Department of Computer Science

Porting and optimising the Isca climate model on Intel and ARM processors

George William Lancaster

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in the Faculty of Engineering

July 2019 | CSMSC-19

Declaration:

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

George William Lancaster, July 2019

Contents

List of Figures

List of Tables

Abstract

This research project explores...

This project provides ports of the Isca climate model to four supercomputers, making over 26,000 cores available for climate research at the University of Bristol. The limitations of the model have been identified, and a Additionally, it provides a valuable benchmark of the ThunderX2 processor in a production environment, contributing to current research.

Part I

Introduction and Background

CHAPTER 1

Introduction

1.1 High Performance Computing

- What is the problem?
- How have I solved it?

High Performance Computing (HPC) is an interdisciplinary field within computer science that uses large distributed supercomputers to solve complex problems across many scientific domains. In academia, HPC is synonymous with scientific computing and has applications including drug discovery, artificial intelligence and the simulation of the natural world.

Supercomputers are usually comprised of many compute nodes, each containing a number of high performance server microprocessors. Recent improvements to computational hardware have been notoriously difficult to utilise, and many scientific codes remain unoptimised throughout many years of service. This trend means that there can be poor '*price per performance*' of hardware, as the latest features available to modern processors are underutilised.

Climate models are an important tool for understanding the global atmospheric behaviour of Earth and other celestial bodies. They provide a medium for the reproduction of past, present and future meteorological events, and offer insight into previously unobservable phenomena. However, complex simulations can take thousands of hours to return substantial results, limiting the speed of research.

Climate models have many parameters governing simulated physical processes. Many of these parameters are well defined by observation, however there are equally many that are idealised, and do not correspond to processes found in the real world. Selecting values for such parameters is non-trivial, and is usually achieved using a brute-force approach known as a 'perturbed physics ensemble', which involves running many simulations using a range of parameter configurations. This strains supercomputer resources, as jobs are typically submitted in large batches, with each job taking many hours to complete.

1.2 Aims and Objectives

This research project aims to present a comprehensive performance analysis of the Isca climate model on both Intel and Arm processors. The model must be ported to four HPC systems, and optimised with the goal of reducing the models runtime. To meet this aim, the following objectives have been identified:

Port Isca to three new HPC systems Isca must be ported from the BlueCrystal phase 4 (BCP4) supercomputer to three other HPC systems: BlueCrystal phase 3 (BCP3), BluePebble (BP) and Isambard. BCP3, BCP4 and BP are based on the Intel x86-64 architecture, and Isambard is based on Arm's ARMv8 instruction set architecture. Isca is dependant on many libraries that are not yet to be available on the Arm machine. Identifying and porting these libraries comprised a significant part of the project.

Characterisation of the Isca code Isca must be benchmarked and profiled using a variety of performance analysis tools to identify the code's limitations. The resulting data must be used to plan at least two performance optimisations. Additionally, runtimes on each system will be measured to find how the total program runtime varies as a function of cell resolution and number of processor cores.

Optimisation of Isca on each system All identified performance optimisations must be implemented to a high standard. All code modifications must follow the same style and naming conventions as found in the rest of the codebase.

Analysis of Optimisations To ensure that the optimisations improve the model's performance, the optimised code must be recharacterised and compared to the unoptimised model. To be deemed successful, an optimisation must deliver a significant improvement to performance and must generate the same output as the unoptimised code. This will verify that the application logic is unchanged. It is also important to measure the performance portability of the optimisations, as a performance improvement on one machine may not carry over to another.

1.3 Contributions

The work presented in this thesis makes the following contributions to the Isca codebase, and the wider area of high-performance computing:

Provision of additional compute resources Prior to this research project, University of Bristol researchers could only access Isca on BCP4, one of five supercomputers available to the university. By providing ports of Isca to other systems, over 14,000 additional cores have be made available for climate research. Not only has this eased congestion on BCP4, but it also makes Isca more accessible to other research groups outside of the University of Bristol. Additionally, the meteorological research group at the University of Bristol has purchased a dedicated £10,000 compute node for the BluePebble supercomputer as a direct result of the work carried out in this research project.

Comprehensive performance analysis A scaling study has been performed, which shows how the total program runtime varies as a function of cell resolution and number of processor cores. This is important for researchers as it allows them to make an informed decision when selecting the number of cores to run different model resolutions. Additionally, a number of performance bottlenecks have been identified, which adds to the understanding of the code.

Optimisation of legacy code for modern hardware This research project demonstrates that Isca does not utilise many of the new hardware features available to modern processors. Specifically, there are many loops found within a bespoke Fast Fourier Transform (FFT) do not make use of vector instructions. As a result of this observation, this FFT has been replaced with a call to the Fastest Fourier Transform in the West (FFTW) library, producing a code speedup of up to 1.17× the original implementation.

Increase speed of research By halving the default precision of floating point numbers, Isca can better utilise vector registers. This presents a performance speedup of 1.69× the original code, and a speedup of 1.78× when used in conjunction with FFTW.

Contribution to hardware development Recent developments in consumer mobile hardware have resulted in a new generation of HPC-optimised Arm processors. This research project provides a comparison of these new processors and the current state-of-the-art Intel pro-

cessors. This is vital to reduce the cost of components and to drive further innovation. This study is a continuation of previous work by other authors, and provides insight into the performance of the hardware on a production scientific code (citation).

Contribution to cluster development The BluePebble cluster was still in its Beta phase of development throughout this research project. All results collected on this machine have been used to influence important design decisions, including the default stack-size limit and default memory limit for jobs submitted using the PBS job scheduler. Additionally, all dependencies required by Isca have been installed as modules using the build configurations defined by this research project, and these are freely available to use by other users of the systems.

Contribution to the Isca codebase All modifications made to the Isca codebase as a result of the work carried out in this project have been integrated back into the public Github repository in a series of pull-requests (reference).

Documentation and support to researchers Documentation has been written to help researchers compile and run Isca on different machines (TODO)

1.4 Heading

The work presented in this thesis was carried out over the the summer of 2019 alongside the University of Bristol HPC internship program.

CHAPTER 2

Climate modelling

Software development for scientific applications is a multidisciplinary task, and requires knowledge of both computer programming and the scientific domain for which the software is being developed. Although a comprehensive understanding of the mathematics used to simulate climate is not necessary to understand the work presented in this thesis, some domain knowledge is required. This chapter presents a summary of climate modelling codes, and provides an extensive overview of the workings of Isca.

2.1 The Isca climate model

Isca is an open-source framework for the modelling of the global circulation of planetary atmospheres. It was developed over four years by the climatology research group at the University of Exeter, with version 1 of the model released in 2017 [1]. The development of Isca was funded by the Natural Environment Research Council, the Engineering and Physical Sciences Research Council (EPSRC) and the UK Met Office [1].

The main goal of Isca is to deliver a user-configurable climate model, that allows for the simulation of both simple and complex scenarios, including those vastly different from Earth. The model has been used to provide evidence for numerous peer-reviewed publications, including the study of monsoons, tidally-locked planets and variations in the seasons [2, 3, 4].

The model itself spans over 260,000 significant lines of code, and is composed primarily of Fortran with some calls to ANSI C, and a Python interface for usability. Isca can be compiled and run on any system with NetCDF and MPI libraries, although a supercomputer is required for anything more than simple experimentation [1].

Because of its growing use as an academic research tool, it is of the utmost importance that the code is portable to a wide variety of computer architectures, and maintains a degree of performance portability. This will allow for the model to be used for research at other institutions and will drive future development, as Isca now has a well-established global user base. .

2.1.1 Global Circulation Model (GCM)

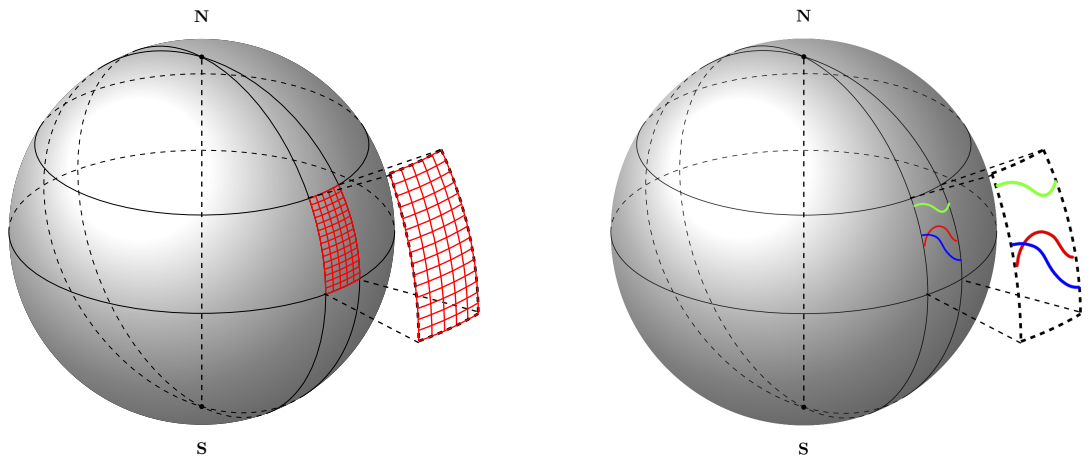
Although Isca is a new model, much of the code responsible for atmospheric simulation has been adapted from the twenty-one-year-old Flexible Modelling System (FMS), on which many modern climate models have been developed [5, 6, 7]. The FMS is a Global Circulation Model (GCM), and handles aspects of simulation including parallelisation, input and output, data exchange between model grids and the orchestration of time stepping [8].

GCMs simulate the changes in global climate behaviour over time using the set of primitive dynamical equations of motion and state, first described by Vilhelm Bjerknes in the early 20th century [9, 10, 11]. These equations include the hydrodynamic state equation, mass conservation, and thermal energy equations, which govern the distribution of energy in the atmosphere [1, 10]. In theory, these equations are applied in a continuous domain on the whole real line,

however this is not possible to do in simulation due to the restrictions imposed by finite memory resources. To bypass this issue, GCMs decompose the problem domain using a grid-point or spectral representation.

2.1.2 Domain decomposition

Grid-point models discretely represent data, decomposing the problem domain into a three-dimensional structured grid, on which the primitive dynamical equations are applied at each time step of the simulation. Structured grid codes often have high spatial locality, with interactions between cells limited to adjacent neighbours only. This property means that they tend to be highly scalable, due to various spatial decomposition methods that utilise distributed machines effectively.



(a) Grid-point model decomposition

(b) Spectral model decomposition

Figure 2.1: A simplified example of a grid-point and spectral model projected onto the sphere. This example uses just 3 waves, which implies a truncation of T3. Most spectral models use a resolution in excess of 21 waves (T21).

Spectral models represent the spatial variations of atmospheric variables as a finite series of waves at various wavelengths, whereby each wave represents the coefficients of a known function. They are typically used for global climate modelling rather than regional weather prediction as wave functions and spherical harmonics operate over a spherical domain. Because of this, all waves must be periodic so that they wrap around the sphere with the start and finish point using the same value. This places some restrictions on the types of algorithms that can be used, and can add an additional overhead to compute costs as the model must convert into a spatial representation for analysis.

Calculating the equations of motion requires solving many partial derivatives in space. Partial derivatives of waves are calculated by summing the derivatives of each basis function, providing an exact result. In contrast, grid-point models must solve partial derivatives by finite differences, and therefore require a higher resolution to provide a comparable degree of accuracy [12].

Resolution

The spatial resolution of a climate model describes the variation in the total amount of data that is used for a given problem size. The resolution of a grid-point model describes the number of

grid cells that the model operates over. A higher cell resolution implies a greater number of cells contained within a grid. Spectral models vary their resolution using a truncation, which refers to the number of waves used to define atmospheric variables. In both cases, there is a trade-off between resolution and model runtime whereby higher resolutions generally result in longer runtimes, but more accurate results.

Climate models can also vary a temporal resolution, which refers to amount of model time that passes in the simulation between calculations. Similarly to spatial resolution, the computational intensity of the simulation is influenced by the granularity of the temporal resolution. Smaller time steps more accurately represent continuous time, but result in longer runtimes.

Both grid-point and spectral models are usually classified as a strong scaling problem, for which the solution time varies with the number of processors for a fixed problem size [9]. This implies that the runtime decreases as the number of processor cores increases, however this is not always the case and is dependant on the problem domain.

2.1.3 Fast Fourier Transform (FFT)

The Isca modeluses both grid-point and spectral methods for domain decomposition. A grid-point representation is used for time-stepping, and the physics simulation is applied in the spherical and frequency domains. To convert between these two states, a FFT is used to compute the Discrete Fourier Transform (DFT) of the grid-point representation, and the Inverse Discrete Fourier Transform (IDFT) of the spectral representation. Although the cost of doing this transformation can be relatively high, it often results in a net computational saving, and can produce more accurate data at lower resolutions (citation).

The FFT algorithm is found across many different scientific domains, and as such writing optimised FFT code is a research topic in and of itself. There are multiple highly optimised FFT libraries available, and there are many different approaches to applying the algorithm, each with their own benefits and drawbacks. Formally the DFT transforms a sequence of N complex numbers $\{x_n\} = \{x_0, x_1, \ldots, x_{N-1}\}$ into another sequence of complex numbers $\{X_k\} = \{X_0, X_1, \ldots, X_{N-1}\}$, defined in Equation 2.1.

$$\mathscr{F}(x) = X_k = \sum_{n=0}^{N-1} x_n \cdot [\cos(2\pi kn/N) - i \cdot \sin(2\pi kn/N)] \tag{2.1}$$

The DFT is invertible, which means that any complex vector whereby $N > 0$ has both a DFT and IDFT of the same form as the original vector. The IDFT is given in Equation 2.2.

$$I\mathscr{F}(x) = x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{i2\pi kn/N} \tag{2.2}$$

2.2 Software architecture

The Isca codebase is vast, being composed of over 290 Fortran90 source files. This is far too much code to review for a project of this scope, however the following section provides a gentle introduction to the software architecture of the model.

2.2.1 General overview

Isca is compiled and run using its own Python library, which is used to populate various Bash scripts, Fortran namelists and other miscellaneous files with data entered into multiple dictionaries in the Python code. This was a design decision based on the usability of Python in comparison to the underlying Fortran model, and allows for a lower barrier to entry in terms of technical ability for climate researchers [1].

Compiling the model using the Python library produces a single executable that is repeatedly run for a number of iterations defined in a Python script. Typically, each iteration lasts for approximately one model month, usually simplified to 30 model days. When run in parallel the diagnostic output is distributed, which means that each processor writes its own files. Upon completion, the data generated by the previous month's simulation is combined into a single file, and is used as an input to the following month. This process is summarised by a flowchart in Figure 2.2. The large number of Python and Bash scripts used to create directories, and populate and move supporting files means that the executable cannot be run alone.

