90	0.89	0.91	0.95	0.04
256	0.35	0.54	0.56	0.55	0.57	0.61	0.03

Table A.23: Hydra, Minerva, Process Timing Range, Memory Management – Data for Figure 4.9(e)

	Madvmx			Madvmy			Madvmz		
	Comm (s)		Compute Diff. (s)	Comm (s)		Compute Diff. (s)	Comm (s)		Compute Diff. (s)
PEs	Min	Max	Madvx	Min	Max	Madvy	Min	Max	Madvz
2	5.09E-3	6.54E-3	1.14E-3	5.12E-3	9.05E-3	4.21E-3	5.10E-3	8.52E-3	1.63E-3
4	1.56E-2	1.72E-2	1.97E-3	1.46E-2	2.28E-2	8.33E-3	1.58E-2	1.74E-2	2.07E-3
8	4.60E-2	6.55E-2	1.99E-2	4.43E-2	2.07E-1	1.62E-1	4.56E-2	6.68E-2	2.27E-2
12	5.05E-2	6.09E-2	3.66E-3	4.84E-2	6.58E-2	8.79E-3	4.93E-2	6.22E-2	4.97E-3
16	4.93E-2	6.34E-2	7.81E-3	4.92E-2	9.60E-2	5.09E-2	4.89E-2	6.42E-2	8.99E-3
24	5.02E-2	7.27E-2	5.55E-3	4.98E-2	7.87E-2	2.03E-2	4.99E-2	7.34E-2	7.68E-3
32	5.13E-2	8.59E-2	2.06E-2	4.89E-2	2.30E-1	1.74E-1	5.15E-2	8.67E-2	2.32E-2
48	4.83E-2	7.62E-2	9.24E-3	4.73E-2	8.85E-2	2.07E-2	4.81E-2	7.94E-2	7.95E-3
64	5.46E-2	9.47E-2	1.89E-2	5.24E-2	1.27E-1	4.98E-2	5.26E-2	9.51E-2	1.32E-2
96	5.63E-2	9.79E-2	1.14E-2	5.46E-2	1.00E-1	3.63E-2	5.65E-2	9.64E-2	1.51E-2
128	5.15E-2	9.76E-2	2.01E-2	5.50E-2	2.27E-1	1.69E-1	5.15E-2	9.82E-2	2.27E-2
192	5.57E-2	9.83E-2	1.16E-2	5.30E-2	1.08E-1	3.68E-2	5.71E-2	1.01E-1	1.53E-2
256	5.70E-2	1.05E-1	2.23E-2	5.49E-2	1.21E-1	5.10E-2	5.84E-2	1.05E-1	1.25E-2

Table A.24: Minerva, Data for Figures 5.2, 5.3

	Empirical					Model				
PEs	Compute	Comms	Coll	Updb	Mem	Compute	Comms	Coll	Updb	Mem
2	424.48	4.77	0.26	18.83	18.36	420.63	2.06	0.00	19.78	18.12
4	499.48	14.02	0.30	23.97	26.06	494.20	16.98	0.01	28.54	26.50
8	548.77	41.00	0.76	23.32	40.74	556.58	23.26	0.01	46.02	43.05
16	549.70	52.98	0.60	24.60	44.00	556.58	31.90	0.01	46.02	43.05
32	549.70	63.50	1.32	21.99	43.91	556.58	43.24	0.02	46.02	43.05
64	551.46	77.40	1.43	19.35	43.99	556.58	53.70	0.04	46.02	43.05
128	552.79	76.66	2.71	20.26	44.21	556.58	53.70	0.05	46.02	43.05
256	553.68	85.43	3.24	17.89	44.55	556.58	57.92	0.07	46.02	43.05

Table A.25: Minerva, Data for Figure 5.9a

	Empirical					Model				
PEs	**Compute**	**Comms**	**Coll**	**Updb**	**Mem**	**Compute**	**Comms**	**Coll**	**Updb**	**Mem**
1	1827.56	0.05	0.04	51.21	60.75	1825.81	0.00	0.00	46.71	61.04
2	957.61	12.21	0.16	29.10	33.91	960.05	5.33	0.00	32.01	34.10
4	533.83	19.55	0.48	20.37	26.31	536.71	11.36	0.01	27.10	26.74
8	301.73	29.19	0.62	15.69	19.02	302.10	16.79	0.01	31.51	19.96
16	145.33	21.53	0.71	9.80	9.91	148.98	18.40	0.01	17.90	10.36
32	72.44	13.63	0.41	5.10	5.13	73.41	14.49	0.03	10.20	5.14
64	35.63	11.94	0.53	3.61	2.68	35.52	9.45	0.05	7.84	2.63
128	17.61	9.55	0.85	1.85	1.43	17.95	6.80	0.07	5.04	1.32
256	8.86	5.76	1.42	1.05	0.61	8.99	5.11	0.09	3.32	0.64

Table A.26: Minerva, Data for Figure 5.9b

For all PAPI statistics, the majority are obtained from executions of Hydra on Minerva. However the lack of L1 cache hit/access counters neccesitated the use of an alternate machine for such values. This machine was a single 2.4GHz Intel X3430 workstation. See 3.3.5 for further details.

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	5.80E-3	2.06E-5	1.20E+6	2.04E+3	1.62E+7	5.93E+3	1.03E+5	1.57E+3	1.09E+5	4.48E+3	2.66E+3	3.66E+1	0.07
50^3	2.49E-2	1.23E-4	5.49E+6	1.45E+4	7.09E+7	2.40E+4	5.16E+5	8.35E+3	5.04E+5	8.51E+3	1.18E+4	5.12E+2	0.08
80^3	9.98E-2	5.77E-4	2.25E+7	9.85E+4	2.82E+8	2.78E+4	2.55E+6	1.70E+4	2.45E+6	2.89E+4	5.01E+4	1.40E+3	0.08
100^3	1.90E-1	2.72E-4	4.36E+7	5.59E+4	5.44E+8	1.75E+5	5.53E+6	3.37E+3	5.30E+6	4.20E+4	1.02E+5	1.05E+3	0.08
120^3	3.25E-1	7.67E-4	7.55E+7	3.31E+5	9.32E+8	1.42E+5	1.01E+7	3.27E+4	9.87E+6	2.34E+4	1.89E+5	3.04E+3	0.08
150^3	6.29E-1	1.28E-4	1.47E+8	—	1.81E+9	3.76E+5	2.10E+7	—	2.07E+7	—	3.66E+5	—	0.08

Table A.27: PAPI Serial Mean Statistics for Kernel Madvx_2, Variant A– Data for Figures 6.1,6.2,6.3(a)

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	7.15E-3	2.23E-5	1.20E+6	8.71E+2	1.72E+7	5.54E+3	5.05E+5	5.29E+2	2.81E+5	1.94E+3	2.15E+5	6.66E+2	0.07
50^3	3.40E-2	1.33E-4	5.50E+6	1.76E+4	7.56E+7	8.42E+3	2.45E+6	5.28E+3	1.32E+6	6.55E+3	1.11E+6	5.08E+2	0.07
80^3	1.41E-1	1.79E-4	2.27E+7	2.24E+4	3.01E+8	2.40E+4	1.02E+7	2.26E+4	5.73E+6	4.63E+3	4.45E+6	6.31E+3	0.08
100^3	2.70E-1	1.49E-4	4.44E+7	3.72E+4	5.82E+8	4.00E+4	2.06E+7	1.44E+4	1.18E+7	2.42E+4	8.64E+6	2.50E+3	0.08
120^3	4.74E-1	4.76E-4	7.67E+7	6.48E+4	9.97E+8	1.09E+5	3.54E+7	2.71E+4	2.02E+7	4.93E+4	1.51E+7	1.03E+4	0.08
150^3	1.03E+0	2.53E-3	1.50E+8	—	1.94E+9	4.08E+5	7.15E+7	—	4.21E+7	—	2.97E+7	—	0.08

Table A.28: PAPI Serial Mean Statistics for Kernel Madvy_2, Variant A– Data for Figures 6.1, 6.2, 6.3(b)

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	6.56E-3	2.16E-5	1.21E+6	2.39E+3	1.69E+7	6.43E+3	4.02E+5	7.63E+2	3.79E+5	3.72E+3	2.98E+4	6.62E+1	0.07
50^3	3.09E-2	1.31E-4	5.53E+6	1.62E+4	7.49E+7	5.25E+3	2.71E+6	9.38E+3	2.54E+6	1.29E+4	1.58E+5	2.59E+3	0.07
80^3	1.24E-1	2.70E-4	2.27E+7	4.19E+4	2.99E+8	3.17E+4	1.04E+7	3.32E+5	9.95E+6	1.13E+5	6.33E+5	1.41E+3	0.08
100^3	2.39E-1	3.43E-4	4.44E+7	1.52E+4	5.81E+8	1.65E+5	2.00E+7	1.89E+4	1.87E+7	3.10E+4	1.29E+6	5.96E+3	0.08
120^3	3.98E-1	7.77E-5	7.66E+7	1.92E+5	9.97E+8	3.71E+5	4.28E+7	4.69E+5	4.08E+7	5.43E+4	2.49E+6	2.94E+4	0.08
150^3	8.10E-1	1.42E-3	1.49E+8	—	1.94E+9	4.36E+5	8.30E+7	—	7.53E+7	—	8.08E+6	—	0.08

Table A.29: PAPI Serial Mean Statistics for Kernel Madvz_2, Variant A– Data for Figures 6.1, 6.2, 6.3(c)

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	2.97E-3	4.83E-6	4.76E+6	2.47E+3	8.48E+6	4.65E+2	9.24E+4	4.77E+2	8.41E+4	6.65E+2	7.12E+3	7.93E+1	0.56
50^3	1.29E-2	1.94E-5	2.10E+7	1.06E+4	3.63E+7	3.53E+3	5.48E+5	1.25E+3	5.02E+5	4.20E+3	3.62E+4	2.51E+2	0.58
80^3	5.00E-2	9.94E-5	8.31E+7	5.77E+4	1.42E+8	8.41E+3	2.29E+6	1.96E+4	2.04E+6	1.48E+4	1.22E+5	4.25E+2	0.59
100^3	9.79E-2	1.19E-4	1.62E+8	5.58E+4	2.73E+8	6.95E+3	5.40E+6	1.75E+4	4.99E+6	3.93E+3	2.60E+5	7.80E+2	0.59
120^3	1.69E-1	4.83E-4	2.77E+8	1.35E+5	4.67E+8	3.02E+4	1.24E+7	3.77E+4	1.16E+7	1.31E+4	5.16E+5	3.42E+3	0.59
150^3	3.22E-1	1.50E-4	5.37E+8	—	9.02E+8	5.40E+4	2.15E+7	—	2.01E+7	—	1.04E+6	—	0.60

Table A.30: PAPI Serial Mean Statistics for Kernel Madvmx_1, Variant A– Data for Figures 6.1, 6.2, 6.3(d)

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	3.09E-3	9.75E-6	4.81E+6	3.12E+3	8.28E+6	2.13E+3	5.43E+5	6.00E+2	5.18E+5	9.00E+2	2.13E+4	3.48E+2	0.58
50^3	1.45E-2	3.49E-5	2.13E+7	4.21E+3	3.53E+7	2.75E+3	3.21E+6	1.83E+3	3.02E+6	2.74E+3	1.85E+5	1.65E+3	0.60
80^3	6.27E-2	1.66E-4	8.44E+7	2.56E+4	1.37E+8	1.52E+4	1.34E+7	5.51E+3	1.22E+7	1.06E+4	1.19E+6	7.59E+3	0.61
100^3	1.24E-1	9.06E-5	1.64E+8	4.39E+4	2.65E+8	4.09E+4	2.71E+7	9.24E+3	2.42E+7	2.20E+4	2.63E+6	4.19E+4	0.62
120^3	2.14E-1	4.19E-4	2.82E+8	1.00E+5	4.54E+8	4.66E+4	4.87E+7	2.85E+3	4.10E+7	4.37E+4	7.53E+6	4.46E+4	0.62
150^3	4.20E-1	2.25E-4	5.49E+8	—	8.86E+8	2.08E+5	9.17E+7	—	6.91E+7	—	2.21E+7	—	0.62

Table A.31: PAPI Serial Mean Statistics for Kernel Madvmy_1, Variant A– Data for Figures 6.1, 6.2, 6.3(e)