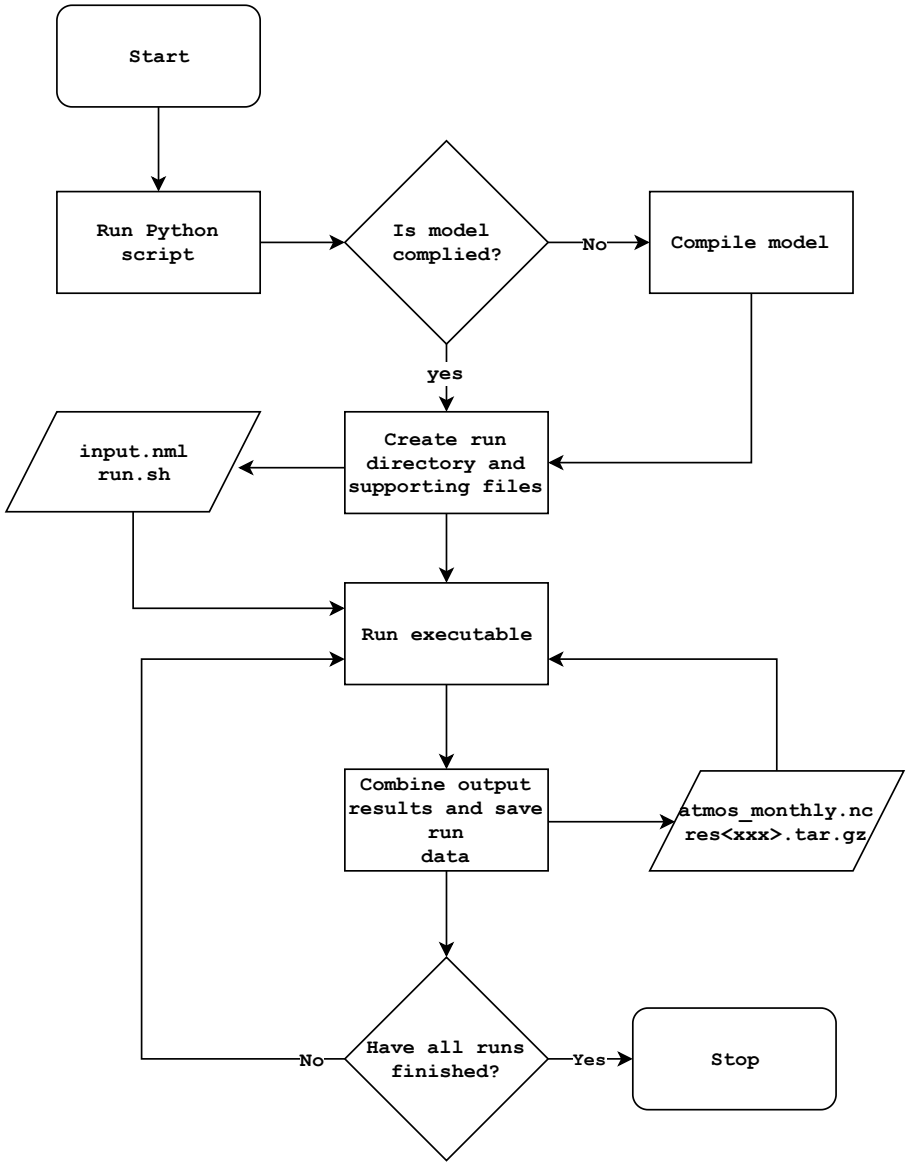


Figure 2.2: Flowchart that illustrates the program flow of Isca, when run using its Python library.

The executable itself follows an 'atmosphere integration loop', whereby the state of the atmosphere is computed for a predefined number of timesteps. Isca is modular, which means that the atmosphere can be simulated using a wide range of different techniques and algorithms at varying degrees of complexity and realism, however the most basic atmosphere integration loop is visualised as pseudocode in Listing 2.1.

```
Time_next = Time + Time_step

if(idealized_moist_model) then
   call idealized_moist_phys(...)
else
   call hs_forcing(...)
endif

call spectral_dynamics(Time, ...)

if(dry_model) then
  call compute_pressures_and_heights(x, z, ...)
else
  call compute_pressures_and_heights(x, y, ...)
endif

call spectral_diagnostics(Time_next, ...)

previous = current
current  = future
```

Listing 2.1: Pseudocode for the atmospheric integration loop found in Isca.

Of greatest interest is the `spectral_dynamics` subroutine, which comprises around 95% of the wallclock runtime of any given simulation. This subroutine contains the code for calculating the atmospheric variables, which involves communication between processors and a number of FFTs.

Isca's spectral model decomposes the horizontal grid into latitude bands, with each band assigned to a processor. When only two processors are used, the grid is split into Southern and Northern Hemispheres [13]. This method of domain decomposition implies that atmospheric variables at the edge cases of each latitudinal band (halo points) must be exchanged with other processors in a process known as a synchronised halo exchange. This allows for parallelism in the Isca model at the cost of an additional overhead incurred by the communication itself. The halo exchange simply interrupts the computational flow of the program, and allows for the exchange of the halo points before the simulation can resume. Figure 2.3 shows a simplified communication pattern similar to that found in Isca.

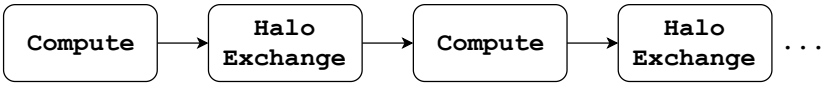


Figure 2.3: Simple communication pattern; sections of compute are are interrupted by calls to a halo exchange.

2.2.2 Dependencies

2.2.3 Fortran Libraries

Isca relies on MPI for interprocess communication, and NetCDF for data storage. These are two technologies that are commonly used in high performance computing and offer interfaces for both the ANSI C and Fortran programming languages.

MPI Message Passing Interface (MPI) is a standardised, portable interface for interprocess communication that allows for direct data transfer between processors without relying on shared memory. The MPI standard has been implemented by numerous companies and organisations, but the most commonly used are OpenMPI, MVAPICH, MPICH, and Intel MPI. All MPI implementations provide the same function calls and interfaces, and can therefore be used interchangeably.

NetCDF Network Common Data Format (NetCDF) is a platform independent binary file type that is commonly used to store and analyse scientific data. NetCDF binary files are self-describing, meaning that they contain the all the necessary information to interpret the data they store. This makes NetCDF files highly portable as a file written on one computer can be read by another without context or specialist tools, aside from the NetCDF library itself. If compiled using an MPI library, NetCDF can provide parallel IO. NetCDF itself is dependant on the HDF5 and zlib libraries, which are used for storage and data compression respectively. When compiling the NetCDF library or any program that uses it, the same compiler must be used to compile HDF5, zlib, NetCDF and the program itself. One of the advantages of NetCDF is that there are many programs available to visualise the data they store. Figure 2.4 show an example of a NetCDF file produced by Isca.

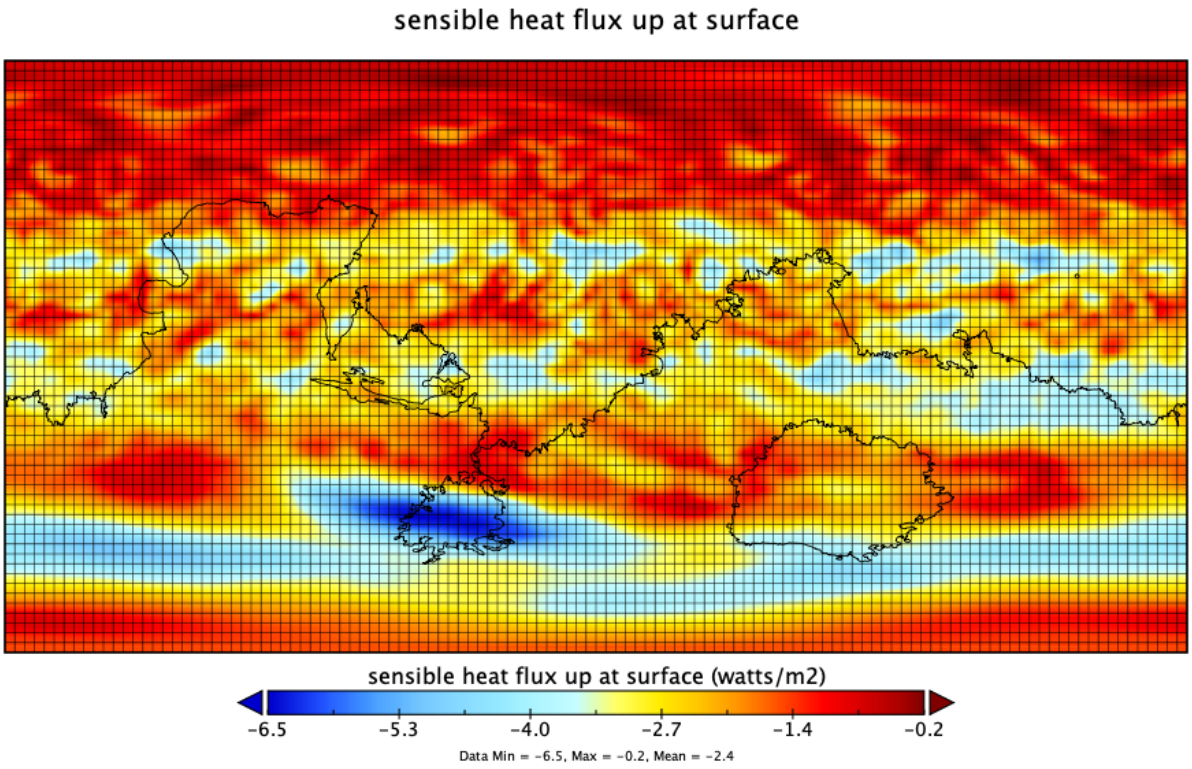


Figure 2.4: Visualisation of an output of the Grey-Mars configuration, showing the amount of heat transferred per unit area per unit time to the Martian surface after 690 days. This visualisation was generated using the Panoply NetCDF data viewer tool (citation).

2.2.4 Python Libraries

Due to the simplicity of the language, Isca uses a Python interface to create and run different model configurations. To do this, it uses a number of popular Python libraries that are commonly available on many different platforms.

Numpy Numpy is a mathematics library for Python that allows for the manipulation of N-dimensional arrays.

sh A full-featured subprocess replacement for Python. It allows for Bash commands to be issued from Python code `https://amoffat.github.io/sh/`.

Jinja2 Jinja2 is a templating language for Python that is typically used in web design. It has been used in Isca to populate a number of Bash script templates with data defined in a series of Python dictionaries `http://jinja.pocoo.org/docs/2.10/`.

f90nml A Python module and command line tool for reading, writing and modifying Fortran namelist files `https://pypi.org/project/f90nml/`.

CHAPTER 3

HPC hardware and parallel processing

There are many different techniques and processor designs that allow for a program to be run in parallel. This chapter presents a brief but thorough overview of some that have been used throughout this research project.

3.1 Parallel processing

To allow for programs to be run in parallel, there are numerous different techniques that can be used. In order to improve the performance of a parallel code, understanding of these techniques are essential.

3.1.1 Flynns Taxonomy

Flynns taxonomy is a classification of parallel computing architectures first proposed by Michael J Flynn in 1966. Flynns taxonomy defines four unambiguous terms to describe the relationship between data and the technique by which it is processed. The entirety of Flynns taxonomy is visualised in Figure 3.1, and the following bullet list details the architectures it describes.

Single Instruction Single Data (SISD) refers to the most basic type of processing; whereby a single instruction is applied to a single data item stored in memory. Code that uses this processing type is often referred to as scalar or serial.

Single Instruction Multiple Data (SIMD) allows for a single instruction to be applied to multiple data items stored in a contiguous piece of memory. To gain the largest performance benefit from SIMD operations, the multiple data items must be read using a single instruction, and then the same operation must be applied to all items. SIMD processing is often referred to as vectorisation, as the data is processed as a one-dimensional vector.

Multiple Instruction Single Data (MISD) is a rarely used processing technique that applies different operations on identical data. Rather than improving the performance of a program, it is often used for mission critical computations where there is no room for error.

Multiple Instruction Multiple Data (MIMD) is currently the most commonly used parallel processing technique. It describes a machine that contains many asynchronous processors that function independently, and as such, most modern processors can be categorised as MIMD machines.

Isca, like most scientific codes, uses the SIMD and MIMD approaches to parallelism. As discussed in Section **??**, the model uses the MPI library to split the domain into latitude bands.

3.1.2 Data-level parallelism (SIMD)

From the 1970's to the early 1990's, high performance machines relied heavily on instruction level vector operations to compute in parallel [14]. These machines used vector processors, and performed operations on one-dimensional arrays of data, rather than single data items using a

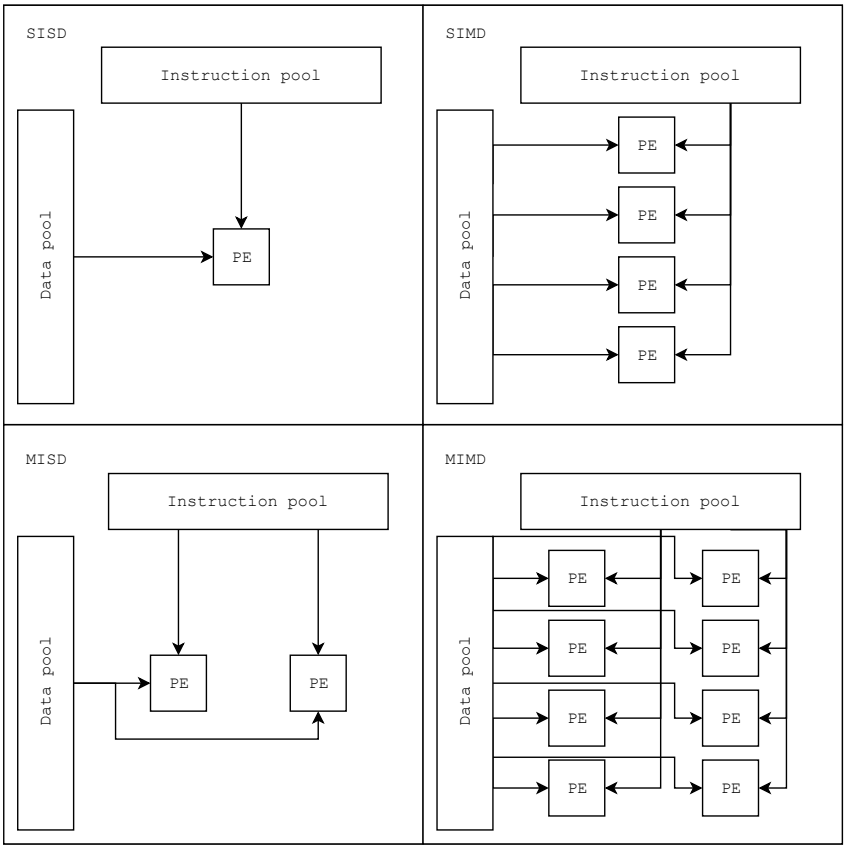


Figure 3.1: Flynns taxonomy; instructions are applied to data by various processing elements (PE) in different ways.

SIMD processor architecture [15]. Many scientific codes from this time were written with this architecture in mind, and it is likely that it influenced the design and implementation of the FMS.

Instruction level parallelism using SIMD is becoming popular once again through the introduction of Intel's Advanced Vector Extensions (AVX). Intel introduced AVX in 2011's Sandy Bridge architecture, AVX-2 in 2013's Haswell architecture and AVX-512 in 2016's Skylake architecture [16, 17]. AVX-2 increased the width of some vector registers to 256 bits, allowing for SIMD operations on four, 64-bit elements per clock cycle. In comparison, the AVX-512 instruction set increased vector register width to 512 bits, allowing for SIMD operations on eight, 64-bit elements per clock cycle, double that of AVX-2 [16, 18].

Although AVX-512 has a higher throughput of operations per clock-cycle, using wider vector registers results in greater power consumption, which in turn causes the processor to generate more heat. In order to maintain a suitable temperature, the processor will usually decrease its clock speed for the duration of the loop using the AVX-512 registers, often resulting in no overall performance gain over AVX-2.

3.1.3 Message passing

To enable communication between nodes, processors can explicitly communicate with each other using a message passing library. Most notably the MPI standard

Isca uses MPI as in 1998, compute nodes would only have a few processors per node.

3.1.4 Hardware

At the time the FMS was developed, many scientific codes were limited by computational power [19]. It is therefore likely that the FMS was optimised for the compute-bound systems of the time.