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	4.13E-3	7.47E-6	4.91E+6	3.55E+2	8.36E+6	2.06E+3	7.87E+5	3.34E+2	4.39E+5	6.44E+2	3.43E+5	4.16E+2	0.59
50^3	2.24E-2	1.27E-4	2.18E+7	2.13E+4	3.58E+7	8.74E+3	4.04E+6	4.49E+3	2.27E+6	4.83E+3	1.76E+6	9.44E+3	0.61
80^3	8.38E-2	4.07E-4	8.63E+7	1.11E+5	1.42E+8	7.38E+4	1.68E+7	2.48E+5	9.49E+6	9.91E+4	7.46E+6	1.81E+4	0.61
100^3	1.63E-1	5.43E-4	1.67E+8	3.77E+4	2.74E+8	6.10E+4	2.94E+7	4.83E+4	1.65E+7	5.01E+4	1.26E+7	7.88E+4	0.61
120^3	4.32E-1	1.17E-3	2.87E+8	1.61E+5	4.78E+8	1.12E+5	6.44E+7	7.99E+4	3.30E+7	6.21E+4	3.12E+7	5.42E+4	0.60
150^3	1.07E+0	2.29E-3	5.57E+8	—	9.33E+8	1.72E+5	1.33E+8	—	5.18E+7	—	8.07E+7	—	0.60

Table A.32: PAPI Serial Mean Statistics for Kernel Madvmz_1, Variant A– Data for Figures 6.1, 6.2, 6.3(f)

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	4.80E-4	3.74E-7	3.25E+5	2.14E+1	5.22E+5	8.82E+0	2.18E+4	9.79E+0	2.08E+4	3.45E+1	7.78E+2	1.62E+1	0.62
50^3	2.19E-3	6.59E-7	1.50E+6	1.69E+1	2.35E+6	6.19E+2	9.81E+4	4.82E+0	9.40E+4	2.15E+1	2.42E+3	1.22E+1	0.64
80^3	9.06E-3	2.43E-6	6.17E+6	3.70E+2	9.53E+6	2.13E+2	3.97E+5	2.65E+1	3.81E+5	7.08E+1	1.04E+4	6.36E+1	0.65
100^3	1.77E-2	1.03E-5	1.20E+7	3.26E+2	1.85E+7	1.58E+3	7.69E+5	2.09E+2	7.37E+5	4.68E+2	2.00E+4	2.99E+2	0.65
120^3	3.07E-2	7.45E-6	2.08E+7	2.11E+2	3.19E+7	2.24E+2	1.32E+6	1.96E+2	1.26E+6	9.04E+1	3.40E+4	1.17E+2	0.65
150^3	6.01E-2	2.49E-5	4.06E+7	—	6.21E+7	3.69E+2	2.57E+6	—	2.44E+6	—	6.66E+4	—	0.65

Table A.33: PAPI Serial Mean Statistics for Kernel MDT_1, Variant A– Data for Figures 6.1

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	1.62E-3	1.48E-6	1.33E+6	3.24E+2	2.37E+6	2.57E+1	7.21E+4	3.82E+1	6.92E+4	7.65E+1	2.44E+3	7.94E+0	0.56
50^3	7.40E-3	1.61E-6	6.14E+6	1.47E+2	1.09E+7	1.56E+2	3.22E+5	2.41E+2	3.07E+5	4.39E+2	1.11E+4	6.04E+1	0.57
80^3	3.00E-2	6.79E-6	2.51E+7	6.89E+2	4.43E+7	2.70E+2	1.30E+6	2.88E+3	1.23E+6	1.60E+3	3.91E+4	1.88E+2	0.57
100^3	5.85E-2	2.87E-5	4.91E+7	1.97E+2	8.62E+7	4.10E+2	2.51E+6	3.11E+2	2.39E+6	3.28E+3	6.78E+4	4.51E+2	0.57
120^3	1.01E-1	2.75E-5	8.49E+7	5.32E+2	1.49E+8	1.31E+3	4.37E+6	1.39E+4	4.12E+6	2.64E+3	1.17E+5	5.98E+2	0.57
150^3	1.97E-1	1.18E-4	1.66E+8	—	2.90E+8	2.48E+3	9.20E+6	—	8.82E+6	—	2.25E+5	—	0.57

Table A.34: PAPI Serial Mean Statistics for Kernel MDT_2, Variant A– Data for Figures 6.1

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	2.79E-4	1.55E-7	7.72E+5	3.41E+2	8.19E+5	3.11E+1	2.96E+4	1.57E+1	2.78E+4	1.40E+1	1.69E+3	9.86E+0	0.94
50^3	1.27E-3	1.52E-6	3.56E+6	6.58E+2	3.68E+6	9.66E+2	1.27E+5	4.92E+1	1.19E+5	1.06E+2	6.88E+3	1.01E+2	0.97
80^3	5.15E-3	3.18E-6	1.46E+7	5.54E+3	1.49E+7	7.82E+2	4.95E+5	6.82E+2	4.53E+5	4.30E+2	3.73E+4	3.13E+2	0.98
100^3	9.95E-3	5.41E-6	2.85E+7	9.15E+3	2.89E+7	5.42E+2	9.47E+5	1.59E+3	8.72E+5	1.78E+3	7.04E+4	6.02E+2	0.99
120^3	1.71E-2	1.27E-5	4.92E+7	1.31E+4	4.96E+7	3.80E+2	1.62E+6	4.97E+2	1.48E+6	6.02E+2	1.24E+5	5.94E+2	0.99
150^3	3.33E-2	2.58E-5	9.60E+7	—	9.65E+7	7.28E+3	3.23E+6	—	2.97E+6	—	3.25E+5	—	0.99

Table A.35: PAPI Serial Mean Statistics for Kernel Mdivu, Variant A– Data for Figures 6.1

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	2.66E-3	1.65E-6	3.96E+6	2.40E+3	3.57E+6	1.03E+2	4.70E+4	1.68E+1	4.48E+4	1.42E+2	1.79E+3	3.66E+1	1.11
50^3	1.26E-2	7.70E-6	1.83E+7	2.17E+3	1.64E+7	7.59E+2	1.97E+5	8.25E+1	1.87E+5	1.82E+2	9.16E+3	2.74E+2	1.11
80^3	4.88E-2	4.96E-6	7.50E+7	5.12E+3	6.67E+7	8.25E+2	8.09E+5	1.34E+3	7.78E+5	2.11E+3	2.82E+4	8.53E+1	1.12
100^3	9.91E-2	5.33E-5	1.46E+8	6.88E+3	1.31E+8	2.04E+3	1.55E+6	1.38E+3	1.46E+6	5.94E+3	5.26E+4	2.57E+2	1.12
120^3	1.66E-1	1.04E-4	2.53E+8	3.06E+3	2.25E+8	5.25E+2	2.76E+6	2.33E+3	2.66E+6	5.65E+2	8.65E+4	3.43E+2	1.13
150^3	3.26E-1	7.26E-5	4.94E+8	—	4.38E+8	8.98E+3	7.01E+6	—	6.85E+6	—	1.66E+5	—	1.13

Table A.36: PAPI Serial Mean Statistics for Kernel Lartvis_1, Variant A– Data for Figures 6.1

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	1.25E-3	4.99E-7	2.48E+6	3.38E+2	3.59E+6	2.19E+2	5.42E+4	3.56E+1	5.17E+4	1.52E+1	2.19E+3	5.91E+0	0.69
50^3	5.56E-3	1.30E-6	1.11E+7	1.43E+3	1.59E+7	3.55E+2	2.27E+5	3.21E+2	2.16E+5	1.59E+2	9.68E+3	1.09E+2	0.70
80^3	2.22E-2	1.17E-5	4.43E+7	9.91E+2	6.33E+7	5.50E+2	9.03E+5	2.23E+3	8.67E+5	6.84E+2	2.92E+4	3.49E+1	0.70
100^3	4.28E-2	2.15E-5	8.59E+7	1.08E+3	1.22E+8	2.98E+3	1.74E+6	7.35E+3	1.68E+6	6.39E+3	5.14E+4	2.32E+2	0.70
120^3	7.35E-2	3.77E-5	1.48E+8	2.12E+3	2.10E+8	2.28E+3	3.10E+6	4.32E+2	3.00E+6	1.48E+3	8.48E+4	4.03E+2	0.70
150^3	1.42E-1	3.25E-5	2.87E+8	—	4.08E+8	7.74E+3	6.17E+6	—	6.03E+6	—	1.63E+5	—	0.70

Table A.37: PAPI Serial Mean Statistics for Kernel UpdVel, Variant A– Data for Figures 6.1

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	7.79E-4	1.91E-6	2.09E+5	1.19E+0	2.03E+6	5.51E+2	3.81E+4	2.53E+0	3.60E+4	1.12E+1	1.56E+3	6.20E+0	0.10
50^3	3.59E-3	1.39E-5	9.26E+5	8.76E+0	9.38E+6	2.41E+4	1.65E+5	1.95E+1	1.56E+5	5.99E+2	6.53E+3	1.72E+2	0.10
80^3	1.47E-2	8.01E-6	3.71E+6	2.50E+1	3.86E+7	5.57E+3	6.52E+5	1.68E+2	5.79E+5	3.60E+2	3.64E+4	6.82E+1	0.10
100^3	2.85E-2	1.79E-5	7.19E+6	6.96E+1	7.53E+7	2.66E+4	1.26E+6	9.01E+1	1.12E+6	2.19E+2	7.30E+4	1.52E+2	0.10
120^3	4.91E-2	2.44E-5	1.24E+7	4.23E+1	1.30E+8	2.99E+4	2.15E+6	1.38E+2	1.91E+6	1.53E+3	1.26E+5	6.76E+2	0.10
150^3	9.54E-2	3.24E-5	2.41E+7	—	2.54E+8	7.07E+3	4.16E+6	—	3.72E+6	—	2.37E+5	—	0.09

Table A.38: PAPI Serial Mean Statistics for Kernel Madv_1, Variant A– Data for Figures 6.1

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	3.29E-3	1.20E-5	1.32E+6	2.53E+3	9.42E+6	1.05E+3	3.26E+4	2.56E+1	3.06E+4	7.22E+1	1.83E+3	3.24E+1	0.14
50^3	1.51E-2	7.10E-6	6.04E+6	1.17E+4	4.28E+7	1.74E+4	1.40E+5	4.58E+2	1.29E+5	3.92E+2	9.76E+3	8.02E+1	0.14
80^3	6.06E-2	1.41E-4	2.45E+7	6.62E+4	1.73E+8	8.75E+3	5.60E+5	1.95E+3	5.24E+5	3.80E+3	3.43E+4	6.37E+2	0.14
100^3	1.19E-1	1.96E-4	4.80E+7	1.18E+5	3.38E+8	1.52E+4	1.10E+6	3.82E+3	1.06E+6	3.39E+2	5.17E+4	1.35E+2	0.14
120^3	2.02E-1	—	8.25E+7	1.69E+5	5.81E+8	7.68E+4	1.96E+6	4.03E+3	1.85E+6	3.91E+3	8.63E+4	8.69E+2	0.14
150^3	3.98E-1	6.72E-4	1.62E+8	—	1.13E+9	6.94E+4	3.94E+6	—	4.02E+6	—	1.55E+5	—	0.14

Table A.39: PAPI Serial Mean Statistics for Kernel Madvx_2, Variant B – Data for Figures 6.3(a)

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	3.49E-3	1.07E-5	1.28E+6	7.64E+2	1.06E+7	5.60E+4	9.72E+4	1.94E+1	9.50E+4	1.28E+2	2.10E+3	2.37E+1	0.12
50^3	1.59E-2	2.32E-5	5.86E+6	7.43E+2	4.78E+7	3.88E+3	4.49E+5	3.31E+2	4.22E+5	2.76E+3	2.52E+4	2.44E+3	0.12
80^3	6.40E-2	4.52E-4	2.39E+7	1.01E+3	1.94E+8	1.17E+5	1.84E+6	1.07E+3	1.74E+6	1.30E+3	9.37E+4	1.19E+3	0.12
100^3	1.24E-1	1.23E-4	4.65E+7	6.84E+3	3.77E+8	6.37E+3	3.53E+6	2.24E+3	3.39E+6	1.31E+2	1.28E+5	7.39E+2	0.12
120^3	2.15E-1	—	8.02E+7	9.43E+4	6.50E+8	3.90E+3	6.14E+6	3.25E+3	5.93E+6	2.09E+3	1.97E+5	1.57E+3	0.12
150^3	4.15E-1	5.65E-4	1.57E+8	—	1.27E+9	1.11E+5	1.18E+7	—	1.14E+7	—	3.58E+5	—	0.12

Table A.40: PAPI Serial Mean Statistics for Kernel Madvy_2, Variant B – Data for Figures 6.3(b)