3.2 HPC clusters

Throughout the course of this research project, Isca has been ported to and run on four different high-performance supercomputers. This section discusses these machines, and the features of their respective processor architectures. A full breakdown of the most important hardware features can be found in Table 3.1. BCP3, BCP4 and BP are all based on the well-established line of x86-64 Intel Xeon processors and Isambard is based on the ARM-v8 Cavium ThunderX2 processor.

3.2.1 BlueCrystal phase 3 (BCP3)

BCP3 is primarily intended for smaller jobs that run on a single node, and it is the oldest cluster still in use at the University of Bristol. A single node of BCP3 contains two, eight-core Sandy Bridge Xeon E5-2670 v1 processors, which were the first line of the Intel processors to use AVX, which increased the width of vector registers to 256-bits.

Table 3.1: Hardware specifications of the target HPC systems.

Attribute	Intel Xeon (x86-64)			ARMv8
	BCP3	**BCP4**	**BP**	**Isambard**
Processor	E5-2670 v1	E5-2680 v4	Gold 5120	ThunderX2
Codename	Sandy Bridge	Broadwell	Skylake	ThunderX2
Instruction set	AVX	AVX-2	AVX-512	NEON
Clock Speed	2.6 GHz	2.4 GHz	2.2 GHz	2.1 GHz
Cores / Node	2×8	2×14	2×14	2×32
Memory / Node	64GB	128GB		256GB
Compute Cores	3,568	14,700	-	10,752
Interconnect	QDR InfiniBand	Omnipath	Ethernet	Cray Aries

3.2.2 BlueCrystal phase 4 (BCP4)

BCP4 has been the University of Bristols main workhorse cluster since 2017. It was designed and configured by OCF in collaboration with Lenovo and is primarily intended for large parallel jobs across multiple nodes. BCP4 now has an established user-base, however the machine is almost at maximum capacity and some longer jobs can spend over a week in the queue before they run.

A compute node of BCP4 contains two fourteen-core Broadwell Xeon E5-2680 v4 processors. They use the AVX2 instruction set architecture and were introduced by Intel in 2016,

3.2.3 BluePebble (BP)

BP is a new Intel-based cluster, managed by the Advanced Computing Research Centre (ACRC) at the University of Bristol. It was created in order to ease congestion on BCP4 by moving some of its heaviest users to their own cluster with dedicated resources. Some members of the meteorological research group at the University of Bristol can be classified as heavy users of BCP4, and have recently purchased a £10,000 dedicated node of BluePebble to conduct their research using Isca.

BP contains two different types of compute node, both using Intel's Skylake architecture. The first contains two twelve-core Xeon Gold 6126 processors and the second contains two, fourteen-core Xeon Gold 5120 processors. Both processors make use of AVX-512 instruction set.

3.2.4 Isambard

The GW4 Alliance, which consists of the Universities of Bath, Bristol, Cardiff and Exeter, together with the UK Met Office and Cray Inc have worked together to deliver the Isambard supercomputer, which is the result of a £3m award by the EPSRC. Isambard provides multiple advanced architectures, however the focus of this research project is the Arm-based Cavium ThunderX2 processor, which forms the basis of the machine. Each of Isambard's 168 compute nodes contain 64 ARMv8 cores in a dual-socket configuration [20].

Cavium ThunderX2 Server Microprocessors

Arm primarily manufactures processors for mobile devices, and has only recently produced hardware optimised for HPC systems [21]. Due to the heat generated by high clock rates, modern

chip designers are now limited by power consumption. Because of this constraint, the current trend in supercomputer design is to use large shared-memory nodes, that use higher core cores and decreased clock rates [22].

As Arm processors were originally designed for mobile devices, they have inherently low power consumption. Because of this, the European Mont-Blanc project begun to investigate the potential of the Arm architecture for HPC in 2011 [23]. This project proved to be successful, however the study uncovered some problems with the architecture that have since been addressed. ThunderX is a line of 64-bit many-core server microprocessors developed by Cavium as a result of over 8 years of work by the Mont-Blanc project and other contributors. The ThunderX2 was first released in early 2018 as the successor to the ThunderX, and is the first generation of Arm-based server microprocessors intended for high performance computing.

Initial studies have found that the ThunderX2 presents as a real alternative to current offerings by vendors of desktop hardware [24, 21], finding that the processor delivers competitive levels of performance to Intel's line of Xeon processors.

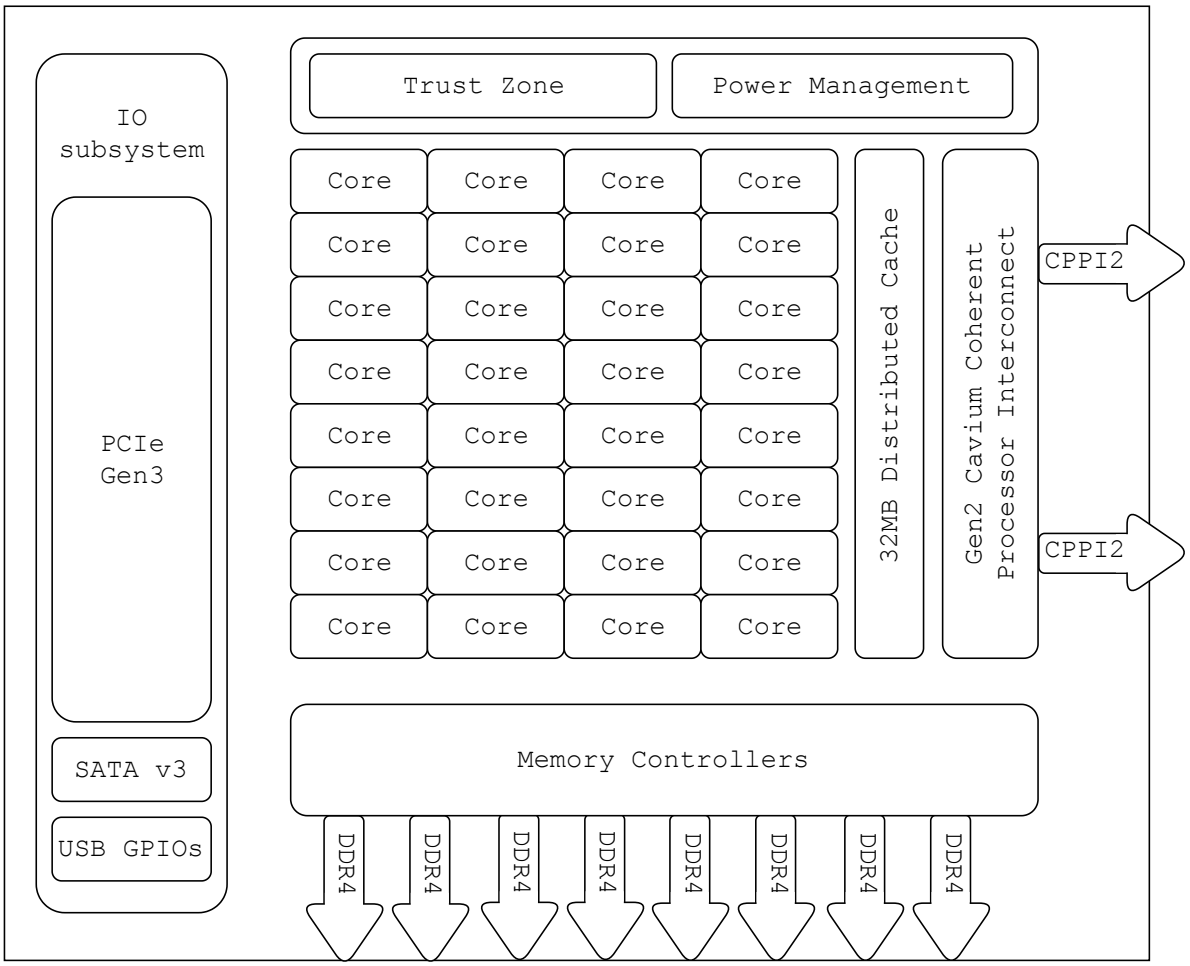


Figure 3.2: Block diagram of the Cavium ThunderX2 server microprocessor. Diagram redrawn from [20]

The ThunderX2 uses the ARMv8.1 instruction set, which allows for the use of 128-bit NEON SIMD vector registers. Perhaps the most interesting feature of the ThunderX2 as noted by McIntosh-Smith et al., is its eight memory controllers per socket, which have been demonstrated to produce a memory bandwidth in excess of 250GB/s [21]. The layout of the processor is shown in Figure 3.2.

A64FX

To meet the compute requirements of future HPC workloads, Fujitsu has recently announced the next generation of Arm chips in their A64FX processor. The A64FX improves upon the NEON instruction set found in the ThunderX2 by introducing Scalable Vector Extensions (SVE), which allow for a flexible vector register length between 128 and 512 bits so that vector length can reflect the compute requirements of different use cases [25, 26]. These processors have not yet been released, however this thesis provides an estimate of their performance based on the performance of the ThunderX2.

CHAPTER 4

Benchmarking and performance analysis

This chapter is an introduction to benchmarking both hardware and software, and describes some of the techniques and metrics used to benchmark the Isca code.

4.1 Cluster benchmarks

The STREAM TRIAD and High Performance Linpack (HPLinpack) benchmarks have been used to measure the peak memory bandwidth and floating point performance of each node configuration used in this study, respectively. This has been done to provide a relative performance overview of each processor architecture, and to highlight the differences between them.

4.1.1 STREAM TRIAD

The speed of processors has increased exponentially over the past twenty years, as described by Moore's law, which states that the number of transistors in a dense integrated circuit doubles approximately every two years [27]. However, the speed of memory has only marginally improved, as manufacturers have historically prioritised memory capacity over speed [19, 28]. The result of this is that many scientific codes are no longer bound by compute, but by the rate at which data can be read from, or stored to memory by the processor. The STREAM memory-bandwidth benchmark was introduced by John McCalpin in 1995 to address the limitations of the benchmarks of the time, and to measure processor performance by its peak memory bandwidth consumption, rather than Floating Point Operations (FLOPS).

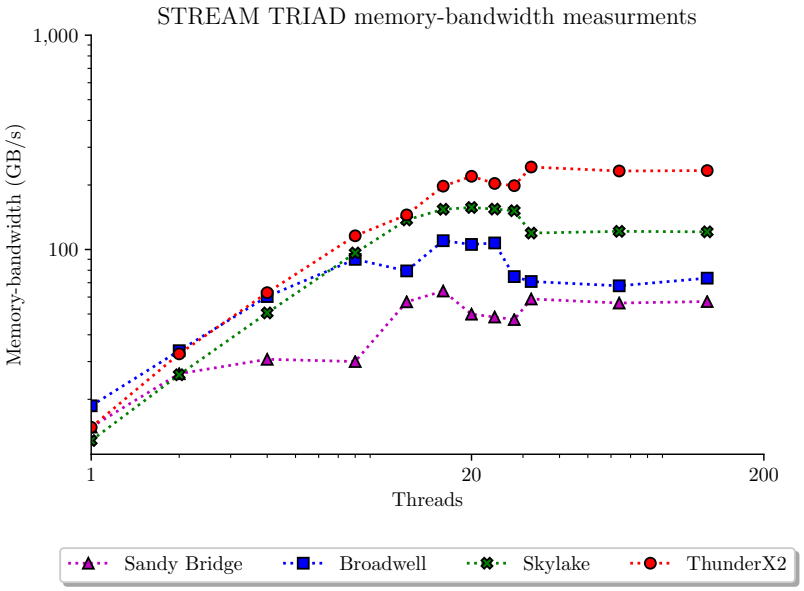


Figure 4.1: STREAM TRIAD results for all processor architectures that have been used as part of this research project.

As core counts and memory-channels continue to grow, it becomes increasingly difficult to measure the memory bandwidth of modern processors, and results can greatly vary depending on the system configuration used to compile and run the benchmark. The results shown in Figure 4.1 were collected using the original STREAM benchmark code [29]. The code was compiled using the Intel compiler with the same flags and environment variables on each cluster with the exception of the ThunderX2 processor, which used the GNU compiler.

The ThunderX2 processor has eight memory controllers per socket, and presents a peak STREAM TRIAD result in excess of 240 GB/s for a dual-socket configuration. In comparison, the Skylake processor provides a result of just 157 GB/s. This observation alone is a testament to the class leading memory bandwidth of the ThunderX2.

4.1.2 High Performance Linpack (HPLinpack)

The theoretical peak performance of a compute node can be calculated. Equation 4.1 is commonly used to find the peak performance of a processor where c denotes the processor speed in GHz, p denotes the number of processor cores, i denotes the number of instructions per clock cycle, and o denotes the number of processors per node.

$$GFLOPS = c \cdot p \cdot i \cdot o \tag{4.1}$$

Generally, the theoretical peak performance of a machine is unattainable. The HPLinpack benchmark aims to measure the attainable percentage of peak processor performance by solving a dense system of $n \times n$ linear equations [30]. Figure 4.2 shows the actual performance measured using the HPLinpack benchmark compared the the peak theoretical machine performance calculated using Equation 4.1.

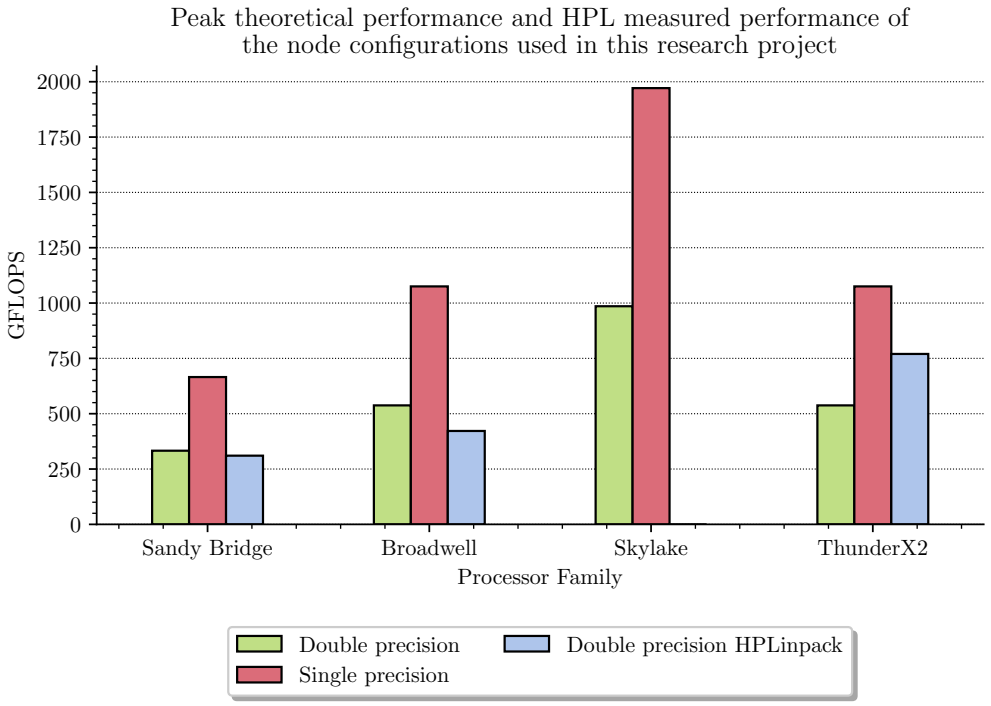


Figure 4.2: Comparison of the theoretical peak machine performance against the performance measured by HPLinpack

As can be expected, the performance measured using the HPLinpack benchmark is less than the theoretical performance in all cases. The large theoretical peak performance of the Skylake processor is attributed to its wide 512-bit vector registers, which can process 16 double precision floating point numbers with a single instruction. The drastically reduced actual performance measurement is because the processor can only use AVX-512 at a reduced clock rate, and therefore the performance model rarely opts to use these registers.