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	3.56E-3	1.62E-5	1.26E+6	2.01E+3	1.04E+7	1.01E+3	1.10E+5	4.69E+1	7.78E+4	2.60E+2	3.19E+4	1.07E+2	0.12
50^3	1.68E-2	5.48E-5	5.76E+6	5.67E+2	4.74E+7	1.77E+4	4.94E+5	2.55E+2	3.21E+5	3.84E+2	1.72E+5	4.26E+2	0.12
80^3	6.54E-2	3.05E-4	2.35E+7	4.88E+3	1.93E+8	2.21E+3	1.96E+6	2.09E+3	1.41E+6	7.39E+3	5.38E+5	1.10E+4	0.12
100^3	1.26E-1	3.08E-4	4.57E+7	2.74E+4	3.76E+8	8.36E+3	3.77E+6	6.33E+3	3.02E+6	1.13E+4	7.25E+5	1.56E+4	0.12
120^3	2.14E-1	—	7.87E+7	9.19E+4	6.49E+8	3.26E+4	6.47E+6	5.83E+3	5.41E+6	8.92E+3	1.02E+6	8.26E+3	0.12
150^3	4.17E-1	2.82E-3	1.54E+8	—	1.26E+9	2.93E+5	1.22E+7	—	1.07E+7	—	1.42E+6	—	0.12

Table A.41: PAPI Serial Mean Statistics for Kernel Madvz_2, Variant B – Data for Figures 6.3(c)

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	2.69E-3	2.46E-5	4.94E+6	6.36E+3	7.58E+6	7.96E+2	1.43E+5	6.38E+2	1.17E+5	1.33E+3	1.90E+4	4.34E+2	0.65
50^3	1.17E-2	4.04E-5	2.15E+7	3.33E+4	3.21E+7	2.05E+3	7.74E+5	5.71E+3	6.55E+5	9.45E+2	6.90E+4	1.62E+3	0.67
80^3	4.53E-2	9.22E-6	8.46E+7	2.61E+4	1.24E+8	1.44E+4	3.23E+6	1.16E+4	2.71E+6	3.51E+3	2.60E+5	4.63E+3	0.69
100^3	8.79E-2	1.27E-4	1.64E+8	5.94E+4	2.38E+8	1.89E+4	7.04E+6	1.16E+4	6.08E+6	2.64E+4	4.71E+5	2.76E+3	0.69
120^3	1.52E-1	—	2.80E+8	1.55E+5	4.04E+8	4.82E+4	1.41E+7	2.25E+4	1.24E+7	1.08E+4	8.67E+5	1.43E+4	0.69
150^3	2.91E-1	5.08E-4	5.43E+8	—	7.78E+8	8.72E+4	2.62E+7	—	2.29E+7	—	1.73E+6	—	0.70

Table A.42: PAPI Serial Mean Statistics for Kernel Madvmx_1, Variant B – Data for Figures 6.3(d)

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	3.48E-3	1.25E-5	5.01E+6	1.82E+2	9.13E+6	7.33E+2	3.14E+5	3.19E+1	2.81E+5	9.23E+2	2.54E+4	8.90E+2	0.55
50^3	1.48E-2	1.26E-5	2.18E+7	9.04E+1	3.86E+7	8.27E+2	1.39E+6	9.27E+2	1.22E+6	3.80E+2	1.17E+5	1.50E+3	0.56
80^3	5.80E-2	1.41E-5	8.60E+7	3.75E+3	1.50E+8	2.02E+3	5.36E+6	4.51E+3	4.66E+6	2.69E+3	4.20E+5	8.11E+2	0.57
100^3	1.11E-1	1.07E-4	1.66E+8	2.91E+3	2.88E+8	2.86E+4	1.10E+7	2.26E+3	9.59E+6	2.66E+3	8.54E+5	2.76E+3	0.58
120^3	1.91E-1	—	2.84E+8	8.26E+3	4.90E+8	1.79E+4	2.09E+7	1.61E+4	1.87E+7	1.58E+4	1.29E+6	4.05E+3	0.58
150^3	3.63E-1	1.34E-4	5.51E+8	—	9.44E+8	2.97E+4	3.68E+7	—	3.23E+7	—	2.76E+6	—	0.58

Table A.43: PAPI Serial Mean Statistics for Kernel Madvmy_1, Variant B – Data for Figures 6.3(e)

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	3.60E-3	1.29E-5	4.99E+6	5.34E+2	9.29E+6	1.11E+3	3.44E+5	7.93E+1	2.33E+5	5.02E+2	1.04E+5	5.44E+2	0.54
50^3	1.55E-2	5.00E-6	2.17E+7	1.54E+3	3.97E+7	7.42E+2	1.54E+6	8.37E+2	9.82E+5	8.75E+3	5.12E+5	8.47E+3	0.55
80^3	5.94E-2	5.30E-5	8.57E+7	1.65E+3	1.54E+8	3.79E+3	5.48E+6	6.66E+3	3.71E+6	1.50E+4	1.48E+6	1.87E+4	0.56
100^3	1.13E-1	1.08E-4	1.65E+8	5.01E+3	2.97E+8	3.67E+4	1.12E+7	5.32E+3	8.40E+6	2.12E+4	2.21E+6	2.54E+4	0.56
120^3	1.99E-1	—	2.84E+8	2.32E+3	5.00E+8	7.20E+3	2.92E+7	1.44E+4	2.54E+7	1.24E+4	2.83E+6	4.61E+3	0.57
150^3	3.70E-1	1.36E-4	5.49E+8	—	9.61E+8	2.00E+4	4.13E+7	—	3.46E+7	—	4.81E+6	—	0.57

Table A.44: PAPI Serial Mean Statistics for Kernel Madvmz_1, Variant B – Data for Figures 6.3(f)

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	4.16E-3	8.92E-6	1.27E+6	6.62E+3	1.13E+7	1.39E+3	8.26E+4	2.88E+1	7.51E+4	6.40E+1	7.62E+3	8.42E+1	1.12E-1
50^3	1.93E-2	4.97E-5	5.80E+6	1.08E+3	5.17E+7	1.74E+3	3.65E+5	1.99E+3	3.36E+5	7.94E+2	2.68E+4	5.80E+2	1.12E-1
80^3	7.55E-2	1.17E-4	2.35E+7	4.19E+3	2.09E+8	3.91E+4	1.57E+6	2.32E+3	1.49E+6	2.04E+3	7.73E+4	2.37E+3	1.12E-1
100^3	1.45E-1	3.47E-4	4.59E+7	1.05E+4	4.09E+8	6.93E+4	3.16E+6	7.68E+3	3.02E+6	6.72E+3	1.32E+5	4.16E+3	1.12E-1
120^3	2.47E-1	6.24E-4	7.91E+7	3.50E+4	7.03E+8	9.39E+4	5.51E+6	1.20E+4	5.31E+6	9.39E+3	2.05E+5	2.32E+3	1.12E-1
150^3	4.78E-1	0.00E+0	1.55E+8	0.00E+0	1.37E+9	1.03E+5	1.08E+7	0.00E+0	1.05E+7	0.00E+0	3.61E+5	0.00E+0	1.13E-1

Table A.45: PAPI Serial Mean Statistics for Kernel Madvx_2, Variant C – Data for Figures 6.3(a)

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	4.32E-3	8.30E-6	1.28E+6	8.90E+1	1.19E+7	3.76E+2	8.36E+4	7.90E+1	7.67E+4	1.82E+2	6.98E+3	9.09E+1	1.07E-1
50^3	2.01E-2	5.99E-5	5.86E+6	3.36E+3	5.44E+7	9.78E+2	3.68E+5	8.95E+2	3.41E+5	5.85E+2	2.56E+4	6.09E+2	1.08E-1
80^3	7.89E-2	6.78E-5	2.39E+7	1.43E+4	2.21E+8	1.48E+4	1.59E+6	9.55E+2	1.52E+6	9.41E+3	7.50E+4	7.12E+2	1.08E-1
100^3	1.51E-1	3.86E-4	4.66E+7	9.26E+4	4.30E+8	6.96E+4	3.19E+6	5.17E+3	3.05E+6	6.55E+3	1.27E+5	4.08E+3	1.08E-1
120^3	2.57E-1	7.71E-4	8.03E+7	6.64E+3	7.42E+8	1.63E+4	5.59E+6	9.96E+3	5.36E+6	1.09E+4	2.03E+5	1.22E+3	1.08E-1
150^3	4.97E-1	0.00E+0	1.57E+8	0.00E+0	1.45E+9	2.04E+5	1.10E+7	0.00E+0	1.06E+7	0.00E+0	3.42E+5	0.00E+0	1.08E-1

Table A.46: PAPI Serial Mean Statistics for Kernel Madvy_2, Variant C – Data for Figures 6.3(b)

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	4.28E-3	1.07E-5	1.28E+6	7.77E+2	1.16E+7	4.27E+2	1.08E+5	1.58E+2	9.99E+4	1.10E+2	8.08E+3	1.39E+2	1.10E-1
50^3	2.00E-2	7.49E-5	5.89E+6	2.32E+4	5.29E+7	2.48E+3	4.73E+5	1.26E+3	4.29E+5	5.94E+2	4.49E+4	1.45E+3	1.11E-1
80^3	7.90E-2	2.34E-4	2.43E+7	1.59E+3	2.15E+8	3.31E+4	2.09E+6	2.81E+3	1.97E+6	5.95E+3	1.35E+5	3.88E+3	1.13E-1
100^3	1.50E-1	1.82E-4	4.65E+7	6.27E+3	4.18E+8	2.58E+4	4.06E+6	6.16E+3	3.81E+6	1.21E+4	2.42E+5	9.13E+3	1.11E-1
120^3	2.56E-1	3.85E-4	8.02E+7	9.14E+3	7.21E+8	1.37E+4	7.05E+6	2.94E+3	6.68E+6	6.68E+3	3.74E+5	1.18E+4	1.11E-1
150^3	4.91E-1	0.00E+0	1.57E+8	0.00E+0	1.40E+9	2.75E+4	1.38E+7	0.00E+0	1.33E+7	0.00E+0	5.58E+5	0.00E+0	1.12E-1

Table A.47: PAPI Serial Mean Statistics for Kernel Madvz_2, Variant C – Data for Figures 6.3(c)

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	4.89E-3	1.61E-5	4.97E+6	2.66E+3	1.12E+7	1.03E+2	3.04E+5	5.49E+1	2.30E+5	2.18E+2	7.18E+4	3.09E+2	4.44E-1
50^3	2.24E-2	6.56E-5	2.17E+7	7.40E+3	4.78E+7	5.27E+3	1.23E+6	1.12E+3	9.98E+5	1.56E+3	2.19E+5	1.20E+3	4.54E-1
80^3	8.02E-2	1.25E-4	8.57E+7	2.07E+4	1.86E+8	4.12E+4	4.70E+6	7.72E+3	4.08E+6	5.68E+3	5.99E+5	1.42E+3	4.59E-1
100^3	1.49E-1	2.36E-4	1.65E+8	1.37E+4	3.59E+8	5.37E+4	9.10E+6	5.29E+3	8.05E+6	9.15E+3	9.92E+5	5.25E+3	4.61E-1
120^3	2.48E-1	1.39E-4	2.83E+8	9.83E+4	6.13E+8	3.33E+4	1.55E+7	2.35E+4	1.39E+7	1.42E+4	1.53E+6	4.13E+3	4.62E-1
150^3	4.75E-1	0.00E+0	5.49E+8	0.00E+0	1.18E+9	2.21E+5	2.98E+7	0.00E+0	2.69E+7	0.00E+0	2.73E+6	0.00E+0	4.63E-1

Table A.48: PAPI Serial Mean Statistics for Kernel Madvmx_1, Variant C – Data for Figures 6.3(d)

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	4.97E-3	9.84E-6	4.93E+6	9.72E+2	1.21E+7	3.04E+2	2.82E+5	2.36E+2	2.40E+5	5.28E+2	4.11E+4	3.05E+2	4.06E-1
50^3	2.28E-2	7.88E-5	2.15E+7	3.92E+3	5.20E+7	3.99E+3	1.20E+6	4.49E+2	1.06E+6	2.71E+3	1.28E+5	1.63E+3	4.14E-1
80^3	8.24E-2	1.76E-4	8.49E+7	4.56E+3	2.03E+8	3.85E+4	4.77E+6	1.25E+4	4.39E+6	2.75E+3	3.62E+5	1.13E+3	4.17E-1
100^3	1.53E-1	2.41E-4	1.64E+8	1.03E+4	3.91E+8	2.40E+4	9.34E+6	1.14E+4	8.67E+6	1.12E+4	6.10E+5	2.73E+3	4.20E-1
120^3	2.56E-1	3.17E-4	2.81E+8	6.78E+3	6.69E+8	6.91E+4	1.61E+7	9.48E+3	1.51E+7	1.17E+4	9.23E+5	3.05E+3	4.20E-1
150^3	4.89E-1	0.00E+0	5.44E+8	0.00E+0	1.29E+9	9.20E+4	3.16E+7	0.00E+0	2.97E+7	0.00E+0	1.72E+6	0.00E+0	4.21E-1