4.2 Application benchmarking

Floating point performance and memory bandwidth are usually measured using idealised techniques like the STREAM and HPLinpack benchmarks, and may not provide the best metrics for a complex code like Isca. Although the model is computationally demanding, there is also a large overhead cost incurred by communication. This includes tasks such as reading and writing files, the movement of data into contiguous memory and the transmission of data between processors. For this reason, the model can only be expected to achieve a fraction of the theoretical peak memory bandwidth and floating point performance. Therefore, the following performance metrics have been defined, and are used throughout this study.

4.2.1 Performance metrics

Wallclock runtime

Wallclock runtime refers to the total amount of real time that has passed from the start of the program to the end. In this study, wallclock runtime has been reported in seconds. Used alone, this metric does not provide a basis for comparison between other configurations.

Speedup

Speedup is a measure of relative performance between two solutions for the same problem. For the metric reported in this study, this is the number of times faster the code ran than some other given benchmark, typically the serial runtime. The speedup S of a code can be calculated given two runtimes R_1 and R_2 using the formula in Equation 4.2, whereby R_1 is $S\times$ faster than R_2.

$$S = \frac{R_1}{R_2} \tag{4.2}$$

Cost per gridpoint

Other studies that have benchmarked parallel climate codes have used the Computational Cost per Grid Point per Time Step (CCPG) as a primary performance metric as it takes into account the cost of interprocess communication [31]. When run on a single core, 100% of the program runtime is spent on computation. As the number of processor cores increases, a larger portion of the runtime is spent on communication, and therefore the cpu time taken to compute a single grid-point increases. The amount of consumed compute resources T_p for a given simulation can be calculated given the wallclock runtime t and the number of processors used p, as shown in Equation 4.3.

$$T_p = t \cdot p \tag{4.3}$$

To provide a meaningful comparison between core counts, the CCPG must be calculated. Given the number of timesteps N_t and number of grid points N_g, we can calculate the total simulation CCPG, C_{tg} as shown in Equation 4.4.

$$C_{tg} = \frac{T_p}{N_t \cdot N_g} \tag{4.4}$$

Although increasing MIMD parallelism by introducing additional processor cores decreases the overall runtime, a greater portion of the runtime is spend idle waiting for data to be sent between processors. The C_{tg} metric doesn't discriminate based on wallclock runtime, and provides a solid basis for comparison between model resolution and number of processor cores.

Operational intensity

The operational intensity I of a code or compute kernel is defined as the ratio of work W to the memory traffic Q. It is a commonly used metric to assist in the identification of performance bottlenecks of high-performance codes. Operational intensity is formally defined in Equation 4.5.

$$I = \frac{W}{Q} \tag{4.5}$$

For the analysis performed in this research project, W denotes the number of FLOPS, and Q denotes the total amount of memory transferred in Bytes. This results in operational intensity measured in FLOPS/Byte.

Summary

To ensure simplicity, the wallclock runtime is the primary performance metric used throughout this paper. However, it is important to consider other metrics as they can uncover important features of the code that are overlooked by runtime alone. Values for all four metrics defined in this section have been calculated using the data collected as part of this research project.

CHAPTER 5

Porting

In the context of software development, porting refers the process of modifying an existing codebase in order for it to run on a different system than it was originally written for. This chapter gives an overview of some of the tools and techniques used when porting Isca, and presents some of the challenges encountered in doing so. Many small changes have been made to the codebase to enable the model to compile and run using a selection of commonly-used compilers and these can be found in appendix A.

5.1 Compilers and MPI libraries

To allow for the best comparison between processors, Isca was compiled using a number of different compilers and MPI libraries. Table 5.1 shows the different configurations used to compile Isca on each of the clusters used in this study.

Table 5.1: Compilers and MPI libraries used for benchmarking

Cluster	Processor Family	Fortran Compiler	MPI library
BCP3	Sandy Bridge	GNU 7.1.0 Intel 13.0.1	OpenMPI OpenMPI
BCP4	Broadwell	GNU 7.2.0 Intel 18.0.3	OpenMPI Intel MPI
BP	Skylake	Intel 19.0.3	Intel MPI
Isambard	ThunderX2	CCE 8.7.9 GNU 8.2.0	Cray MPI Cray MPI

Isca uses a Perl script called MakeMakeFile (MKMF) to construct makefiles for different model configurations. Before compilation, Isca runs a series of 'template scripts' to export environment variables and to load relevant module files to be used by MKMF. In order to compile on a new system, a new template script must be written to setup the compilation environment for the machine in question. Although Isca provides some example scripts to do this on existing systems, a new script had to be written for each compiler and processor pair in this study. Two examples of such scripts can be found in Appendix C.

5.1.1 GNU Compiler Collection (GCC)

The GNU Compiler Collection (GCC) is a selection of compilers for various programming languages produced and maintained by the GNU project. GCC is available on many different computer architectures, providing the same interfaces and compile flags options on each. This makes it a convenient compiler for porting code, as similar configurations can be used on different machines. The Isca codebase already has an existing GCC template script, however it required some minor changes to allow for compilation on each system.

5.1.2 Intel Compiler Collection (ICC)

The Intel Compiler Collection (ICC) provides numerous premium compilers specifically for Intel-based machines, and is bundled as part of Intel Parallel Studio XE. As Intel develops its compilers alongside its hardware, ICC generally produces well-optimised instructions, however they are not portable and are limited to Intel processors only. Contained within ICC is the Intel MPI library, which is focussed on making parallel applications perform better on Intel-based clusters.

The Intel template script generally worked, however the locations of some libraries needed to be specified. Additionally, the NetCDF library in needed to be recompiled using the latest version of the Intel compiler in order to work.

5.1.3 Cray Compiling Environment (CCE)

The Cray Compiling Environment (CCE) is a Fortran 90 compiler developed by Cray Inc. This compiler is relatively new in comparison to the Intel and GNU compilers, and as such is strict to the Fortran standard. This caused many issues when porting the code using this compiler, and flagged up a number of issues with the Isca codebase. The process of porting for CCE turned into a stringent debugging exercise that has inevitably improved the reproducibility of the code on different platforms. The following subsections describe some of the code changes required to compile and run Isca using the CCE compiler, however a full list of code changes can be found in appendix A.

Implicit type conversion

To provide interprocess communication, Isca uses a *'Massively-parallel'* module, codenamed MPP. It is a set of simple calls to provide a uniform interface to a collection of commonly used message-passing routines for climate modelling, implemented in different libraries [5]. This module defines many subroutines that depend on the definition of the `MPP_DEFAULT_VALUE_` macro, which is defined using preprocessor directives at compile time. The `MPP_DEFAULT_VALUE_` can be assigned as either real, integer or logical. As an extension for backwards compatibility with other compilers, the GNU and Intel compilers allow for the implicit conversion of logicals to numericals and vice versa. When converting from a logical to an integer, the numeric value of `.false.` is 0, and that of `.true.` is 1. When converting from integer to logical, the value 0 is interpreted as `.false.` and any non-zero value is interpreted as `.true.`. This does not conform to the Fortran 90 standard, which disallows implicit conversion between numeric variables and logicals [32, 33]. This error was found throughout the codebase and changes were made to resolve this issue by enforcing the Fortran90 standards.

Namelist read errors

Isca uses Fortran namelist files to read large numbers of parameters into existing variables and data structures at runtime. Using CCE, many of the namelist files were being read incorrectly, as Isca did not open and close files between reads causing the code to hang during execution. Additionally, the Cray compiler requires that the representation of the value in the namelist file reflects the type of variable that it will be used for. This means that an integer value must be stored in the namelist as `variable = 1`, and not `variable = 1.0`.

Ambiguous arithmetic

The Cray compiler required brackets around some arithmetic where it could be considered ambiguous. This was only found in a few places in the codebase, and was trivial to fix.

5.1.4 Discussion

As the Fortran standard has evolved over the past twenty years, many features that were once commonplace are no longer deemed to fit the language standard, and are therefore not supported by newer compilers. Some of the more popular compilers like GCC and ICC have been updated to allow for backwards compatibility with legacy code, however this is not the case for the CCE compiler, which strictly follows the Fortran standard.

Both ICC and GCC overlook many negligent programming practices. However to remain portable, codes should be written to the standards of the programming language. Many legacy codes suffer from this issue whereby outdated code remains unchanged throughout many years of use, and part of this research project was to update this code to improve its portability.

5.2 Cluster feedback

5.2.1 Stack size

Prior to this project, the default stack size on BP was 8Kb. However, as Isca uses a large amount of memory, this caused a stack overflow error when running the model at resolutions greater than T42. Due to some configuration restraints on the cluster, the PBS scheduler does not allow for the stack size to be increased using `ulimit -s unlimited`, as used in Isca's run script. To resolve this issue, the default stack size was increased to 64MB by the system administrator.

As a temporary work around before this issue was resolved, and before interactive jobs were available on the cluster, a regular job can be submitted the queue that sleeps for an hour in the submission script. The details of the job can be found using the `qstat -f <jobid>` PBS command, which can then be used to SSH to the node running the sleeping job. As PBS has been not been used to get access to the node, the stack size can be increased using the command `ulimit -s unlimited`, and the code will then run as if in an interactive job. Most of the single-node runtimes for BluePebble were collected using this technique as it took a long time to modify the cluster configuration.

5.2.2 Strict processor enforcement

Isca's Python library is multithreaded, creating additional threads of execution when running experiments. This caused some issues when the model was run on BluePebble, as its job scheduler was configured to strictly enforce the CPU limitations defined in the job submission script. After running for an arbitrary amount of time, the job would fail as additional Python threads were created, causing the CPU to try to burst past the scheduler limit. This issue took many weeks to fix, and required working alongside the BluePebble system administrator.

5.3 Libraries and dependencies

Although many of the libraries required by Isca were already available as module files, in some cases they were not. This meant that they needed to be built in the `$HOME` directory to allow for the model to be compiled. These builds were then used to create module files by the system administrators. Throughout the course of the project, the NetCDF, Git, FFTW, Anaconda Python, zlib and HDF5 software packages were installed multiple times using different compilers.

5.4 Development tools

All code developments were made remotely over SSH, and all work was done on the filesystem of each supercomputer. Microsoft Visual Studio Code has a Remote-SSH plugin that allows for files to be edited as if they were on a local machine. This plugin is still in its Beta phase of development, and is therefore not reliable. When it failed, work continued using both emacs and vim. To allow for code changes to be synchronised across machines, a new fork of the Isca Git repository was created. Each cluster had their own development branch as well as a main development branch for merging changes between them.

5.5 Verification of results

When porting a codebase, it is important to test that that the changes made to the code are backwards compatible. This means that all changes must be non-intrusive, and configurations must default to the original behaviour. In the case of Isca, all code changes that have been made to run the model on a different cluster have been tested on the BCP4 supercomputer to ensure that they produce the same outputs.

5.5.1 Units of last place

To ensure that the model produces the same results on each system, the model outputs have been checked using the Units of Last Place (ULP) numerical analysis technique, which can be used to measure the spacing between floating point numbers. As the whole real line cannot be represented in memory, there is a minimum difference between two numbers occupying the full space offered by floating point numbers. NetCDF files aim to record data in a continuous fashion, which is an impossible task for the discrete numbers used in computational simulations like Isca.

The outputs of a simulation run by Isca can be verified by comparing the ULP of each parameter in the resulting NetCDF file. As the model is chaotic, even small changes to model parameters can produce vastly different results. However, as the model is not stochastic, the same simulation configuration ran on two separate machines should produce identical results, assuming that the compilers store variables to the same degree of precision.

Rounding errors

Rounding errors are a commonly occurring quantisation problem in scientific codes (citation needed). Although double-precision floating point variables can store numbers to a high degree of precision, they can only store a finite number of digits. Rounding errors are a result of performing arithmetic on continuous numbers in a discrete representation. The IEEE Standard for

Floating-Point Arithmetic (IEEE 754) states that all floating point arithmetic must be correctly rounded to within 0.5 ULP of the true mathematical result, therefore any differences greater that 1 ULP suggest inconsistencies in the code.

5.5.2 Program verifcation

A C++ program written by Gethin Williams in 2009, and modified for this research project was used to measure the difference in ULP between NetCDF files obtained on different machines. The program allows for a tolerance to be given, to accommodate any rounding errors that may have accumulated throughout the course of the simulation. Due to the differences in compiler and processor architecture, a tolerance of 2 ULP was deemed acceptable. All changes to the Isca codebase were verified using this metric.

Part II

Benchmarking, performance analysis and optimisation

CHAPTER 6

Experimental Methodology

This chapter outlines the experimental methodology used for benchmarking and analysing the performance of Isca. The collective aim of the following experiments is to characterise the code, to identify potential optimisations and to provide a comparison of the processors themselves.

Experiment A: Scaling study The wallclock runtime taken to complete a simulation for various spatial resolutions and core counts was measured.

Experiment B: Compiler comparison The per-node performance of the model was compared for two different compilers on each processor, excluding Skylake.

Experiment C: Vectorisation analysis The per-node performance of the model was measured with SIMD instructions enabled and disabled, to determine the importance of data-level parallelism.

Experiment D: Communication analysis The interprocess communication times were measured.

Experiment E: Runtime variation The wallclock runtime was measured for each of the iterations that comprise a full simulation.

Performance projection Using the results from these experiments, the performance of Isca has been projected to Fujitsu's A64FX Arm processor, which is planned to be debuted in the Post-K supercomputer in 2021. Most notably, the A64FX will be the first production processor to use the new SVE, which allow for 512-bit SIMD registers.

In order to collect reliable data, a full node was used for each experiment. Even when run in serial, the model used the resources of an entire node, so that the performance would not be affected by shared resource usage by other programs running on the cluster. To account for variations in runtime caused by factors outside the control of the experiment, all runtimes reported in this chapter are the mean value of three repeat measurements, unless stated otherwise. The results presented in the following section are the consequence of over 2,000 hours of experimental runtime.

6.1 Model configurations

Isca is a coupled model, allowing for the simulation of either the atmospheric or oceanic components of a planet, or both components simultaneously. The complexity of these simulations are defined at compile time, and allow for different algorithms to be applied depending on the model configuration. Although this means that the model is highly flexible, it introduces a challenge when trying to profile the code as a whole, as optimising one configuration may have no impact on another. This research project focuses on the optimisation of two test configurations: the well-known Held-Suarez configuration and a Grey-Mars radiation model.

6.1.1 Held-Suarez test case

The Held-Suarez simulation was designed by Held and Suarez in 1994 [34] to allow for comparison between GCMs. It is well studied, and is considered to be the gold standard for benchmarking climate modelling code (citation needed). It is configured to simulate only the 'dynamical core' of a planet, which contains the discretised equations of motion and state. In terms of complexity, the Held-Suarez model is one of the simplest configurations available, and is essentially the foundations upon which more complex models are built. The simulation maintains a constant climate throughout its duration by forcing many parameters to predefined values. This allows for the dynamical core to be run by itself, without the need for coupling with other complex model components. This makes the Held-Suarez a good candidate for benchmarking and optimisation as the dynamical core code is used in all other model configurations that model the atmosphere.

Isca's Held-Suarez simulation computes over an idealised model of the Earth. In terms of measuring its performance, the simulation has been run for 12 model months with each month simplified to last 30 days, for a total of 360 model days per simulation. This length of time was chosen to allow for the performance to be measured at each phase of the Earth's orbit of the Sun. The Held-Suarez simulation does not use solar radiation as a model parameter. However, it is important to measure the performance for a full year to model other seasonal parameters.