Table A.49: PAPI Serial Mean Statistics for Kernel Madvmy$_1$, Variant C – Data for Figures 6.3(e)

Cells	Time (s)	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	L1M	$\sigma_{\overline{x}}$	L2H	$\sigma_{\overline{x}}$	L2M	$\sigma_{\overline{x}}$	$\frac{\text{DPOPS}}{\text{L1A}}$
30^3	5.24E-3	5.49E-7	4.99E+6	4.20E+2	1.23E+7	7.68E+2	3.75E+5	1.17E+2	2.95E+5	7.96E+3	7.91E+4	8.02E+3	4.05E-1
50^3	2.49E-2	8.31E-5	2.17E+7	1.55E+3	5.23E+7	1.67E+3	1.60E+6	2.28E+3	1.10E+6	3.02E+3	4.89E+5	3.48E+3	4.16E-1
80^3	9.45E-2	3.46E-4	8.58E+7	3.49E+3	2.05E+8	1.17E+3	6.26E+6	1.33E+4	4.40E+6	3.65E+3	1.83E+6	1.55E+3	4.18E-1
100^3	1.80E-1	1.81E-4	1.66E+8	7.71E+3	3.95E+8	9.13E+3	1.21E+7	3.70E+3	8.57E+6	1.23E+4	3.51E+6	1.11E+4	4.20E-1
120^3	3.01E-1	1.03E-4	2.84E+8	4.62E+4	6.73E+8	1.32E+4	2.12E+7	1.97E+4	1.56E+7	8.89E+4	5.40E+6	9.23E+4	4.22E-1
150^3	5.86E-1	0.00E+0	5.50E+8	0.00E+0	1.30E+9	3.85E+4	4.00E+7	0.00E+0	2.84E+7	0.00E+0	1.15E+7	0.00E+0	4.23E-1

Table A.50: PAPI Serial Mean Statistics for Kernel Madvmz$_1$, Variant C – Data for Figures 6.3(f)

	Variant A		Variant B		Variant C	
Cells	Time (s)	$\sigma_{\overline{x}}$	Time (s)	$\sigma_{\overline{x}}$	Time (s)	$\sigma_{\overline{x}}$
27 000	11.18	0.03	8.66	0.04	10.56	0.01
125 000	50.94	0.14	38.23	0.07	47.25	0.12
512 000	205.09	0.27	152.21	0.28	183.33	0.24
1 000 000	418.70	0.31	314.26	0.35	371.55	0.39
1 728 000	809.37	0.81	587.57	—	683.50	0.71
3 375 000	1941.77	0.63	1355.32	2.13	1566.48	—

Table A.51: Hydra Serial Walltimes, Minerva Intel-12.0/OpenMPI-1.4.3 – Data for Figure 6.4(a)

	Strong Scaling (150^3)						Weak Scaling (100^3)					
	Variant A		Variant B		Variant C		Variant A		Variant B		Variant C	
PEs	Time (s)	$\sigma_{\overline{x}}$	Time (s)	$\sigma_{\overline{x}}$	Time (s)	$\sigma_{\overline{x}}$	Time (s)	$\sigma_{\overline{x}}$	Time (s)	$\sigma_{\overline{x}}$	Time (s)	$\sigma_{\overline{x}}$
1	1941.77	0.63	1355.32	2.13	1565.15	0.77	418.70	0.31	314.26	0.35	372.22	0.25
2	1035.26	1.32	722.85	1.09	833.35	0.79	468.52	1.18	328.87	0.25	387.85	0.36
4	601.47	0.19	434.37	0.27	506.54	0.29	564.68	0.34	396.21	8.47	459.23	1.18
8	367.41	0.27	269.47	0.29	316.68	0.22	655.61	0.44	471.08	1.27	563.84	1.02
16	187.30	0.15	143.65	0.12	168.98	0.18	671.10	0.49	485.08	0.60	580.81	0.42
32	97.21	0.48	76.61	0.29	89.62	0.11	678.20	0.13	491.48	0.97	586.63	—
64	54.62	0.04	45.08	0.07	51.68	0.09	689.28	0.30	501.07	1.17	594.73	—
128	31.80	0.25	26.21	0.11	29.24	0.20	708.31	8.88	503.38	0.47	597.64	0.27
256	19.97	1.86	15.13	0.04	16.27	0.08	700.35	1.38	512.26	5.25	603.90	—

Table A.52: Hydra Strong and Weak-Scaling Walltimes – Data for Figures 6.4(b), 6.4(c), 6.6(a), 6.6(b)

Variant:	D				E							
Cells	Walltime	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	VECOPs	$\sigma_{\overline{x}}$	Walltime	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$
30^3	4.11E-4	2.52E-7	7.15E+5	7.97E+0	4.89E+5	6.25E+2	2.30E+5	3.14E+2	2.62E-4	1.75E-5	2.43E+5	2.27E+1
50^3	1.85E-3	5.99E-7	3.24E+6	8.36E+1	2.26E+6	3.05E+1	1.06E+6	1.34E+1	1.05E-3	1.92E-5	1.04E+6	6.41E+2
80^3	7.63E-3	1.70E-5	1.31E+7	1.07E+2	9.22E+6	3.88E+3	4.35E+6	1.92E+3	4.23E-3	2.84E-5	4.00E+6	1.91E+3
100^3	1.48E-2	1.31E-5	2.55E+7	1.06E+3	1.80E+7	9.20E+3	8.50E+6	4.60E+3	8.15E-3	3.10E-5	7.64E+6	5.38E+3
120^3	2.56E-2	2.28E-5	4.39E+7	1.62E+3	3.10E+7	2.60E+3	1.46E+7	1.54E+3	1.40E-2	1.25E-5	1.30E+7	1.20E+3
150^3	4.97E-2	1.42E-5	8.56E+7	7.72E+2	6.07E+7	5.29E+3	2.86E+7	2.64E+3	2.72E-2	1.71E-5	2.53E+7	3.73E+2

Table A.53: PAPI Serial Mean Statistics for Kernel MDT_1, Minerva, Variants D and E – Data for Table 6.4, Figure 6.5(a)

Variant:	D				E							
Cells	Walltime	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	VECOPs	$\sigma_{\overline{x}}$	Walltime	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$
30^3	1.13E-3	1.29E-6	2.88E+6	4.75E+1	1.20E+6	1.21E+2	6.01E+5	6.03E+1	6.78E-4	1.74E-5	1.27E+6	1.64E+1
50^3	5.14E-3	3.25E-7	1.32E+7	1.94E+3	5.56E+6	2.85E+1	2.78E+6	1.42E+1	2.95E-3	1.89E-5	5.76E+6	3.74E+2
80^3	2.11E-2	4.87E-5	5.38E+7	5.51E+2	2.28E+7	3.56E+3	1.14E+7	1.78E+3	1.20E-2	2.34E-5	2.33E+7	2.47E+2
100^3	4.10E-2	3.25E-5	1.05E+8	6.27E+2	4.44E+7	1.03E+4	2.22E+7	5.14E+3	2.33E-2	3.44E-5	4.53E+7	3.25E+2
120^3	7.08E-2	4.95E-5	1.81E+8	6.59E+4	7.68E+7	9.38E+3	3.84E+7	6.03E+3	4.01E-2	1.73E-5	7.81E+7	5.39E+2
150^3	1.39E-1	1.24E-3	3.53E+8	6.72E+3	1.50E+8	1.01E+4	7.49E+7	5.03E+3	7.80E-2	2.04E-5	1.52E+8	5.03E+3

Table A.54: PAPI Serial Mean Statistics for Kernel MDT_2, Minerva, Variants D and E – Data for Table 6.4, Figure 6.5(b)

Variant:	D				E							
Cells	Walltime	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	VECOPs	$\sigma_{\overline{x}}$	Walltime	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$
30^3	1.08E-3	6.94E-7	2.99E+6	7.24E+1	2.42E+6	3.75E+2	1.17E+6	1.94E+2	7.93E-4	1.90E-5	2.03E+6	2.68E+1
50^3	4.77E-3	4.69E-6	1.31E+7	1.41E+2	1.07E+7	5.74E+2	5.26E+6	2.34E+2	3.34E-3	2.12E-5	8.50E+6	4.09E+2
80^3	1.89E-2	3.66E-5	5.20E+7	5.66E+2	4.29E+7	2.12E+3	2.12E+7	1.06E+3	1.29E-2	2.93E-5	3.30E+7	7.48E+2
100^3	3.65E-2	5.08E-5	1.00E+8	2.41E+3	8.31E+7	4.71E+2	4.11E+7	1.54E+2	2.46E-2	4.63E-5	6.32E+7	4.47E+2
120^3	6.24E-2	3.11E-5	1.72E+8	3.43E+3	1.43E+8	1.57E+3	7.08E+7	9.24E+2	4.19E-2	1.69E-5	1.08E+8	2.27E+3
150^3	1.21E-1	1.90E-5	3.34E+8	5.50E+3	2.78E+8	2.31E+3	1.38E+8	8.74E+2	8.07E-2	2.45E-5	2.08E+8	2.54E+3

Table A.55: PAPI Serial Mean Statistics for Kernel UpdVel, Minerva, Variants D and E – Data for Table 6.4, Figure 6.5(c)

Variant:	D				E							
Cells	Walltime	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	VECOPs	$\sigma_{\overline{x}}$	Walltime	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$
30^3	2.57E-3	5.16E-8	2.98E+6	1.25E+2	4.44E+6	3.62E+3	2.22E+6	1.81E+3	1.65E-3	1.88E-5	1.95E+6	1.23E+2
50^3	1.22E-2	2.17E-6	1.38E+7	2.24E+2	2.06E+7	5.64E+3	1.03E+7	2.82E+3	7.64E-3	1.77E-5	8.88E+6	4.28E+2
80^3	4.73E-2	1.32E-4	5.52E+7	1.56E+2	8.37E+7	6.10E+3	4.19E+7	3.05E+3	2.95E-2	2.41E-5	3.60E+7	4.01E+2
100^3	9.60E-2	1.42E-4	1.09E+8	1.94E+3	1.65E+8	2.10E+4	8.23E+7	1.05E+4	5.96E-2	5.17E-5	7.00E+7	2.80E+3
120^3	1.60E-1	7.32E-5	1.86E+8	3.20E+3	2.83E+8	1.67E+4	1.41E+8	6.23E+3	1.00E-1	2.87E-5	1.21E+8	4.42E+3
150^3	3.15E-1	3.12E-5	3.63E+8	6.60E+3	5.54E+8	6.29E+4	2.77E+8	3.14E+4	1.96E-1	3.48E-5	2.35E+8	1.84E+3

Table A.56: PAPI Serial Mean Statistics for Kernel $Lartvis_1$, Minerva, Variants D and E – Data for Table 6.4, Figure 6.5(d)

Variant:	D				E							
Cells	Walltime	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	VECOPs	$\sigma_{\overline{x}}$	Walltime	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$
30^3	4.78E-4	2.75E-7	1.14E+6	1.72E+1	8.31E+5	3.47E+3	4.15E+5	1.73E+3	3.88E-4	2.00E-5	9.51E+5	7.28E+0
50^3	2.16E-3	4.01E-7	5.18E+6	6.31E+2	3.83E+6	2.37E+3	1.92E+6	1.19E+3	1.64E-3	2.85E-5	4.29E+6	2.74E+1
80^3	8.85E-3	2.01E-5	2.10E+7	1.79E+2	1.57E+7	4.11E+3	7.84E+6	2.05E+3	6.49E-3	3.27E-5	1.73E+7	6.41E+2
100^3	1.72E-2	2.54E-5	4.08E+7	2.94E+1	3.05E+7	2.04E+4	1.53E+7	1.02E+4	1.25E-2	3.62E-5	3.37E+7	1.02E+3
120^3	2.96E-2	2.18E-5	7.02E+7	2.24E+2	5.29E+7	2.79E+4	2.64E+7	1.61E+4	2.16E-2	2.21E-5	5.80E+7	3.79E+2
150^3	5.75E-2	1.36E-5	1.37E+8	8.16E+2	1.03E+8	2.20E+4	5.16E+7	1.10E+4	4.20E-2	2.85E-5	1.13E+8	8.43E+3