6.1.2 Grey-Mars test case

The Grey-Mars simulation is configured to simulate the effect of grey radiation on the planet Mars over time, building upon the dynamical core code used in the Held-Suarez configuration. It was chosen for optimisation due to its frequent use by academics at the University of Bristol, as well as its demonstration of some of the more complex features of the model.

The axes of both Earth and Mars are not orthogonal to their orbit of the sun. Earth's axis is at a 23.5° tilt, and Mars' axis is at 25° (ref). These tilted axes are responsible for the seasons, however this causes many climate models to suffer from a load-balancing issue whereby calculations take longer on the side of the planet facing the sun due to increased levels of thermal radiation (ref). To test for this, the Grey-Mars configuration has been run for 690 model days to account for the 687 martian days it takes for Mars to orbit the Sun. This simulation is broken into 23 sub-simulations, each lasting 30 days.

6.1.3 Domain decomposition

When running in parallel, Isca requires that the number of latitudes divided by the number of cores must be divisible by 2 [13]. Therefore the T21 resolution, which splits the planetary domain into 32 latitudes, can be run on 1, 2, 4, 8 or 16 cores. The simulation cannot be run on more processor cores, as the number of processors will be equal to the number of latitude bands. A full list of compatible resolutions and core counts can be found in Table 6.1.

This inherent domain decomposition constraint imposed by the model means that some nodes are unable to run Isca at full capacity. For example, a single node configuration of BCP4 can only run Isca on 16 out of 28 cores per node. This poses an interesting problem, whereby the model is a better fit for some nodes than others, simply due of the number of processor cores per node.

Although Isca can vary both in its spatial and temporal resolution, the scaling study undertaken as part of this research project focuses solely on variations in spatial resolution. This decision was made in order to simplify the process of performance modelling by limiting the number of problem sizes. Additionally, changes to performance as a result of time stepping are generally predictable,

Table 6.1: Resolutions and their compatible core counts. Lower resolutions are limited in the number of cores they can use.

Truncation	Latitudes×longitudes	Available core count
T21	32×64	1, 2, 4, 8, 16
T42	64×128	1, 2, 4, 8, 16, 32
T85	128×256	1, 2, 4, 8, 16, 32, 64
T170	256×512	1, 2, 4, 8, 16, 32, 64, 128

and will not contribute to a further understanding of the code. A model that performs twice as many time steps will perform twice as many calculations and will therefore be twice as slow.

6.2 Automated data collection

In order to collect reliable and consistent data, it is important to use the same method of data collection for different configurations. When benchmarking high performance applications, data collection can be a laborious process. The runtimes of the configurations used in this study are in the range of 3 minutes for simple configurations at low resolutions, up to 10 days for high resolution complex scenarios running in serial. Running this range of simulations manually would be incredibly time consuming, therefore a Python library was written to automate this process. The source code for this library can be found on GitHub [35].

The Python library was written to sequentially run a number of different experimental configurations given a set of parameters, including the core count, resolution and model configuration. This allowed for a number of experiments to be run from within a single job submission script, with the results of each experiment automatically stored in a spreadsheet. Each experiment defined by the Python script recorded the total time taken to complete the simulation, as well as each thirty-day time step.

6.2.1 Job submission

BCP3 uses the Portable Batch System (PBS) job scheduler, BP and Isambard use the PBS Pro job scheduler and BCP4 uses the SLURM scheduler. These are tools that allow for applications to be submitted to a queue and run on a compute node when the required resources are available. Each of these schedulers use a slightly different syntax, therefore a number of submission scripts have been created for each cluster based on the amount of resources required and expected runtime. Example job submission scripts can be found in Appendix D.

As the clusters used in this project are actively used for research, there is naturally some competition for compute resources between users. A trial and error approach was used to find the right parameters for the job script in order for the job to be processed from the queue quickly, whilst ensuring that the runtime was adequate to complete the entirety of the job.

6.3 Experiments

6.3.1 Experiment A: Scaling study

To determine how well the model performs when presented with additional compute resources, Isca was run on 1 core, up to and including the maximum number of cores available on a node of each cluster. In addition to this, it was also run across all possible combinations of model configuration and resolution in order to measure its performance at various levels of complexity and realism.

Whilst the majority of the results observed in this experiment were as expected, the performance is significantly affected when running across multiple nodes. For this reason, the results have been presented as a comparison of the performance on a single node, and a further analysis has been done for the multi-node configurations.

6.3.2 Experiment B: Compiler comparison

To find the optimal compilation settings for each processor, both the Held-Suarez and Grey-Mars model configurations were complied using two different compilers on each cluster. All cases compiled using the GNU compiler used the same flags, and all cases compiled using the Intel compiler used the same flags. The flags used for the GNU compiler on Isambard were those recommended by the ARM64 Best Practices Guide, and the equivalent flags were used on the Intel machines [36]. At the time of writing, only the Intel compiler and MPI library was available on BluePebble, therefore there is no comparison of different compilers on a Skylake node.

Table 6.2: Number of processor cores used to measure the performance of different compilers at the T21 and T42 resolutions.

Processor Family	**Number of cores**	
	T21	**T42**
Sandy Bridge	16	16
Broadwell	16	16
ThunderX2	16	32

For this test, the per-node performance was considered only. The model was run on up to the maximum number of cores available on each node for the given model configuration as shown in Table 6.2. This provided a comparison of the compilers in relation to the performance available on other processors. The observations made in this experiment informed the choice of compiler for all other experiments. For all results reported on Intel nodes, Isca was compiled using ICC. For all ThunderX2 results Isca was compiled using GCC.

6.3.3 Experiment C: Vectorisation analysis

The per-node performance of the code was measured with SIMD instructions enabled and disabled for the Held-Suarez and Grey-Mars model configurations running at T42 resolution. This determined the importance of data-level parallelism in the model. Isca was compiled using the relevant flags to disable vectorisation, and vector reports were produced to ensure that the no automatic vectorisation occurred. For this experiment, all other optimisations were enabled.

6.3.4 Experiment D: Communication analysis

This experiment measured the percentage of runtime spent in the MPI library and the total communication time between processors. This was done to determine if there was load imbalance affecting the performance of the model, and to find the percentage of total runtime spent in communication.

6.3.5 Experiment E: Runtime variation

Isca simulations are made up of many sub-simulations called epochs, with each epoch often lasting for one model month at a time. Epochs differ from time steps, which describe the amount of time between each global calculation. To determine if any parts of the simulation are more computationally demanding than others, the runtime of each epoch was measured for both the Held-Suarez and Grey-Mars configurations at various model resolutions.

6.3.6 Experiment F: Roofline analysis

A roofline model is an insightful visual performance analysis technique used to identify the hardware-limiting factors of an application, or compute kernels within an application. It plots the floating point performance as a function of peak machine performance, peak machine memory-bandwidth and the operational intensity of the code itself. The performance limiting factor of a code or compute kernel can be determined by looking at where it sits on the roofline.

CHAPTER 7

Results

This chapter presents the findings of the experiments described in Section 6.3, demonstrating an extensive performance analysis of Isca running two unique model configurations on four different compute nodes. These results have been used to inform the design and implementation of two performance optimisations in Chapter 8.

7.1 Experiment A: Scaling study

Figures 7.2, 7.3 and 7.5 show how the runtime of the model varies as a function of processor cores. The vertical coloured bars on the y plane indicate the maximum number of processors available on each cluster.

7.1.1 Results

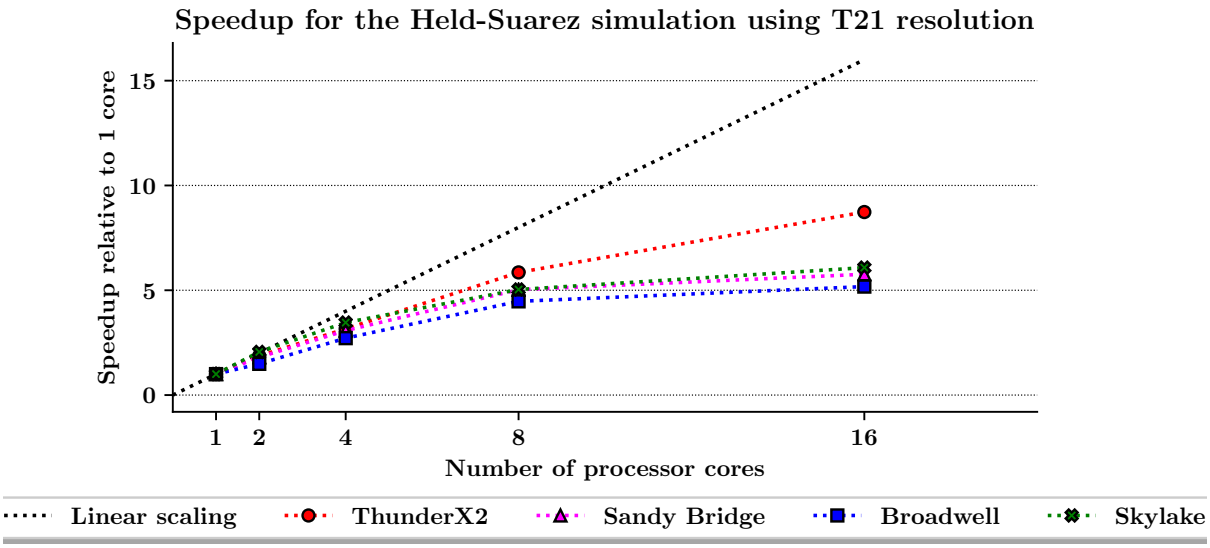


Figure 7.1: Parallel speedup relative to serial performance for the Held-Suarez configuration running at T21 resolution.

When Isca is run on 8 cores, the Sandy Bridge, Broadwell and Skylake processors see a performance improvement of 5.1×, 4.5× and 5.0× relative to the serial runtime, respectively, before plateauing between 8 and 16 processor cores (Figure 7.1). The ThunderX2 speeds up by 5.9× when run on 8 cores, and 8.7× when run on 16 cores. When increasing the number of processor cores from 8 to 16 for the Held-Suarez configuration, there is only an approximate 1.15× performance gain for the Intel processors. This is typical of many parallel codes, whereby the performance benefit of additional compute resources decreases as more processor cores are utilised (citation). As this problem size is small, more time is spent in communication relative to compute when a greater number of processor cores are used.

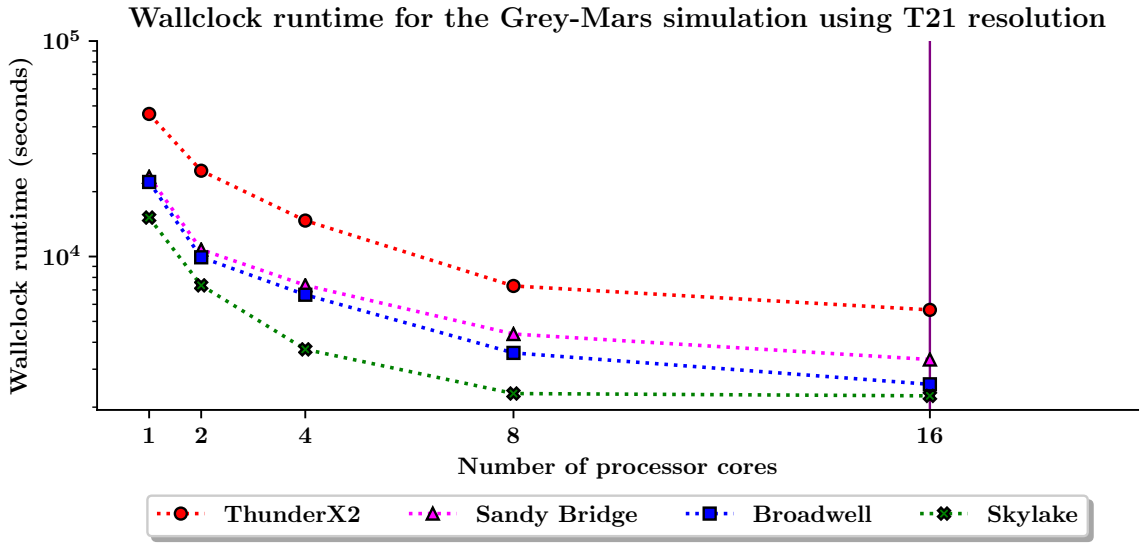


Figure 7.2: Wallclock runtime of the Grey-Mars configuration running at T21 resolution across all processor architectures.

The scaling curve in Figure 7.2 shows a sublinear plateau for all node configurations, whereby the wallclock runtime of all four processors reduces steadily from 1 to 8 processor cores. The wallclock runtime is greatest when the program is run in serial and lowest when run on 16 cores, which is the maximum number of cores possible for this resolution. The trend observed for the Grey-Mars configuration (Figure 7.2) is comparable to that of the Held-Suarez result, whereby the slowest runtime is the serial case and the fastest is the 16 core case (Figure 7.1). The Skylake processor massively outperforms all other processors when run on 8 cores, however it quickly tapers off when run on 16 cores, running just 1.02× faster than the 8 core case.

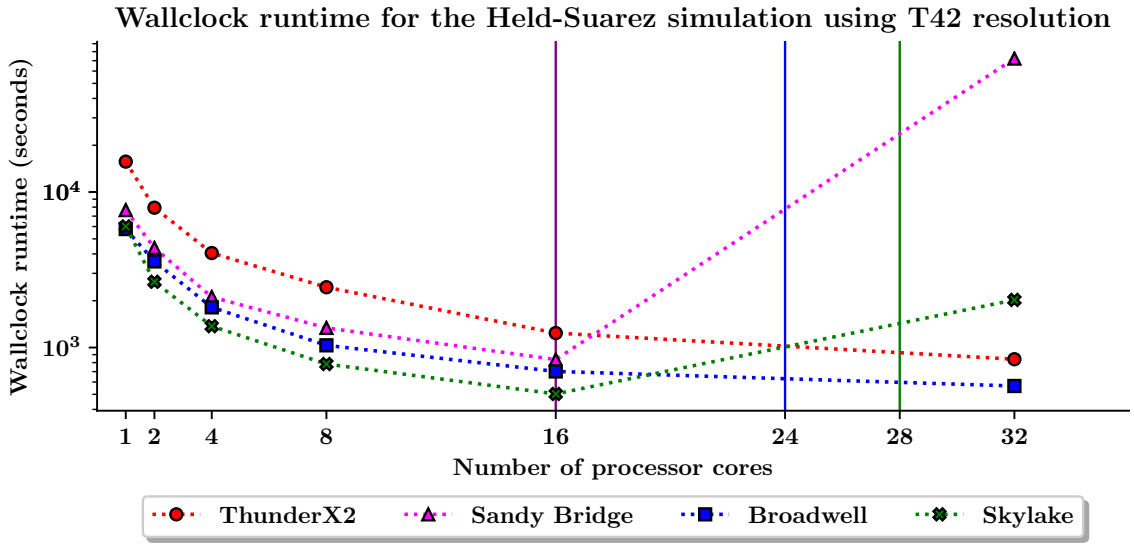


Figure 7.3: Wallclock runtime of the Held-Suarez configuration running at T42 resolution across all processor architectures.