Table A.57: PAPI Serial Mean Statistics for Kernel Mdivu, Minerva, Variants D and E – Data for Table 6.4, Figure 6.5(e)

Variant:	D				E							
Cells	Walltime	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	VECOPs	$\sigma_{\overline{x}}$	Walltime	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$
30^3	2.74E-4	1.77E-7	5.48E+5	4.62E+1	4.78E+5	9.78E+0	2.37E+5	5.49E+0	2.54E-4	1.75E-5	5.92E+5	3.03E+1
50^3	1.28E-3	5.00E-7	2.38E+6	2.35E+2	2.16E+6	1.18E+2	1.07E+6	5.91E+1	1.07E-3	1.74E-5	2.52E+6	1.51E+2
80^3	5.24E-3	8.88E-6	9.38E+6	1.66E+2	8.67E+6	1.01E+2	4.32E+6	4.94E+1	3.96E-3	2.15E-5	9.75E+6	2.75E+3
100^3	1.00E-2	2.92E-6	1.81E+7	1.91E+2	1.68E+7	4.55E+2	8.38E+6	2.27E+2	7.52E-3	2.80E-5	1.87E+7	6.27E+3
120^3	1.72E-2	1.16E-5	3.10E+7	1.03E+3	2.90E+7	1.13E+3	1.44E+7	7.32E+2	1.28E-2	1.43E-5	3.20E+7	7.68E+3
150^3	3.31E-2	1.16E-5	6.00E+7	4.05E+2	5.63E+7	1.19E+3	2.81E+7	5.97E+2	2.54E-2	2.60E-5	6.19E+7	9.58E+2

Table A.58: PAPI Serial Mean Statistics for Kernel Mvolflx, Minerva, Variants D and E – Data for Table 6.4, Figure 6.5(f)

Variant:	D				E							
Cells	Walltime	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	VECOPs	$\sigma_{\overline{x}}$	Walltime	$\sigma_{\overline{x}}$	L1A	$\sigma_{\overline{x}}$
30^3	5.69E-3	5.98E-5	1.18E+7	4.86E+2	5.73E+6	1.39E+3	2.31E+6	1.61E+2	5.11E-3	2.85E-5	1.11E+7	1.44E+2
50^3	2.50E-2	8.36E-5	4.91E+7	7.66E+2	2.50E+7	1.47E+3	1.03E+7	5.26E+2	2.24E-2	4.45E-5	4.51E+7	1.81E+3
80^3	9.01E-2	2.30E-4	1.88E+8	3.06E+3	9.85E+7	1.05E+4	4.09E+7	6.88E+3	7.83E-2	1.30E-4	1.70E+8	9.80E+2
100^3	1.67E-1	2.03E-4	3.60E+8	5.13E+3	1.90E+8	2.43E+3	7.92E+7	8.41E+3	1.45E-1	1.77E-4	3.23E+8	1.60E+4
120^3	2.79E-1	2.19E-4	6.11E+8	2.38E+4	3.26E+8	8.53E+3	1.36E+8	1.68E+3	2.41E-1	6.88E-5	5.47E+8	1.55E+4
150^3	5.32E-1	1.86E-4	1.18E+9	5.06E+4	6.33E+8	4.24E+4	2.65E+8	2.12E+4	4.59E-1	1.76E-4	1.05E+9	7.38E+4

Table A.59: PAPI Serial Mean Statistics for Kernel Madvmx$_1$, Minerva, Variants D and E – Data for Table 6.4, Figure 6.5(g)

	Strong-Scaling						Weak-Scaling					
PEs	**F**	$\sigma_{\overline{x}}$	**G**	$\sigma_{\overline{x}}$	**H**	$\sigma_{\overline{x}}$	**F**	$\sigma_{\overline{x}}$	**G**	$\sigma_{\overline{x}}$	**H**	$\sigma_{\overline{x}}$
1	1565.75	1.24	1545.64	1.48	1541.94	0.28	372.09	0.22				
2	835.41	1.65	832.84	0.63	885.31	3.24	387.49	0.20	388.97	0.45	401.56	1.02
4	506.56	0.33	519.34	0.18	548.38	1.85	460.68	0.22	477.63	0.49	502.48	0.70
8	318.69	0.52	342.50	0.15	369.22	2.00	564.90	0.44	599.25	0.26	643.97	4.29
16	170.32	0.15	181.41	0.17	204.57	0.14	582.71	0.83	615.33	0.29	674.35	1.00
32	90.38	0.04	95.21	0.12	110.08	0.34	586.28	0.39	619.25	0.43	707.26	3.27
64	52.52	0.08	56.60	0.13	62.18	0.11	596.06	0.16	629.95	0.27	713.58	1.32
128	30.44	0.08	30.83	0.04	31.60	0.08	598.40	0.36	636.50	0.79	734.89	1.02
256	16.73	0.04	28.26	0.04	30.00	0.42	603.91		658.24	0.48	766.49	1.06

Table A.60: Minerva – Strong and Weak-Scaling Walltime – Data for Figures 6.6(a), 6.6(b)

Variant	G						H					
Exchange Phase	**Min (s)**	$\sigma_{\overline{x}}$	μ **(s)**	$\sigma_{\overline{x}}$	**Max (s)**	$\sigma_{\overline{x}}$	**Min (s)**	$\sigma_{\overline{x}}$	μ **(s)**	$\sigma_{\overline{x}}$	**Max (s)**	$\sigma_{\overline{x}}$
MadvExch	2.33E-02	4.00E-04	3.13E-02	1.57E-04	1.22E-01	1.68E-03	9.89E-02	1.09E-03	1.19E-01	7.75E-04	1.66E-01	1.29E-03
LartvisExch	2.15E-03	2.18E-04	9.52E-03	1.82E-04	8.25E-02	2.08E-03	9.91E-03	4.06E-04	1.71E-02	1.81E-04	3.43E-02	3.96E-04
Mlagh1Exch	1.56E-02	1.73E-04	2.45E-02	5.41E-04	6.97E-02	7.49E-04	2.56E-02	8.79E-04	3.74E-02	6.64E-04	8.22E-02	7.65E-04
Mlagh2Exch	1.19E-02	3.40E-05	1.61E-02	7.67E-05	2.40E-02	1.33E-03	4.19E-02	1.66E-03	6.07E-02	1.41E-03	7.49E-02	2.08E-03
MadvmxExch	4.44E-02	5.21E-04	7.13E-02	1.64E-04	9.05E-02	6.25E-04	5.45E-02	9.02E-04	7.98E-02	5.71E-04	1.24E-01	1.90E-04
MadvmyExch	4.35E-02	7.68E-04	7.14E-02	2.36E-04	9.01E-02	7.70E-04	5.50E-02	1.48E-03	8.17E-02	3.55E-04	1.19E-01	1.11E-03
MadvmxExch	4.24E-02	8.90E-04	7.14E-02	1.78E-04	9.03E-02	8.90E-04	5.14E-02	5.78E-04	8.01E-02	3.38E-04	1.25E-01	1.54E-03

Table A.61: Minerva – 256 PEs, 100^3, Weak-Scaling – Communication Phase Timings – Data for Figure 6.7(a)

Variant	F						H					
Exchange Phase	**Min (s)**	$\sigma_{\overline{x}}$	μ **(s)**	$\sigma_{\overline{x}}$	**Max (s)**	$\sigma_{\overline{x}}$	**Min (s)**	$\sigma_{\overline{x}}$	μ **(s)**	$\sigma_{\overline{x}}$	**Max (s)**	$\sigma_{\overline{x}}$
LartvisComp	9.82E-2	—	9.89E-2	—	1.03E-1	—	1.02E-01	7.90E-05	1.04E-01	2.03E-05	1.08E-01	2.26E-04
MadvmxComp	1.85E-1	—	2.55E-1	—	2.63E-1	—	1.84E-01	7.32E-04	2.84E-01	5.78E-04	2.99E-01	4.96E-04
Madvx1Comp	8.80E-3	—	1.45E-2	—	1.55E-2	—	8.54E-03	1.46E-04	1.35E-02	4.22E-05	1.53E-02	1.98E-04
Madvx2Comp	1.53E-1	—	1.64E-1	—	1.69E-1	—	1.94E-01	5.66E-04	2.05E-01	7.62E-05	2.17E-01	3.77E-04

Table A.62: Minerva – 256 PEs, 100^3, Weak-Scaling – Compute Kernel Timings – Data for Figure 6.7(b)

Variant	Strong-Scaling						Weak-Scaling					
	C		I		J		C		I		J	
PEs	**Walltime (s)**	$\sigma_{\overline{x}}$	**Walltime (s)**	$\sigma_{\overline{x}}$	**Walltime (s)**	$\sigma_{\overline{x}}$	**Walltime (s)**	$\sigma_{\overline{x}}$	**Walltime (s)**	$\sigma_{\overline{x}}$	**Walltime (s)**	$\sigma_{\overline{x}}$
1	—	—	1753.12	1.86	1772.94	1.00	371.95	0.31	434.60	18.08	420.65	0.84
2	—	—	948.50	2.87	1024.76	0.91	387.69	0.74	451.21	0.87	486.88	1.62
4	—	—	584.97	0.13	685.99	0.28	460.00	0.68	551.58	1.64	664.68	0.50
8	—	—	316.55	0.58	366.02	0.38	562.57	0.50	568.82	0.37	656.21	1.15
12	219.24	0.20	261.11	0.37	338.54	0.20	578.91	0.54	673.18	1.10	926.19	0.37
16	169.15	0.11	—	—	—	—	579.58	0.24	—	—	—	—
24	115.60	0.05	143.11	0.08	174.31	0.20	582.04	0.31	683.83	0.35	936.38	0.16
32	89.64	0.14	—	—	—	—	586.26	0.15	—	—	—	—
48	64.75	0.25	86.05	0.02	100.17	0.23	591.29	0.24	720.93	0.37	971.20	0.42
64	53.15	0.50	—	—	—	—	594.43	0.30	—	—	—	—
96	35.35	0.14	48.01	0.05	53.55	0.07	597.90	0.36	738.41	0.91	984.73	0.03
128	29.41	0.08	—	—	—	—	598.54	0.11	—	—	—	—
192	19.80	0.11	27.36	0.06	30.06	0.08	601.24	0.41	755.58	0.35	1000.10	0.46
256	15.93	0.08	—	—	—	—	601.83	0.01	—	—	—	—

Table A.63: Minerva, Strong and Weak-Scaling Walltimes — Variants C, I, J — Data for Figure 6.8

BlockSize	**Walltime (s)**	$\sigma_{\overline{x}}$
1	337.70	0.16
8	323.32	0.07
32	318.73	0.26
500	277.30	0.70
1000	273.29	0.46
1500	324.67	0.28
1875	264.34	0.08

Table A.64: Minerva Intel-12.0/OpenMPI-1.4.4, Dynamic Block Size Performance, 150^3, 12 Threads – Data for Figure 6.9

Variant	C		J		K		L	
PEs	**Walltime (s)**	$\sigma_{\overline{x}}$	**Walltime (s)**	$\sigma_{\overline{x}}$	**Walltime (s)**	$\sigma_{\overline{x}}$	**Walltime (s)**	$\sigma_{\overline{x}}$
1	—	—	1772.94	1.00	1561.69	1.81	1561.07	2.06
2	—	—	1024.76	0.91	964.84	2.14	974.48	3.33
4	—	—	685.99	0.28	666.40	0.56	666.71	0.32
8	—	—	366.02	0.38	354.71	0.32	367.31	0.51
12	219.24	0.20	338.54	0.20	336.44	0.21	326.94	0.23
16	169.15	0.11	—	—	—	—	—	—
24	115.60	0.05	174.31	0.20	172.73	0.13	167.07	0.71
32	89.64	0.14	—	—	—	—	—	—
48	64.75	0.25	100.17	0.23	101.53	0.19	93.30	0.20
64	53.15	0.50	—	—	—	—	—	—
96	35.35	0.14	53.55	0.07	53.79	0.13	49.27	0.09
128	29.41	0.08	—	—	—	—	—	—
192	19.80	0.11	30.06	0.08	29.55	0.22	27.63	0.16
256	15.93	0.08	—	—	—	—	—	—