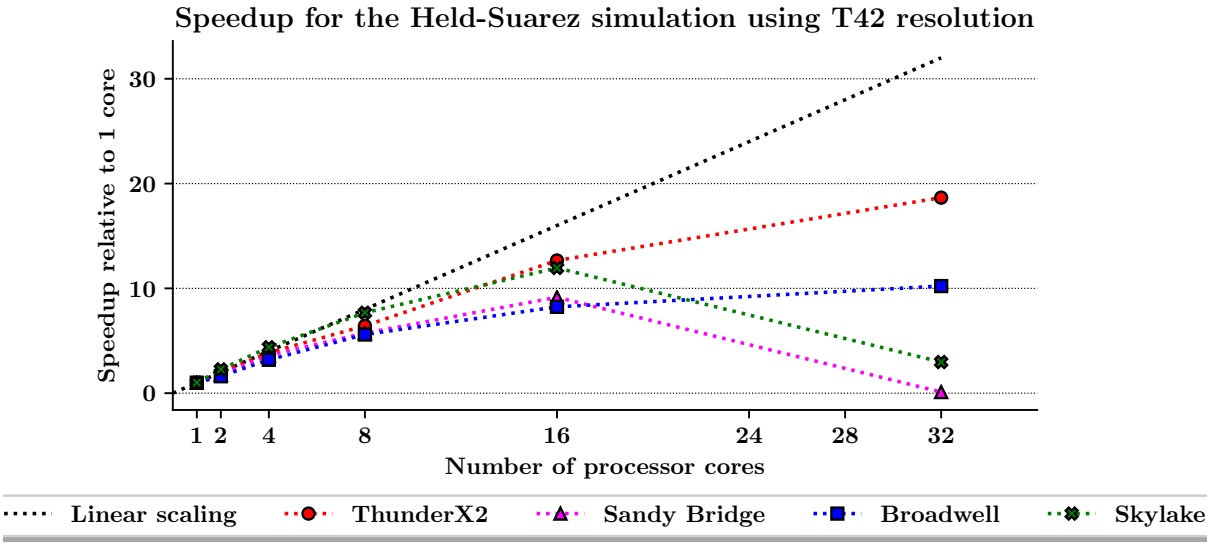


Figure 7.4: Speedup of the Held-Suarez configuration running at T42 resolution relative to serial runtime across all processor architectures.

Increasing the spatial resolution to T42 (Figures 7.3, 7.4) presents a similar scaling curve to that observed for the T21 resolution (Figure 7.2). For all processors except Sandy Bridge, the slowest runtime is measured for the serial code, and the performance improves until the program is run on 16 cores. For the Intel processors, running on more than 16 cores requires multiple nodes, which has a dramatic impact on the performance of the Sandy Bridge and Skylake processors. At 32 cores, the performance of the Sandy Bridge and Skylake processors reduces from 9.2× to 0.1× and 12.0× to 3.0×, respectively. The Broadwell and ThunderX2 processors improve at this core count, from 8.2× to 10.2× and 12.7× to 18.7×, respectively. Although the ThunderX2 does not provide the fastest overall runtime (Figure 7.3), plotting the speedup relative to the serial runtime demonstrates that it exhibits the best level of scaling (Figure 7.4). This can be attributed to the large core count of the processor, which means that it does not have to rely on internode communication.

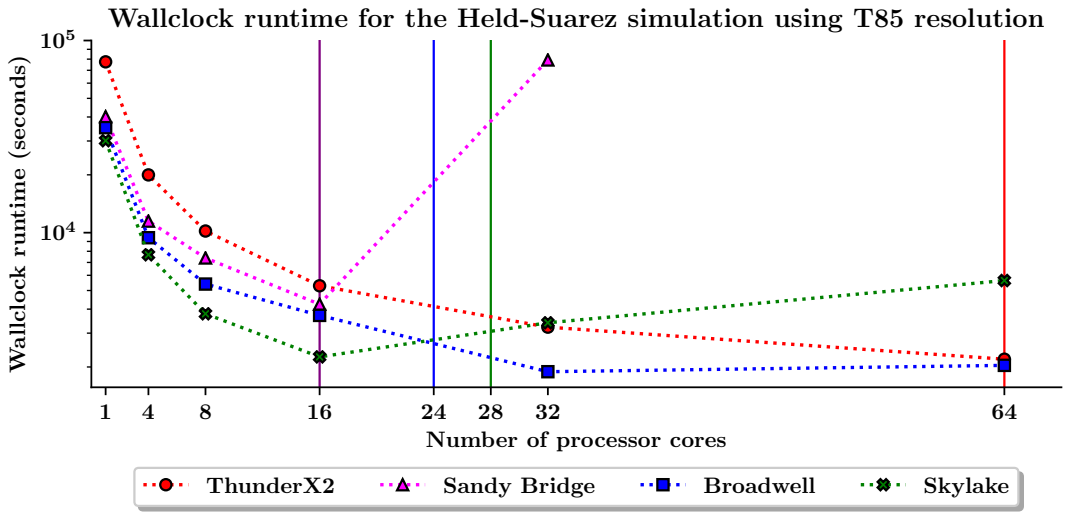


Figure 7.5: Wallclock runtime of the Held-Suarez configuration running at T85 resolution across all processor architectures.

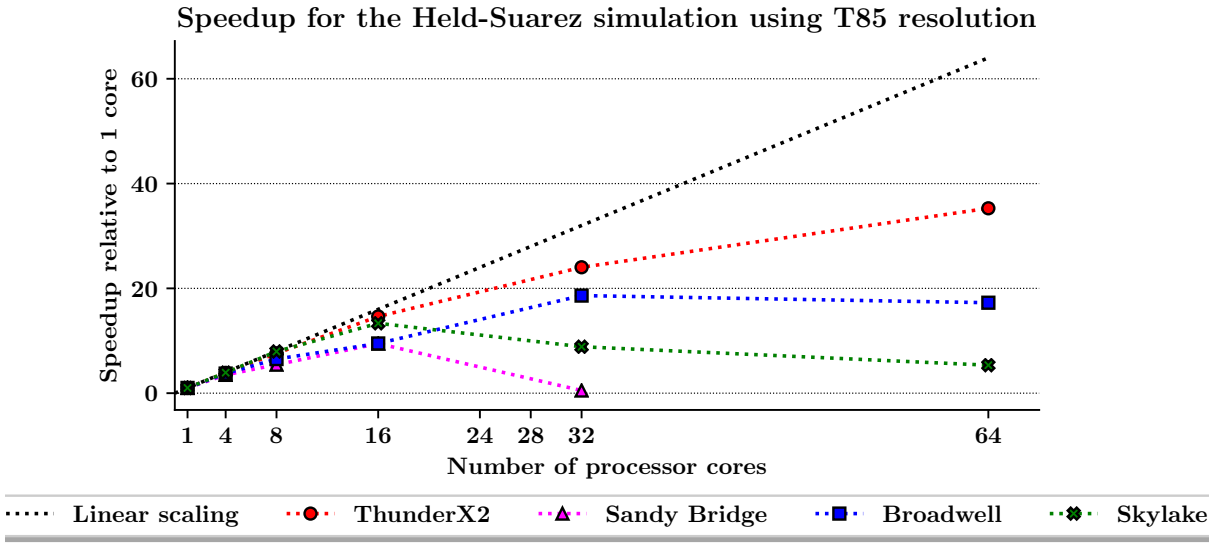


Figure 7.6: Wallclock runtime of the Held-Suarez configuration running at T85 resolution across all processor architectures.

For the T85 resolution (Figures 7.5 and 7.6), the performance of the Broadwell processor begins to decline when run on 64 cores across three nodes, reducing from a 18.6× speedup at 32 cores to a 17.3× speedup at 64 cores (Figures and 7.5). This suggests that this is the point at which the cost of communication outweighs the benefit of additional parallelism for this processor. The performance of the ThunderX2 continues to improve when run on 64 cores, increasing from 24.0× at 32 cores to 35.3× at 64 cores.

7.1.2 Conclusions and discussion

This scaling study highlighted some of the architectural differences between Intel and Arm processors. Arms approach to processor design relies on a greater number of simple processor cores, in comparison to Intel's fewer more complex cores. This design complemented the the Isca code by keeping communication inside a single node for resolutions of T85 and below, allowing for competitive runtimes to that of the Intel processors.

MPI is primarily used for communication between nodes, however messages are exchanged using shared memory when used within a single node (citation). This can greatly reduce the amount of time spent on communication as there is no data transfer between nodes, which can be an expensive process. This is likely what is causing Isca to scale well on the ThunderX2. In contrast, memory-bandwidth bound codes can see significant gains to performance when run on multiple nodes, which increases the total memory bandwidth and therefore reduces the cache-contention between processors (citation).

The performance of the Intel nodes suffered when ran on multiple nodes. This is to be expected on the Skylake nodes, as the BP supercomputer does not have a high-speed interconnect, using only ethernet to connect compute nodes. Interestingly, the lowest performance observed for the dual-node configuration is the Sandy Bridge processor, which takes 82× longer to complete the same job than when running on a single node. BCP3 uses QDR InfiniBand, which has a theoretical throughput of 8GB/s per connection and should therefore not exhibit this behaviour (citation). The Sandy Bridge node was the only Intel configuration to use OpenMPI instead of Intel MPI. Unlike Intel MPI, OpenMPI has not been developed specifically for Intel compute nodes, and this

could have affected the internode performance for this configuration. Both the ThunderX2 and Broadwell processors do not exhibit this behaviour when run on 32 and 64 cores, which suggests that the issue is most likely caused by internode communication rather than a bug in the model that manifests at higher core counts.

Due to restrictions imposed by domain decomposition, the Grey-Mars configuration cannot be run at resolutions higher than T42. Because of this, the results for the T85 resolution are limited to the Held-Suarez configuration only. Additionally, the time limit imposed by the queuing system prevented results from being gathered for the Sandy Bridge processor when running on 64 cores as the runtime was greater than 360 hours, which caused PBS to cancel the job.

7.2 Experiment B: Compiler comparison

This experiment measured the wallclock runtime of Isca when complied using the ICC, GCC and CCE compilers using two different model configurations and resolutions (Tables 5.1, 6.2).

7.2.1 Results

For all configurations tested on Intel nodes, ICC outperformed GCC (Figure 7.7). ICC provides a mean performance improvement of 1.26× over GCC across both the T21 and T42 resolutions. On the ThunderX2, GCC was 1.4× faster than CCE at the T21 resolution, but just 1.01× faster at the T42 resolution.

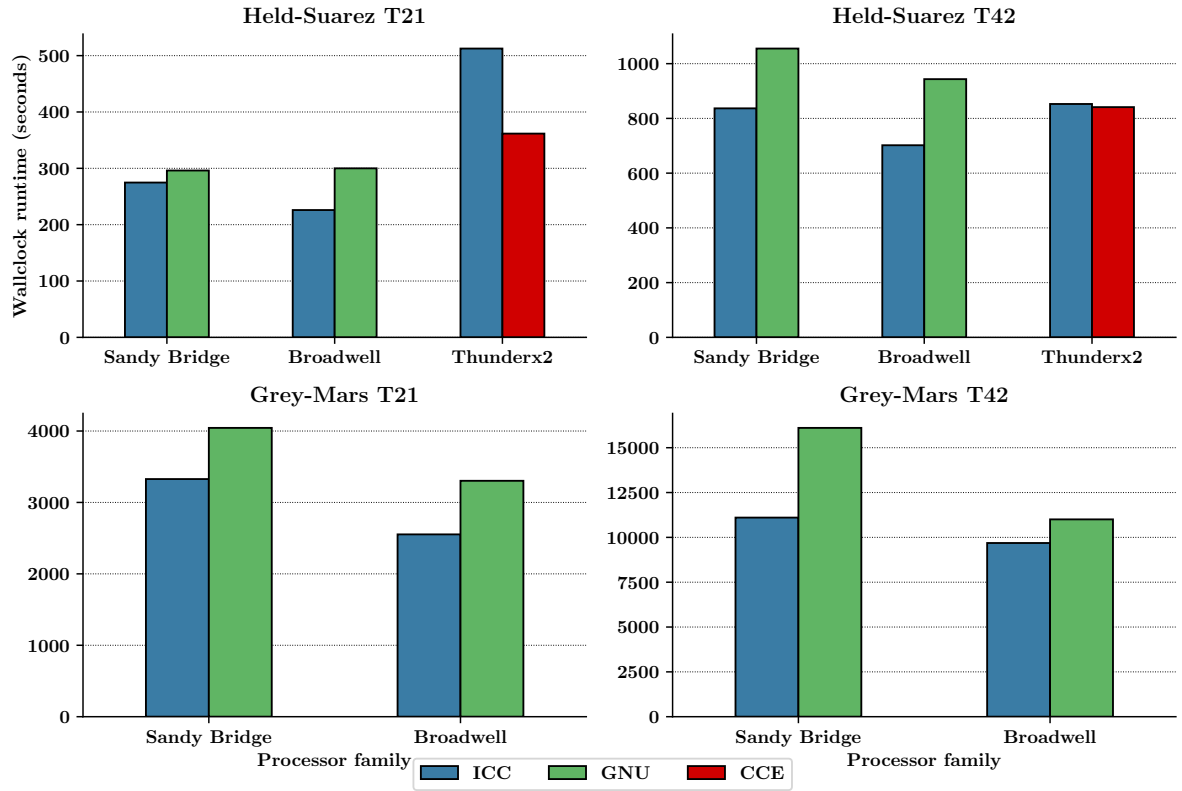


Figure 7.7: Runtimes for the Held-Suarez and Grey-Mars configurations using different compilers.

7.2.2 Conclusions and discussion

It can be expected that ICC outperforms GCC on Intel hardware, as Intel develops their compilers alongside their processors. Consequently, the Intel compilers produce well-optimised instructions for the architecture. This experiment confirmed this expectation as ICC outperformed GCC in all cases. For the ThunderX2, GCC outperformed CCE. For both compilers, flags were specified to produce hardware specific instructions for the ThunderX2 processor, however neither compiler has been exclusively developed for the Armv8 architecture. Therefore, this performance difference could be explained by the the maturity of GCC in comparison to CCE, as the compiler has been in constant development since 1987. However there is no evidence to support this claim.

7.3 Experiment C: Vectorisation analysis

This experiment measured the performance speedup of Isca when the model was compiled to use SIMD instructions, compared to when run in scalar.

7.3.1 Results

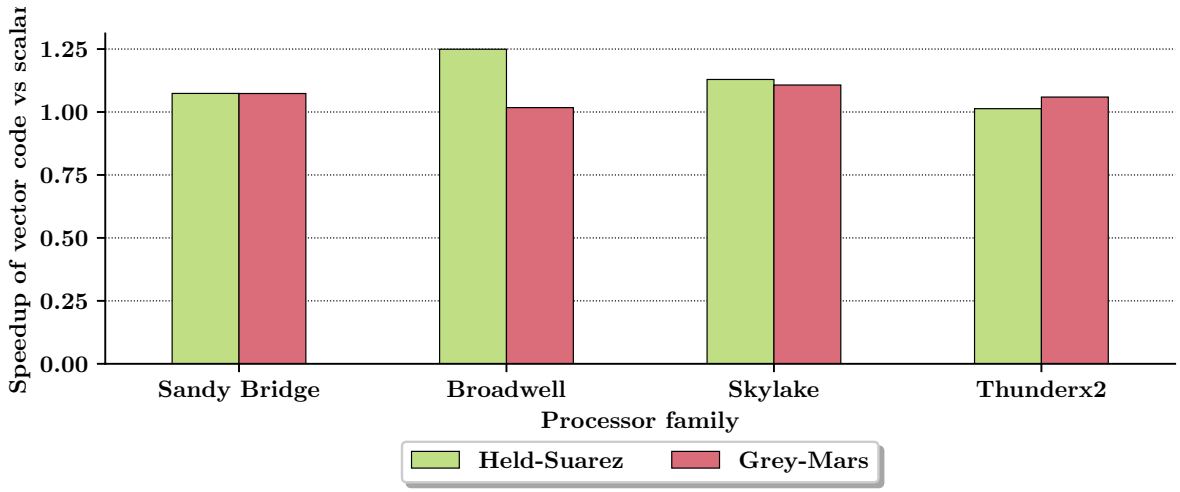


Figure 7.8: Speedup for the vectorised and scalar code.

Only marginal performance gains are observed when running Isca using SIMD instructions compared to scalar (Figure 7.8). For the Held-Suarez model, there is a 1.07×, 1.25×, 1.13× and 1.01× performance improvement for the Sandy Bridge, Broadwell, Skylake and ThunderX2 processors, respectively. For the Grey-Mars configuration, there is a 1.07×, 1.02×, 1.11× and 1.06× performance improvement for the same processors.

7.3.2 Conclusions and discussion

Some of the most time consuming compute kernels in Isca operate over arrays of double precision complex numbers. In Fortran, a complex number is composed of a pair of floating point numbers, representing both real and imaginary parts of the complex number. This means that a double precision complex number of kind 8 uses 128 bits of memory; both real and imaginary parts of the number are a real value of 8 bytes each. This means that vectorisation is costly in parts

of the code that iterate over complex data structures, and impossible on the 128-bit registers used in Intels SSE, and Arms NEON SIMD extensions. Interestingly, Intel's cost model often determines that there is no benefit to using SIMD instructions on arrays of double precision complex numbers, even when 256-bit and 512-bit registers are available as a results of AVX-2 and AVX-512 respectively.

Only small, but consistent performance gains were measured when the model was compiled to use SIMD instructions. This suggests that the model is not dependant on vectorisation to provide its performance. For the Held-Suarez configuration, the ThunderX2 saw the least improvement, which was expected as the ThunderX2 only has 128-bit vector registers. Following the ThunderX2 was the Sandy Bridge processor, which uses the AVX instruction set. Although this allows for a vector width of up to 256-bits, it does not include fuded multiply-accumulate operations like the AVX-2 instructions used by Broadwell and the AVX-512 instructions used by Skylake (citation).

7.4 Experiment D: Computation rate

Section 4.2 defines CCPG as the cost of computing a single grid cell per time step of the simulation. This experiment measures the cost of computation relative to the number of concurrently utilised processors.

7.4.1 Results

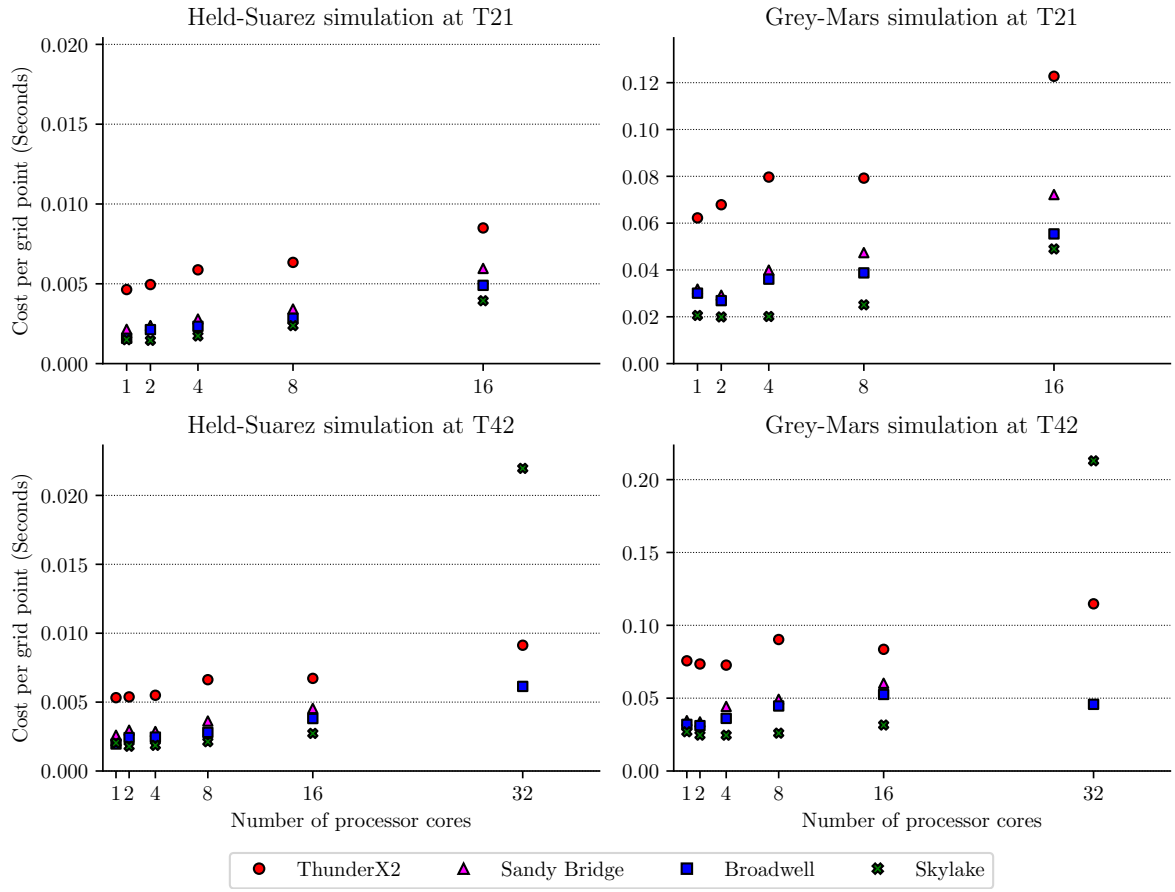


Figure 7.9: Cost per grid point for the Held-Suarez and Grey-Mars configurations at T21 and T42 resolutions.

For the Held-Suarez configuration running at the T21 resolution, the introduction of parallelism causes the CCPG to increase for all node configurations except Skylake (Figure 7.9). The CCPG decreased for the Skylake node when parallelism was introduced to the model. Increasing the amount of parallelism past the utilisation of two cores causes the CCPG to increase almost linearly for all processors at this resolution. The same trend was observed for the Held-Suarez configuration when running at the T42 resolution. However, for the dual-node case the CCPG sharply increases for the Skylake node, with a cost 6.7× greater than when run on the maximum capacity of a single node.

For Intel nodes running the Grey-Mars configuration at the T21 resolution, the CPPG initially decreases when parallelism is introduced. However running on more than 4 processor cores increases the CCPG. This is not the case for the ThunderX2, for which the CCPG increases until it is ran on 8 processor cores at which a plateau is reached, before increasing again at 32 cores. This same pattern is observed Grey-Mars configuration running at the T42 resolution, however the CCPG dips for the ThunderX2 when run on 16 cores, rather than reaching a plateau.

7.4.2 Conclusions and discussion

Plotting the CCPG demonstrates that increasing the number of processors also increases the cost to compute a single grid point (Figure 7.9). Although there are more processors collectively working on the problem, a greater portion of the runtime is spent on communication and therefore processors spend more time idle. Although increasing the amount of parallelism causes the the overall wallclock runtime to decrease, the total CPU time increases, which indicates that the code is less efficient.

7.5 Experiment E: Time spent in the MPI library

This experiment measured the percentage of runtime spent in the MPI library and the communication time between processors.

7.5.1 Results

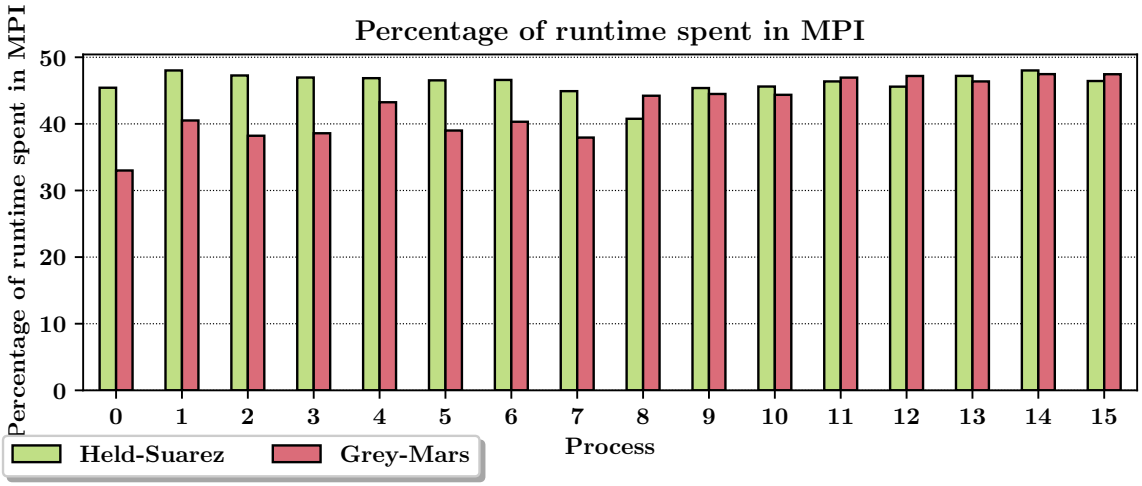


Figure 7.10: Percentage of runtime spent in MPI across processes for both the Held-Suarez and Grey-Mars model configurations.

For the Held-Suarez configuration, each process spends between 40.8% and 48.1% of its total runtime in calls to MPI, and no discernible patterns can be identified through visual inspection of Figure 7.10. For the Grey-Mars configuration, each process spends between 33.0% and 47.5% of its total runtime in calls to MPI. Processes 0-7 spend a much smaller percentage of their runtime in calls to MPI in comparison to processes 8-15, and process 0 spends a significantly smaller percentage of runtime in calls to MPI than any other process. Additionally, the time spent in the MPI library increases relative to the processor rank number.

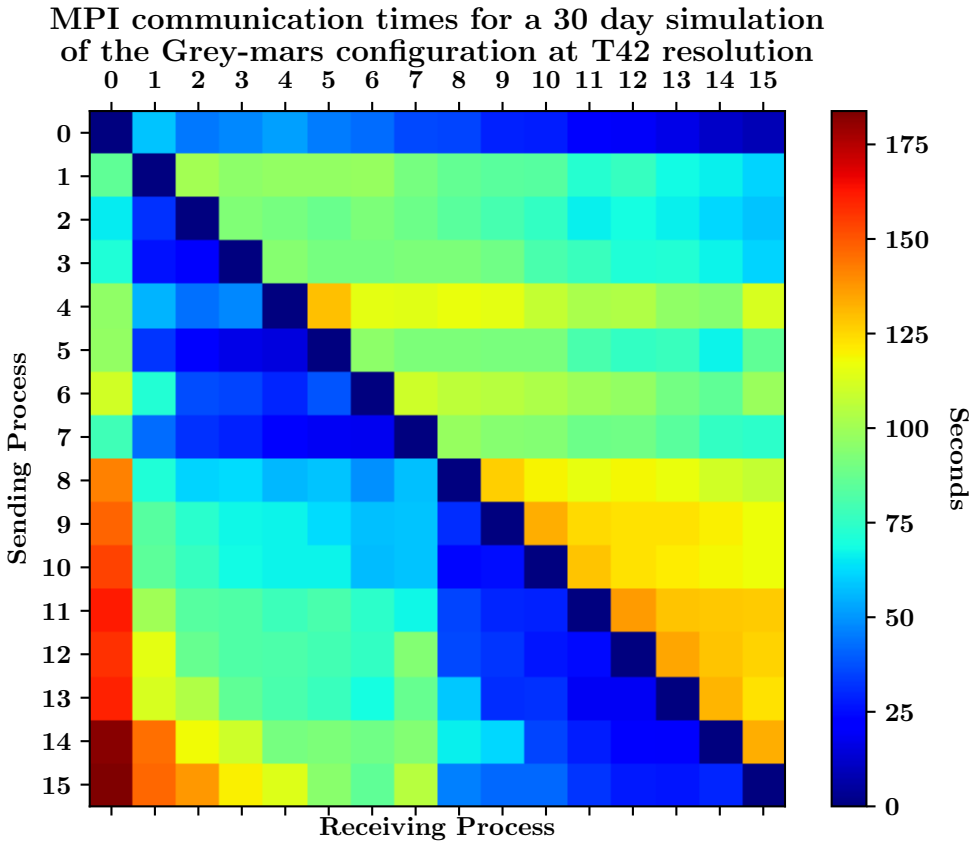


Figure 7.11: Communication matrix for the Grey-Mars model configuration when run at a resolution of T42. Communication time has been measured as the sum of all time spent inside the MPI library

The longest communication times are measured when processes 8-15 transmit data to process 0 (Figure 7.11), with a mean communication time of 118.5 seconds. However, process 0 has the lowest sending times of any process, with a mean communication time of just 31.3 seconds. This suggests that process 0 is consistently taking the longest amount of time to reach points of synchronisation, and can exchange data almost immediately upon reaching MPI barriers.

7.5.2 Conclusions and discussion

Load imbalance refers to an uneven distribution of work across compute resources. In the domain of HPC, it affects the performance of parallel codes only. The large variation in MPI time observed in Isca suggests that it suffers from a significant load balancing issue, which is amplified by the large portion of runtime that the model spends inside calls to MPI.

Many of the subroutines found in Isca with long execution times inherently involve expensive

communication. For example, when calculating global sums over 2D fields, each processor gathers the missing pieces of data they require to construct the global field from all other processes using blocking communication [31]. As Isca operates on a global domain, these points of synchronisation are unavoidable as communication must finish between all processes before the program can continue. Subroutines like this are found all throughout the Isca code, and each contributes to the large amount of time the model spends on communication. Unfortunately, Isca has been built around these points of synchronisation, and optimisation is not possible by using asynchronous message passing without affecting the scientific validity of the model.

Referring back to Figure 7.10, the Held-Suarez configuration generally presents more consistency amongst its processors than the Grey-Mars configuration, which is to be expected as it is much simpler simulation. Although some variation was measured in the time spent in MPI across processes for the Held-Suarez model configuration, this can be attributed to environmental variables outside the control of the experiment. The Grey-Mars simulation exhibits more severe symptoms of load imbalance. As discussed in Section x, some radiation models can suffer from a load-balancing issue whereby calculations take longer on the side of the planet facing the Sun. Although Isca

Check if this is true by running with data from run13.

7.6 Experiment E: Runtime variation

This experiment measured the differences in runtimes for each individual epoch comprising an Isca simulation.

7.6.1 Results

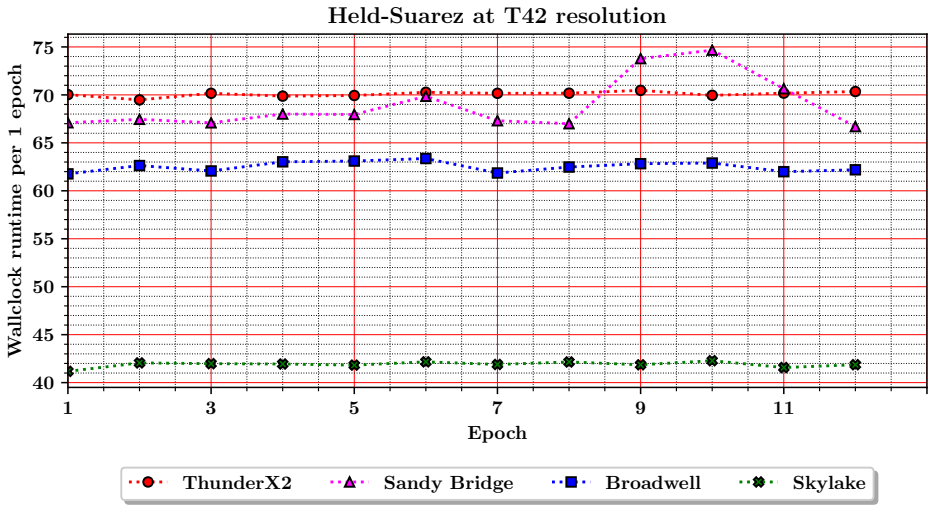


Figure 7.12: Variation in runtimes for the Held-Suarez configuration running at T42 resolution.