Table A.65: Minerva, Intel-12.0/OpenMPI-1.4.4 – 100^3, Strong-Scaling Walltimes – Data for Figure 6.11(a)

Variant	C		J		K		L	
PEs	**Walltime (s)**	$\sigma_{\overline{x}}$	**Walltime (s)**	$\sigma_{\overline{x}}$	**Walltime (s)**	$\sigma_{\overline{x}}$	**Walltime (s)**	$\sigma_{\overline{x}}$
1	371.95	0.31	420.65	0.84	368.19	0.52	338.10	0.78
2	387.69	0.74	486.88	1.62	461.06	0.75	440.22	2.40
4	460.00	0.68	664.68	0.50	652.55	1.50	651.51	0.66
8	562.57	0.50	656.21	1.15	622.91	0.67	643.39	1.59
12	578.91	0.54	926.19	0.37	915.07	0.37	866.47	1.24
16	579.58	0.24	—	—	—	—	—	—
24	582.04	0.31	936.38	0.16	926.00	0.44	876.48	1.39
32	586.26	0.15	—	—	—	—	—	—
48	591.29	0.24	971.20	0.42	964.54	0.33	877.46	0.61
64	594.43	0.30	—	—	—	—	—	—
96	597.90	0.36	984.73	0.03	973.56	0.19	883.12	0.43
128	598.54	0.11	—	—	—	—	—	—
192	601.24	0.41	1000.10	0.46	985.88	0.33	892.63	0.54
256	601.83	0.01	—	—	—	—	—	—

Table A.66: Minerva, Intel-12.0/OpenMPI-1.4.4 – 100^3, Strong-Scaling Walltimes – Data for Figure 6.11(b)

Variant	K						L					
PEs	Min (s)	$\sigma_{\overline{x}}$	μ (s)	$\sigma_{\overline{x}}$	Max (s)	$\sigma_{\overline{x}}$	Min (s)	$\sigma_{\overline{x}}$	μ (s)	$\sigma_{\overline{x}}$	Max (s)	$\sigma_{\overline{x}}$
1	9.95E-02	1.28E-05	9.95E-02	1.28E-05	9.95E-02	1.28E-05	3.32E-02	3.32E-02	9.95E-02	4.08E-05	9.95E-02	4.08E-05
2	1.03E-01	2.85E-05	1.03E-01	2.85E-05	1.03E-01	2.85E-05	3.44E-02	3.44E-02	1.03E-01	7.98E-05	1.03E-01	7.98E-05
4	1.10E-01	1.21E-04	1.10E-01	1.21E-04	1.10E-01	1.21E-04	3.66E-02	3.65E-02	1.10E-01	1.28E-04	1.10E-01	1.28E-04
8	1.05E-01	2.29E-04	1.05E-01	6.70E-05	1.06E-01	1.24E-04	3.54E-02	3.53E-02	1.15E-01	3.99E-04	1.15E-01	6.55E-04
12	1.05E-01	6.04E-04	1.06E-01	3.24E-04	1.08E-01	1.12E-03	3.65E-02	3.58E-02	1.10E-01	1.12E-03	1.11E-01	1.43E-03
24	1.04E-01	3.38E-04	1.08E-01	3.59E-04	1.11E-01	3.11E-04	3.72E-02	3.69E-02	1.10E-01	5.98E-04	1.12E-01	7.82E-04
48	1.06E-01	4.73E-04	1.10E-01	3.71E-04	1.15E-01	6.47E-04	3.85E-02	3.80E-02	1.08E-01	2.07E-04	1.09E-01	1.11E-04
96	1.08E-01	1.79E-04	1.12E-01	3.53E-04	1.18E-01	1.04E-03	3.98E-02	3.91E-02	1.08E-01	3.35E-04	1.10E-01	1.42E-04
192	1.07E-01	2.48E-04	1.14E-01	1.49E-04	1.23E-01	3.88E-04	4.11E-02	4.08E-02	1.10E-01	2.86E-04	1.13E-01	5.08E-04

Table A.67: Minerva, Intel-12.0/OpenMPI-1.4.4 – 100^3, Weak-Scaling, Lartvis Walltimes – Data for Figure 6.12(a)

Variant	K						L					
PEs	Min (s)	$\sigma_{\overline{x}}$	μ (s)	$\sigma_{\overline{x}}$	Max (s)	$\sigma_{\overline{x}}$	Min (s)	$\sigma_{\overline{x}}$	μ (s)	$\sigma_{\overline{x}}$	Max (s)	$\sigma_{\overline{x}}$
1	9.97E-02	1.42E-05	9.97E-02	1.42E-05	9.97E-02	1.42E-05	9.95E-02	7.55E-05	9.95E-02	7.55E-05	9.95E-02	7.55E-05
2	1.03E-01	2.35E-05	1.03E-01	2.35E-05	1.03E-01	2.35E-05	1.03E-01	9.38E-05	1.03E-01	9.38E-05	1.03E-01	9.38E-05
4	1.09E-01	1.05E-04	1.09E-01	1.05E-04	1.09E-01	1.05E-04	1.09E-01	9.93E-05	1.09E-01	9.93E-05	1.09E-01	9.93E-05
8	1.03E-01	9.40E-05	1.03E-01	2.25E-05	1.04E-01	4.93E-05	9.57E-02	1.15E-03	9.73E-02	2.69E-04	9.90E-02	6.32E-04
12	1.03E-01	5.45E-05	1.03E-01	2.70E-05	1.04E-01	7.20E-06	9.88E-02	3.59E-04	9.98E-02	3.77E-04	1.01E-01	4.71E-04
24	1.02E-01	4.15E-05	1.03E-01	1.89E-05	1.04E-01	4.38E-06	8.85E-02	1.63E-03	9.36E-02	1.73E-04	1.01E-01	1.60E-03
48	1.02E-01	9.80E-06	1.02E-01	5.56E-06	1.04E-01	2.00E-05	8.61E-02	1.21E-03	8.98E-02	6.16E-04	9.60E-02	1.47E-03
96	1.02E-01	4.01E-05	1.02E-01	5.46E-06	1.04E-01	2.58E-05	8.33E-02	2.52E-04	8.81E-02	4.02E-04	9.33E-02	1.90E-04
192	1.01E-01	3.18E-05	1.02E-01	7.86E-06	1.04E-01	2.23E-05	7.89E-02	1.05E-03	8.57E-02	5.09E-04	9.15E-02	8.15E-05

Table A.68: Minerva, Intel-12.0/OpenMPI-1.4.4 – 100^3, Weak-Scaling, Lartvis_1 Walltimes – Data for Figure 6.12(a)

Variant	K						L					
PEs	Min (s)	$\sigma_{\overline{x}}$	μ s)	$\sigma_{\overline{x}}$	Max (s)	$\sigma_{\overline{x}}$	Min (s)	$\sigma_{\overline{x}}$	μ s)	$\sigma_{\overline{x}}$	Max (s)	$\sigma_{\overline{x}}$
1	1.42E-05	9.23E-08	1.42E-05	9.23E-08	1.42E-05	9.23E-08	1.45E-05	1.34E-07	1.45E-05	1.34E-07	1.45E-05	1.34E-07
2	1.76E-05	3.42E-08	1.76E-05	3.42E-08	1.76E-05	3.42E-08	1.69E-05	1.41E-07	1.69E-05	1.41E-07	1.69E-05	1.41E-07
4	2.06E-05	3.30E-08	2.06E-05	3.30E-08	2.06E-05	3.30E-08	2.01E-05	3.03E-07	2.01E-05	3.03E-07	2.01E-05	3.03E-07
8	1.05E-03	1.71E-04	1.41E-03	4.53E-05	1.77E-03	1.54E-04	1.25E-02	5.58E-04	1.44E-02	6.05E-04	1.63E-02	1.54E-03
12	1.13E-03	3.92E-05	2.03E-03	1.62E-04	2.94E-03	3.34E-04	6.95E-03	6.74E-04	8.18E-03	8.65E-04	9.41E-03	1.09E-03
24	1.45E-03	1.13E-04	3.78E-03	2.40E-04	5.26E-03	3.76E-04	5.91E-03	9.02E-04	1.38E-02	5.38E-04	1.98E-02	1.70E-03
48	3.23E-03	2.66E-04	5.51E-03	2.06E-04	7.76E-03	3.12E-04	9.47E-03	1.11E-03	1.62E-02	7.07E-04	2.04E-02	1.52E-03
96	4.70E-03	5.08E-05	7.15E-03	2.14E-04	1.03E-02	4.64E-04	1.20E-02	1.18E-04	1.82E-02	5.02E-04	2.43E-02	2.17E-04
192	4.61E-03	2.22E-04	8.75E-03	8.90E-05	1.31E-02	1.52E-04	1.55E-02	2.34E-04	2.16E-02	5.77E-04	2.89E-02	1.58E-03

Table A.69: Minerva, Intel-12.0/OpenMPI-1.4.4 – 100^3, Weak-Scaling, Lartvis Walltimes – Data for Figure 6.12(a)

Variant	K						L					
PEs	Min (s)	$\sigma_{\overline{x}}$	μ s)	$\sigma_{\overline{x}}$	Max (s)	$\sigma_{\overline{x}}$	Min (s)	$\sigma_{\overline{x}}$	μ s)	$\sigma_{\overline{x}}$	Max (s)	$\sigma_{\overline{x}}$
1	1.36E-01	2.74E-04	1.36E-01	2.74E-04	1.36E-01	2.74E-04	1.36E-01	1.75E-04	1.36E-01	1.75E-04	1.36E-01	1.75E-04
2	1.93E-01	9.35E-04	1.93E-01	9.35E-04	1.93E-01	9.35E-04	2.04E-01	2.41E-03	2.04E-01	2.41E-03	2.04E-01	2.41E-03
4	3.26E-01	1.18E-03	3.26E-01	1.18E-03	3.26E-01	1.18E-03	3.25E-01	2.41E-04	3.25E-01	2.41E-04	3.25E-01	2.41E-04
8	3.07E-01	1.29E-04	3.08E-01	2.44E-04	3.08E-01	4.24E-04	2.95E-01	8.21E-04	2.97E-01	1.48E-03	2.99E-01	2.15E-03
12	5.74E-01	9.18E-04	5.75E-01	9.88E-04	5.75E-01	1.07E-03	5.01E-01	5.04E-04	5.02E-01	5.18E-04	5.03E-01	5.76E-04
24	5.85E-01	9.67E-04	5.86E-01	5.98E-04	5.87E-01	1.51E-04	5.04E-01	1.87E-04	5.05E-01	2.61E-04	5.06E-01	7.19E-04
48	6.18E-01	4.46E-04	6.20E-01	2.56E-04	6.23E-01	3.19E-04	5.05E-01	3.36E-04	5.06E-01	1.49E-04	5.08E-01	1.40E-04
96	6.20E-01	8.89E-04	6.26E-01	3.40E-04	6.30E-01	4.35E-04	5.05E-01	3.43E-04	5.08E-01	2.18E-04	5.11E-01	1.94E-04
192	6.26E-01	3.03E-04	6.33E-01	2.02E-04	6.41E-01	1.20E-03	5.06E-01	3.32E-04	5.11E-01	1.90E-04	5.15E-01	1.60E-04

Table A.70: Minerva, Intel-12.0/OpenMPI-1.4.4 – 100^3, Weak-Scaling, Madvmx Walltimes – Data for Figure 6.12(b)

Variant	K						L					
PEs	Min (s)	$\sigma_{\overline{x}}$	μ s)	$\sigma_{\overline{x}}$	Max (s)	$\sigma_{\overline{x}}$	Min (s)	$\sigma_{\overline{x}}$	μ s)	$\sigma_{\overline{x}}$	Max (s)	$\sigma_{\overline{x}}$
1	1.27E-01	2.87E-04	1.27E-01	2.87E-04	1.27E-01	2.87E-04	1.27E-01	3.02E-04	1.27E-01	3.02E-04	1.27E-01	3.02E-04
2	1.84E-01	9.35E-04	1.84E-01	9.35E-04	1.84E-01	9.35E-04	1.95E-01	4.19E-03	1.95E-01	4.19E-03	1.95E-01	4.19E-03
4	3.13E-01	1.16E-03	3.13E-01	1.16E-03	3.13E-01	1.16E-03	3.11E-01	4.32E-04	3.11E-01	4.32E-04	3.11E-01	4.32E-04
8	2.83E-01	2.04E-04	2.84E-01	1.17E-04	2.84E-01	1.84E-04	2.50E-01	7.44E-04	2.53E-01	2.38E-03	2.55E-01	4.55E-03
12	5.45E-01	1.08E-03	5.46E-01	1.06E-03	5.48E-01	1.07E-03	4.43E-01	4.03E-04	4.44E-01	8.80E-04	4.46E-01	1.90E-03
24	5.45E-01	9.36E-04	5.46E-01	6.49E-04	5.48E-01	6.76E-04	3.97E-01	4.13E-03	4.02E-01	1.90E-03	4.10E-01	2.48E-03
48	5.41E-01	7.61E-04	5.43E-01	2.76E-04	5.44E-01	8.21E-05	3.14E-01	8.97E-03	3.26E-01	6.25E-03	3.33E-01	7.30E-03
96	5.39E-01	4.18E-04	5.42E-01	2.51E-04	5.45E-01	5.56E-04	2.49E-01	1.17E-03	2.70E-01	1.62E-03	2.98E-01	3.92E-03
192	5.39E-01	7.77E-04	5.42E-01	3.27E-04	5.45E-01	5.93E-04	2.13E-01	2.35E-03	2.37E-01	2.08E-03	2.61E-01	4.10E-03

Table A.71: Minerva, Intel-12.0/OpenMPI-1.4.4 – 100^3, Weak-Scaling, Madvmx_1 Walltimes – Data for Figure 6.12(b)

Variant	K						L					
PEs	Min (s)	$\sigma_{\overline{x}}$	μ s)	$\sigma_{\overline{x}}$	Max (s)	$\sigma_{\overline{x}}$	Min (s)	$\sigma_{\overline{x}}$	μ s)	$\sigma_{\overline{x}}$	Max (s)	$\sigma_{\overline{x}}$
1	1.47E-05	5.45E-08	1.47E-05	5.45E-08	1.47E-05	5.45E-08	1.46E-05	1.01E-07	1.46E-05	1.01E-07	1.46E-05	1.01E-07
2	1.70E-05	1.17E-08	1.70E-05	1.17E-08	1.70E-05	1.17E-08	1.73E-05	1.30E-07	1.73E-05	1.30E-07	1.73E-05	1.30E-07
4	2.00E-05	3.67E-08	2.00E-05	3.67E-08	2.00E-05	3.67E-08	2.13E-05	2.73E-07	2.13E-05	2.73E-07	2.13E-05	2.73E-07
8	1.45E-02	1.46E-04	1.48E-02	1.23E-04	1.50E-02	1.10E-04	3.30E-02	3.60E-04	3.53E-02	3.39E-04	3.76E-02	9.04E-04
12	1.43E-02	1.02E-04	1.52E-02	3.21E-04	1.60E-02	5.42E-04	4.33E-02	8.07E-04	4.41E-02	4.73E-04	4.49E-02	3.25E-04
24	2.65E-02	6.55E-04	2.81E-02	7.13E-05	2.91E-02	1.56E-04	8.36E-02	1.52E-03	9.09E-02	1.33E-03	9.51E-02	2.51E-03
48	6.73E-02	3.87E-04	6.95E-02	1.17E-04	7.11E-02	2.83E-04	1.66E-01	3.74E-03	1.72E-01	3.71E-03	1.83E-01	5.09E-03
96	7.00E-02	8.00E-04	7.61E-02	1.50E-04	8.14E-02	6.90E-04	2.02E-01	1.93E-03	2.31E-01	8.05E-04	2.55E-01	1.33E-03
192	7.48E-02	6.03E-04	8.49E-02	1.23E-04	9.49E-02	4.50E-04	2.39E-01	1.95E-03	2.67E-01	1.15E-03	2.94E-01	6.60E-04

Table A.72: Minerva, Intel-12.0/OpenMPI-1.4.4 – 100^3, Weak-Scaling, Madvmx Walltimes – Data for Figure 6.12(b)

PEs	Iteration Time (%)					
	MatMult (Max)	VecNorm (Min)	VecTDot (Min)	VecAXPY (Max)	VecAYPX (Max)	ApplyPC (Max)
1	68.67	2.45	8.63	11.50	4.83	3.92
2	63.19	3.12	9.96	11.87	6.28	5.58
4	62.02	2.54	9.08	15.23	6.53	4.59
8	57.11	1.81	10.50	17.06	7.33	6.18
16	57.14	1.76	10.82	17.06	7.40	5.83
32	57.35	1.76	10.52	16.95	7.44	5.98
64	56.99	1.85	11.39	16.54	7.45	5.79
128	57.63	1.94	11.03	16.35	7.36	5.70
256	57.53	2.08	11.33	16.10	7.32	5.63
512	57.72	2.13	11.39	15.92	7.21	5.63
1024	56.82	2.33	12.61	15.65	7.13	5.45
2048	57.63	2.24	11.30	15.96	7.29	5.58
4096	58.08	2.22	11.46	15.50	7.29	5.45
8192	55.84	3.39	12.92	15.65	6.94	5.25
16 384	57.55	3.84	12.96	14.14	6.52	4.99

Table A.73: HECToR, PGI-12.10/MPICH2-5.6.1 – CG Algorithm Breakdown by Function– Data for Figure 7.3

PEs	Time (s)			
	Multiply Compute (s)	Multiply-Add Compute (s)	VecScatterBegin (s)	VecScatterEnd (s)
1	3.15E-03	0.00E+00	0.00E+00	0.00E+00
2	4.24E-03	5.79E-04	1.90E-05	3.20E-05
4	5.40E-03	6.10E-04	4.60E-05	6.50E-05
8	8.44E-03	1.01E-03	1.25E-04	1.54E-04
16	8.38E-03	9.86E-04	1.39E-04	2.39E-04
32	8.32E-03	1.02E-03	1.60E-04	2.79E-04
64	8.34E-03	1.04E-03	1.84E-04	3.88E-04
128	8.34E-03	1.05E-03	2.01E-04	4.10E-04
256	8.36E-03	1.05E-03	2.19E-04	4.83E-04
512	8.35E-03	1.06E-03	2.47E-04	5.33E-04
1024	8.35E-03	1.06E-03	2.46E-04	5.75E-04
2048	8.35E-03	1.07E-03	2.60E-04	5.70E-04
4096	8.35E-03	1.07E-03	2.89E-04	6.73E-04
8192	8.36E-03	1.07E-03	3.34E-04	6.67E-04
16 384	8.34E-03	1.07E-03	4.08E-04	9.16E-04

Table A.74: HECToR, PGI-12.10/MPICH2-5.6.1 – Single Matrix-Multiply Call Mean Breakdown – Data for Figure 7.4

	Time (s)						
	Compute			AllReduce			
PEs	Min	μ	Max	Min	μ	Max	IMB
1	1.03E-4	1.03E-4	1.03E-4	0.00E+0	0.00E+0	0.00E+0	0.00E+0
2	2.19E-4	2.20E-4	2.21E-4	7.00E-6	1.00E-5	1.30E-5	1.20E-6
4	2.23E-4	2.24E-4	2.25E-4	1.50E-5	1.60E-5	1.60E-5	2.29E-6
8	2.76E-4	2.87E-4	2.96E-4	1.90E-5	3.30E-5	4.60E-5	3.89E-6
16	2.60E-4	2.87E-4	3.15E-4	1.90E-5	8.30E-5	1.64E-4	5.26E-6
32	2.68E-4	2.84E-4	3.03E-4	2.30E-5	1.35E-4	2.15E-4	6.65E-6
64	2.66E-4	2.85E-4	3.08E-4	4.20E-5	1.36E-4	1.87E-4	1.01E-5
128	2.61E-4	2.86E-4	3.41E-4	6.70E-5	1.57E-4	2.24E-4	1.94E-5
256	2.60E-4	2.86E-4	3.10E-4	9.90E-5	2.14E-4	3.55E-4	1.59E-5
512	2.58E-4	2.84E-4	3.13E-4	1.02E-4	2.36E-4	3.90E-4	3.00E-5
1024	2.61E-4	2.85E-4	3.31E-4	1.48E-4	2.80E-4	4.42E-4	3.58E-5
2048	2.66E-4	2.85E-4	3.26E-4	1.33E-4	2.60E-4	4.36E-4	3.83E-5
4096	2.47E-4	2.85E-4	3.18E-4	1.39E-4	2.66E-4	5.39E-4	1.20E-4
8192	2.16E-4	2.87E-4	5.05E-4	3.44E-4	5.06E-4	6.03E-4	9.84E-5
16 384	2.59E-4	2.90E-4	3.47E-4	4.91E-4	6.45E-4	7.68E-4	1.04E-4

Table A.75: HECToR, PGI-12.10/MPICH2-5.6.1 – Data for Figure 7.5

	Time (s)						
	Compute			AllReduce			
PEs	Min	μ	Max	Min	μ	Max	IMB
1	1.96E-4	1.96E-4	1.96E-4	0.00E+0	0.00E+0	0.00E+0	0.00E+0
2	3.68E-4	3.70E-4	3.73E-4	6.00E-6	8.00E-6	1.10E-5	1.20E-6
4	4.29E-4	4.33E-4	4.35E-4	9.00E-6	2.00E-5	3.10E-5	2.29E-6
8	8.76E-4	8.86E-4	8.95E-4	1.70E-5	4.90E-5	9.70E-5	3.89E-6
16	7.74E-4	8.75E-4	9.44E-4	1.70E-5	1.63E-4	4.04E-4	5.26E-6
32	8.50E-4	9.09E-4	9.44E-4	2.30E-5	1.42E-4	2.27E-4	6.65E-6
64	8.10E-4	9.09E-4	9.66E-4	6.90E-5	2.40E-4	4.89E-4	1.01E-5
128	7.36E-4	9.01E-4	9.72E-4	8.90E-5	2.97E-4	5.92E-4	1.94E-5
256	7.39E-4	9.06E-4	9.81E-4	1.43E-4	4.20E-4	7.10E-4	1.59E-5
512	7.59E-4	9.20E-4	9.83E-4	1.56E-4	4.03E-4	6.97E-4	3.00E-5
1024	7.56E-4	9.12E-4	9.79E-4	2.56E-4	5.87E-4	8.94E-4	3.58E-5
2048	7.29E-4	9.23E-4	9.97E-4	1.62E-4	4.09E-4	7.22E-4	3.83E-5
4096	7.18E-4	9.21E-4	9.95E-4	1.30E-4	5.40E-4	9.74E-4	1.20E-4
8192	7.12E-4	9.22E-4	1.02E-3	3.04E-4	7.08E-4	1.08E-3	9.84E-5
16 384	6.94E-4	9.13E-4	1.03E-3	3.67E-4	1.14E-3	1.74E-3	1.04E-4

Table A.76: HECToR, PGI-12.10/MPICH2-5.6.1 – Data for Figure 7.6

	Time Per Iteration (s)			
	Minerva		HECToR	
PEs	Base	Coalesced	Base	Coalesced
1	3.07E-03	3.45E-03	4.66E-03	5.51E-03
2	3.87E-03	4.39E-03	7.83E-03	8.87E-03
4	6.04E-03	7.43E-03	1.00E-02	1.16E-02
8	9.72E-03	1.20E-02	1.74E-02	2.06E-02
16	1.09E-02	1.20E-02	1.76E-02	2.05E-02
32	1.02E-02	1.21E-02	1.77E-02	2.05E-02
64	1.09E-02	1.22E-02	1.81E-02	2.11E-02
128	1.11E-02	1.21E-02	1.82E-02	2.11E-02
256	1.12E-02	1.21E-02	1.86E-02	2.12E-02
512	—	—	1.87E-02	2.14E-02
1024	—	—	1.91E-02	2.17E-02
2048	—	—	1.88E-02	2.14E-02
4096	—	—	1.92E-02	2.15E-02
8192	—	—	1.98E-02	2.24E-02
16 384	—	—	2.11E-02	2.33E-02

Table A.77: Minerva (Intel-12.0/OpenMPI-1.4.3), HECToR (PGI-12.10/MPICH2-5.6.1) – Data for Figures 7.7(a), 7.7(b)

Function	Time (s)	
	Base	**Coalesce**
MatMult (Max)	112.53	111.50
VecNorm (Min)	7.51	10.95
VecDot (Min)	25.34	23.00
VecAXPY (Max)	27.64	32.13
VecAYPX (Max)	12.75	26.58
ApplyPC (Max)	9.76	12.95

Table A.78: HECToR (PGI-12.10/MPICH2-5.6.1), 16384 Cores, Weak-Scaling 50^3, CG Function Breakdown – Data for Figures 7.8

PEs	Time (s)					
	Base			**Coalesced**		
	Min	μ	Max	Min	μ	Max
1	1.03E-04	1.03E-04	1.03E-04	1.63E-04	1.63E-04	1.63E-04
2	2.19E-04	2.20E-04	2.21E-04	2.75E-04	2.75E-04	2.75E-04
4	2.23E-04	2.24E-04	2.25E-04	3.46E-04	3.50E-04	3.54E-04
8	2.76E-04	2.87E-04	2.96E-04	7.92E-04	7.95E-04	8.00E-04
16	2.60E-04	2.87E-04	3.15E-04	5.93E-04	7.27E-04	8.03E-04
32	2.68E-04	2.84E-04	3.03E-04	7.46E-04	7.81E-04	8.03E-04
64	2.66E-04	2.85E-04	3.08E-04	5.04E-04	7.42E-04	8.32E-04
128	2.61E-04	2.86E-04	3.41E-04	4.81E-04	7.60E-04	8.38E-04
256	2.60E-04	2.86E-04	3.10E-04	4.78E-04	7.52E-04	8.38E-04
512	2.58E-04	2.84E-04	3.13E-04	7.05E-04	7.85E-04	8.38E-04
1024	2.61E-04	2.85E-04	3.31E-04	6.74E-04	7.79E-04	8.39E-04
2048	2.66E-04	2.85E-04	3.26E-04	6.99E-04	7.93E-04	8.67E-04
4096	2.47E-04	2.85E-04	3.18E-04	4.35E-04	7.77E-04	8.47E-04
8192	2.16E-04	2.87E-04	5.05E-04	3.99E-04	7.73E-04	8.83E-04
16 384	2.59E-04	2.90E-04	3.47E-04	4.82E-04	7.62E-04	8.76E-04

Table A.79: HECToR (PGI-12.10/MPICH2-5.6.1) – Data for Figures 7.9(a)

PEs	Time (s)					
	Base			**Coalesced**		
	Min	μ	Max	Min	μ	Max
1	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
2	7.00E-06	1.00E-05	1.30E-05	8.00E-06	1.10E-05	1.40E-05
4	1.50E-05	1.60E-05	1.60E-05	1.30E-05	2.90E-05	5.20E-05
8	1.90E-05	3.30E-05	4.60E-05	2.50E-05	4.30E-05	7.20E-05
16	1.90E-05	8.30E-05	1.64E-04	2.50E-05	2.38E-04	5.78E-04
32	2.30E-05	1.35E-04	2.15E-04	6.40E-05	2.10E-04	4.62E-04
64	4.20E-05	1.36E-04	1.87E-04	1.28E-04	5.33E-04	1.19E-03
128	6.70E-05	1.57E-04	2.24E-04	1.79E-04	6.16E-04	1.35E-03
256	9.90E-05	2.14E-04	3.55E-04	1.75E-04	7.70E-04	1.35E-03
512	1.02E-04	2.36E-04	3.90E-04	1.75E-04	7.46E-04	1.33E-03
1024	1.48E-04	2.80E-04	4.42E-04	4.00E-04	1.07E-03	1.73E-03
2048	1.33E-04	2.60E-04	4.36E-04	1.79E-04	7.61E-04	1.37E-03
4096	1.39E-04	2.66E-04	5.39E-04	2.27E-04	8.30E-04	1.55E-03
8192	3.44E-04	5.06E-04	6.03E-04	3.83E-04	1.17E-03	1.98E-03
16 384	4.91E-04	6.45E-04	7.68E-04	3.50E-04	1.65E-03	2.44E-03

Table A.80: HECToR (PGI-12.10/MPICH2-5.6.1) – Data for Figures 7.9(b)

	Time (s)					
	Base			Coalesced		
PEs	Min	μ	Max	Min	μ	Max
1	1.96E-04	1.96E-04	1.96E-04	3.50E-04	3.50E-04	3.50E-04
2	3.68E-04	3.70E-04	3.73E-04	5.98E-04	6.00E-04	6.02E-04
4	4.29E-04	4.33E-04	4.35E-04	8.62E-04	8.64E-04	8.67E-04
8	8.76E-04	8.86E-04	8.95E-04	1.68E-03	1.72E-03	1.73E-03
16	7.74E-04	8.75E-04	9.44E-04	1.65E-03	1.69E-03	1.72E-03
32	8.50E-04	9.09E-04	9.44E-04	1.66E-03	1.69E-03	1.71E-03
64	8.10E-04	9.09E-04	9.66E-04	1.60E-03	1.67E-03	1.73E-03
128	7.36E-04	9.01E-04	9.72E-04	1.59E-03	1.67E-03	1.73E-03
256	7.39E-04	9.06E-04	9.81E-04	1.60E-03	1.67E-03	1.74E-03
512	7.59E-04	9.20E-04	9.83E-04	1.60E-03	1.67E-03	1.75E-03
1024	7.56E-04	9.12E-04	9.79E-04	1.59E-03	1.67E-03	1.75E-03
2048	7.29E-04	9.23E-04	9.97E-04	1.58E-03	1.67E-03	1.75E-03
4096	7.18E-04	9.21E-04	9.95E-04	1.60E-03	1.67E-03	1.75E-03
8192	7.12E-04	9.22E-04	1.02E-03	1.59E-03	1.68E-03	1.84E-03
16 384	6.94E-04	9.13E-04	1.03E-03	1.60E-03	1.68E-03	1.82E-03

Table A.81: HECToR (PGI-12.10/MPICH2-5.6.1) – Data for Figures 7.10(b)

	Time (s)					
	Base			Coalesced		
PEs	Min	μ	Max	Min	μ	Max
1	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
2	6.00E-06	8.00E-06	1.10E-05	5.00E-06	8.00E-06	1.00E-05
4	9.00E-06	2.00E-05	3.10E-05	1.10E-05	1.40E-05	1.70E-05
8	1.70E-05	4.90E-05	9.70E-05	1.50E-05	2.40E-05	6.70E-05
16	1.70E-05	1.63E-04	4.04E-04	1.40E-05	4.50E-05	8.50E-05
32	2.30E-05	1.42E-04	2.27E-04	2.20E-05	5.00E-05	7.90E-05
64	6.90E-05	2.40E-04	4.89E-04	3.80E-05	1.03E-04	1.69E-04
128	8.90E-05	2.97E-04	5.92E-04	4.80E-05	1.12E-04	1.89E-04
256	1.43E-04	4.20E-04	7.10E-04	6.00E-05	1.27E-04	2.00E-04
512	1.56E-04	4.03E-04	6.97E-04	5.70E-05	1.37E-04	2.09E-04
1024	2.56E-04	5.87E-04	8.94E-04	1.20E-04	1.97E-04	2.75E-04
2048	1.62E-04	4.09E-04	7.22E-04	7.10E-05	1.55E-04	2.46E-04
4096	1.30E-04	5.40E-04	9.74E-04	8.50E-05	1.60E-04	2.38E-04
8192	3.04E-04	7.08E-04	1.08E-03	4.37E-04	6.03E-04	6.88E-04
16 384	3.67E-04	1.14E-03	1.74E-03	6.54E-04	7.91E-04	8.66E-04

Table A.82: HECToR (PGI-12.10/MPICH2-5.6.1) – Data for Figures 7.10(b)

APPENDIX B

Other Validation Data

To verify that the L1 Data Cache Accesses can be transferable between machines, two different chips were used in Table B.3 to contrast their reported values. In addition, reports [100, 112, 138] of potential inaccuracies for reported DPOPS on SandyBridge let to a comparison of Nehalem vs Westmere vs Sandybridge to determine whether such an overprediction can also occur for the Minerva readings (Westmere). Table B.4 shows that the large overprediction only appears to occur for Sandybridge in the kernels with more L1 cache misses, as reported. Since the sources above also report a potential overprediction of up to 5% for Nehalem, the results included in this work must be considered with this in mind. Nevertheless, they are still very useful for comparisons and trends between kernels.

B.1 OpenMPI Comparison

PEs	OpenMPI 1.4.3		OpenMPI 1.4.4	
	Walltime (s)	$\sigma_{\overline{x}}$	Walltime (s)	$\sigma_{\overline{x}}$
1	1941.77	0.63	1942.90	2.82
2	1035.26	1.32	1034.31	0.73
4	601.47	0.19	600.47	0.76
8	367.41	0.27	366.48	0.33
16	187.30	0.15	187.05	0.34
32	97.21	0.48	96.40	0.04
64	54.62	0.04	54.92	0.17
128	31.80	0.25	31.74	0.17

Table B.1: OpenMPI 1.4.3 vs 1.4.4 Hydra Walltime Comparison – Strong Scaling

B.2 Hydra Critical Path by Function

Problem	Total Iterations	Mlag Iterations				MDT (s)	Mlagh (s)	Madv (s)	Shortprint (s)	Memory (s)	Walltime (s)	Sum (s)	Diff(%)
		1	2	3	4								
30^3	209	209	0	0	0	1.19	1.13	7.80	0.41	0.65	11.18	11.17	−0.06
50^3	209	193	16	0	0	5.48	5.73	34.69	1.83	3.20	50.94	50.92	−0.04
80^3	210	169	17	10	14	21.84	31.50	134.61	7.18	10.08	205.09	205.21	0.06
100^3	217	157	18	10	32	44.70	77.47	264.74	14.80	17.51	418.70	419.22	0.12
120^3	229	148	17	10	54	80.38	164.88	508.38	26.34	29.54	809.37	809.52	0.02
150^3	258	136	18	10	94	176.99	446.34	1199.70	58.31	60.63	1941.77	1941.97	0.01

Table B.2: Minerva, Serial, Time spent by Function

B.3 PAPI Behaviour

Problem	Madvz$_2$					Madvmz$_1$				
	Intel X5550		Intel X3430			Intel X5550		Intel X3430		
	L1 DCA	$\sigma_{\overline{x}}$	L1 DCA	$\sigma_{\overline{x}}$	% Diff.	L1 DCA	$\sigma_{\overline{x}}$	L1 DCA	$\sigma_{\overline{x}}$	% Diff.
30^3	1.70E+7	3.76E+4	1.69E+7	6.43E+3	−0.12	8.41E+6	5.51E+3	8.36E+6	2.06E+3	−0.52
50^3	7.47E+7	5.33E+4	7.49E+7	5.25E+3	0.23	3.60E+7	6.72E+4	3.58E+7	8.74E+3	−0.60
80^3	2.99E+8	4.99E+4	2.99E+8	3.17E+4	−0.04	1.42E+8	1.25E+5	1.42E+8	7.38E+4	−0.19
100^3	5.81E+8	5.25E+5	5.81E+8	1.65E+5	0.02	2.76E+8	1.88E+5	2.74E+8	6.10E+4	−0.61
120^3	9.96E+8	1.01E+6	9.97E+8	3.71E+5	0.01	4.79E+8	4.80E+4	4.78E+8	1.12E+5	−0.25
150^3	1.94E+9	5.87E+5	1.94E+9	4.36E+5	0.03	9.38E+8	7.61E+5	9.33E+8	1.72E+5	−0.52

Table B.3: Comparison of Measured L1 Data Cache Accesses

Problem	MDT$_1$						Madvmz$_1$					
	Nehalem		Westmere		Sandy Bridge		Nehalem		Westmere		Sandy Bridge	
	DPOPs	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$	DPOPs	$\sigma_{\overline{x}}$
30^3	3.25E+5	1.83E+1	3.25E+5	2.14E+1	3.26E+5	1.25E+1	4.91E+6	2.82E+2	4.91E+6	3.55E+2	5.40E+6	6.41E+3
50^3	1.50E+6	2.71E+2	1.50E+6	1.69E+1	1.51E+6	6.71E+2	2.17E+7	1.73E+2	2.18E+7	2.13E+4	2.56E+7	2.43E+4
80^3	6.16E+6	5.35E+1	6.17E+6	3.70E+2	6.19E+6	2.17E+2	8.79E+7	3.87E+3	8.63E+7	1.11E+5	1.08E+8	3.87E+4
100^3	1.20E+7	4.85E+1	1.20E+7	3.26E+2	1.21E+7	5.01E+2	1.70E+8	2.11E+4	1.67E+8	3.77E+4	2.11E+8	4.66E+5
120^3	2.08E+7	1.05E+1	2.08E+7	2.11E+2	2.09E+7	1.06E+4	2.90E+8	1.76E+4	2.87E+8	1.61E+5	3.59E+8	3.18E+5
150^3	4.06E+7	6.95E+1	4.06E+7	—	4.08E+7	2.31E+4	5.69E+8	5.68E+4	5.57E+8	—	7.23E+8	4.12E+5

Table B.4: Comparison of Measured DPOPs across Architectures