For the Held-Suarez model configuration, the runtimes for individual epochs have a range of 0.99 seconds, 1.11 seconds and 1.60 seconds for the ThunderX2, Skylake and Broadwell processors, respectively (Figure 7.12). However, the Sandy Bridge processor had more fluctuations in runtime throughout the full simulation, with a range of 7.97 seconds.

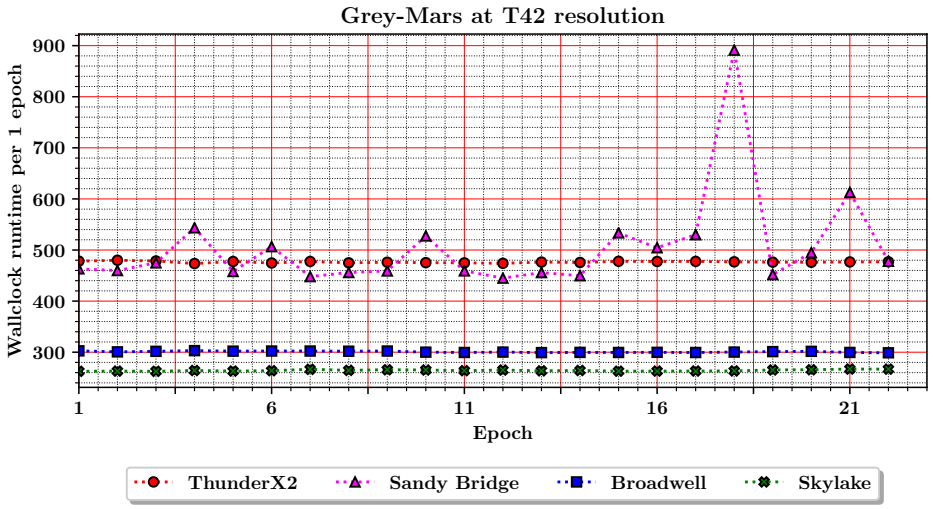


Figure 7.13: Variation in runtimes for the Grey-Mars configuration running at T42 resolution.

The Grey-Mars simulation presents similar results, showing only minor variations in epoch runtime across all processors except Sandy Bridge (Figure 7.13). There is a large peak for the Sandy Bridge processor at epoch 18, however rerunning this experiment causes peaks at seemingly random intervals.

7.6.2 Conclusions and discussion

The FMS Manual warns of a spin-up time that can slow the performance of the the first epoch of a simulation due to the initialisation of the global starting state [5]. This effect has not been observed in either the Held-Suarez or Grey-Mars model configurations, even in simulations that have not been visualised in Figures 7.12 and 7.13. The FMS manual was written in 2002, when the clock speeds of high performance processors were in the range of 1.1GHz to 1.5 GHz, and memory-bandwidth rarely exceeded 1600MB/s [37, 38]. Perhaps the spin-up cost is now negligible when the model is run on modern hardware.

The runtimes observed on the ThunderX2, Broadwell and Skylake processors indicate that each epoch contains the same amount of work and no part of the planetary orbit is more computationally expensive than others. The results obtained on the Sandy Bridge processor can be attributed to environmental factors, and can therefore be treated as an outlier.

7.7 Experiment F: Roofline model analysis

The Held-Suarez simulation running at the T42 resolution delivers an operational intensity of 0.11 FLOPS/Byte and a double-precision floating point performance of 1.54 GFLOPS (Figure 7.14). This indicates that the configuration is limited by memory-bandwidth, as the program total performance is located underneath the DRAM ceiling. The Grey-Mars configuration presents a floating point performance of 1.96 GFLOPS, with the same operational intensity of the Held-Suarez configuration. Intel Advisor suggests that the peak double-precision floating point performance of a code running on the Skylake architecture using SIMD is 584.99 GFLOPS. This means that the Isca code is only running at 0.26% of the peak performance available on the hardware.

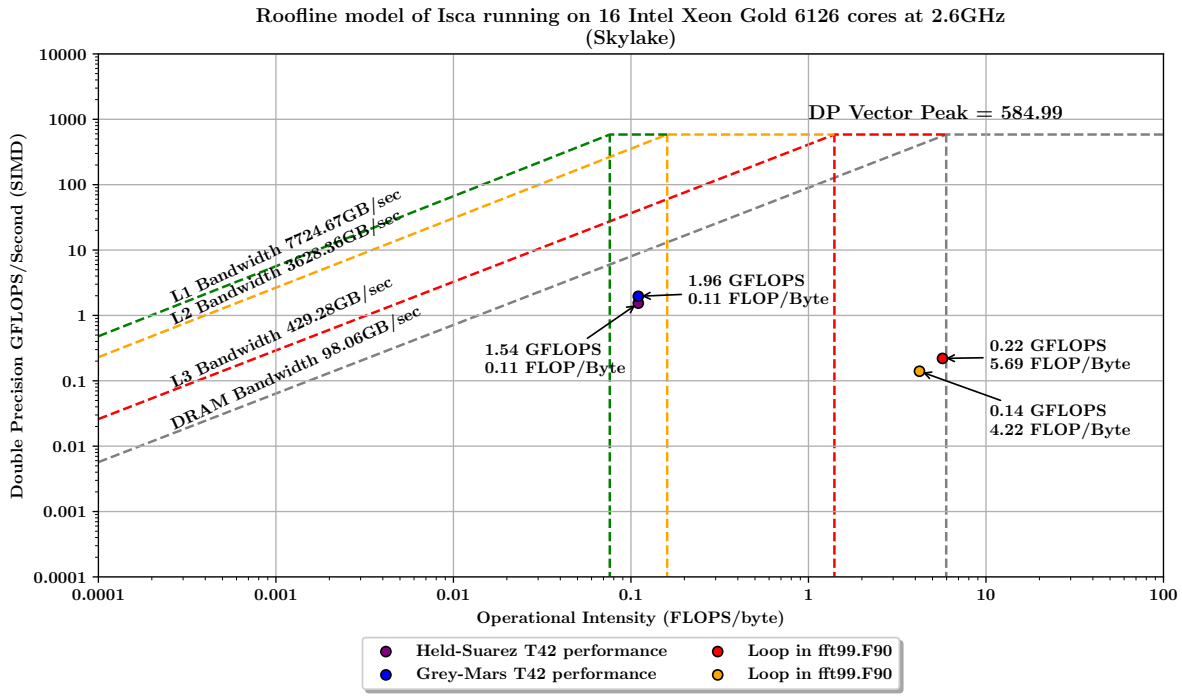


Figure 7.14: Roofline model for 16 cores of two Intel Xeon Gold processors at 2.6 GHz.

7.7.1 Conclusions and discussion

In addition to quantifying the performance of Isca, Figure 7.14 identifies 2 compute kernels referred to as Loop a and Loop b, for optimisation. Loops a and b are both found in Isca's FFT code, and provide especially bad floating-point performance of 0.22 GFLOPS and 0.14 GFLOPS, respectively. Further analysis of these compute kernels are found in Section **??**

7.8 Summary

The main observations of all experiments are summarised:

Scalability For the T21 and T42 resolutions, the Skylake processor presents the best level of single-node performance. However, the large core count of the ThunderX2 allows for the best level of single node performance at the T85 resolution.

Load balancing A serious load balancing issue has been discovered, whereby approximately 40% of the program runtime is spent inside MPI. Although communication is the biggest performance limiting factor of the code, the identified points of synchronisation are unavoidable, and cannot be optimised without rewriting the entirety of the model, which is well outside the scope of this project.

Vectorisation As Isca operates on double precision floating point values, the small vector registers of the ThunderX2 cannot be used on complex numbers. This also affects the performance of the Intel processors, as vectorisation only accounts for a performance gain of between 1.02× and 1.25× the un-vectorised code.

Compilers On the Intel machines, ICC outperforms GCC in all cases, however on the ThunderX2 GCC provides better performance than CCE. These results were used to inform the choice of compiler for all other experiments.

Slow FFT

CHAPTER 8

Optimisation

This chapter outlines the various performance bottlenecks identified in the Isca codebase, describes the steps taken to address their underlying issues, and evaluates the performance of the code changes that have been made. In some cases, a significant performance improvement was observed.

8.1 Fast Fourier Transform optimisation

As discussed in Section 2.1.2, spectral climate models use a FFT to transform data between the spacial and frequency domains. Isca does this using the `fft991` subroutine, found in the `fft99.F90` module. This subroutine is used to perform multiple one-dimensional FFT's in succession over a two-dimensional array of sequential data when converting from a grid-point decomposition to the frequency domain and vice versa. This implementation of the FFT has been adapted from a Fortran77 code written by Clive Temperton in 1981, and can be found in the EMOSLIB library by The European Centre for Medium-Range Weather Forecasts (references).

Although the `fft991` subroutine includes preprocessor directives to ignore vector dependencies at the most time-consuming loops, neither the Cray, GNU or Intel compilers will perform automatic vectorisation. This results in four loops in the `fft99.F90` module being run as scalar code, even when vectorisation is possible. Intel Advisor indicates that this is caused by a fixed-width iteration through multiple data structures using a non-contiguous stride.

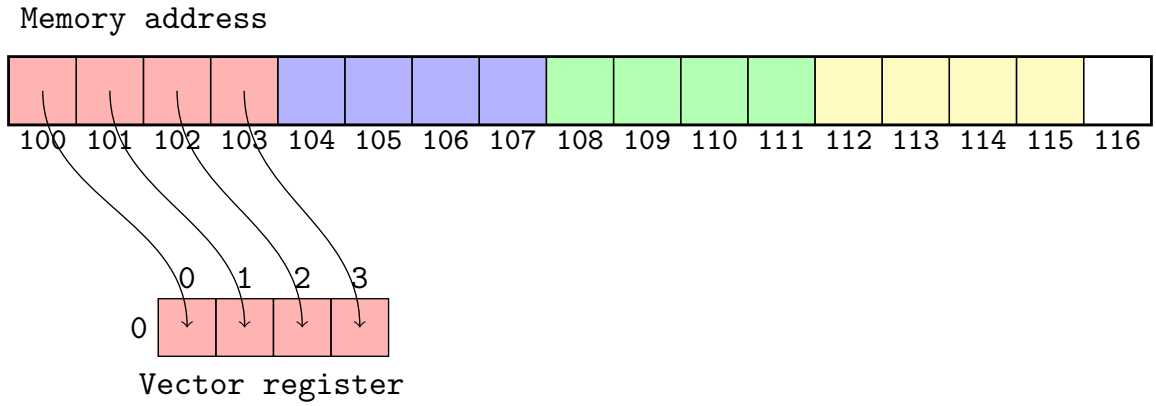
In the case of the 256-bit wide vector registers found in BCP4's Broadwell processors, eight consecutive floats, or four consecutive doubles may be loaded from memory with a single AVX-2 instruction. However, if the memory locations are not adjacent, then they must be loaded using multiple instructions, negating the benefit of using vector registers. This is illustrated in Figure 8.1.

When forcing the compiler to use vector instructions by using the `!DIR$ VECTOR ALWAYS` preprocessor directive, there is only a marginal improvement over the scalar code, as shown in Table 8.1. This provides evidence to support that the code has not been written to vectorise on modern hardware. Also, was compiled using xHost. Perhaps in the 80's the cost of the non-contiguous memory access was not as high? look into this...

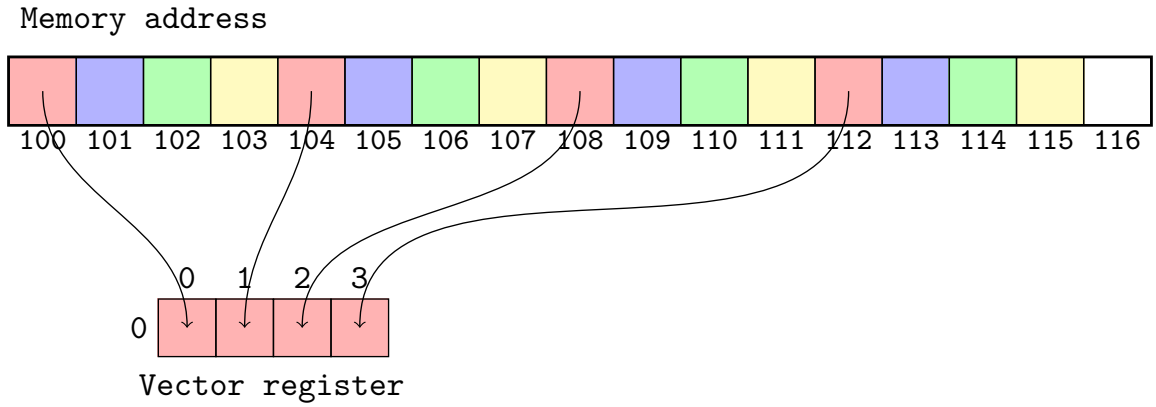
Table 8.1: Runtime spent inside two compute kernels for both scalar and vector code

Loop	Scalar time (milliseconds)	Vector time (milliseconds)
(a) `vpassm:1081`	822.5	649.8
(b) `vpassm:1049`	793.6	459.8

One of the biggest problems with Temperton's FFT is that it performs the transformation in-place. Although this reduces memory consumption, it introduces additional algorithm complexity as the results of intermediate calculations are not written to a temporary array. This may have been important in the late 80's and early 90's when memory was in short supply, however modern



(a)



(b)

Figure 8.1: (a) demonstrates that four contiguous single-precision floating point numbers can be read from memory with a single AVX-2 instruction, whilst (b) shows how four separate loads would be required for the same operation for non-contiguous data with a stride of 4.

processors often have in excess of 18 MB of on-chip cache and the total amount of memory usage is rarely an issue.

Although it may possible to rewrite Temperton's FFT to better make use of vector registers, this would be a massively time-consuming task, and does not guarantee a performance improvement. Therefore, modifications to the codebase have been made to allow for the use of the FFTW library instead.

8.1.1 FFTW

FFTW is an implementation of a DFT that aims to adapt to the hardware on which it is run [39]. The library has been written in ANSI C, however it provides interfaces for other programming languages including Fortran. Rather than providing a hand-tuned implementation for all possible hardware configurations, FFTW uses a plan to precompute various sub-arrays based on the shape, size and memory layout of the input data, without requiring the data itself [39]. The planning process yields a plan, which is an executable data structure that returns the DFT of the given input data.

To create a plan optimised for the hardware on which the code is compiled, the planner measures the runtime of many different plan configurations, returning the plan which results in the quickest runtime [39]. The planning process is computationally expensive, however it is only performed once, and the resulting plan can then be reused on different input data of the same dimensions. If many FFT's of the same type are repeatedly called in an application, this generally provides a net performance gain.

Plans are created using FFTW's own compiler called `genFFT`. Whilst the FFTW library itself is written in ANSI C, `genFFT` is written in Objective Caml, and is used to produce a number of small hard-coded transforms called codelets. Codelets are well-optimised simple straight line programs, which compute the DFT of a small sequence of data. The speed of FFTW is largely accredited to these codelets, which are sucessively applied to sections of a larger sequence.

Although not a requirement of using FFTW, the input data should be contiguous in memory so that vector instructions can be exploited. FFTW Version 3.3.8 officially supports AVX x86 extensions and Version 3.3.1 introduced support for the ARM Neon extensions [40]. Version 3.3.8 of the library was chosen so that it targets the vector extensions on all hardware configurations used in this research project.

Cooley-Tukey algorithm

Despite FFTW using many different FFT algorithms, the most commonly used is the Cooley-Tukey algorithm. This algorithm was popularised in 1965, however variations of the algorithm have been known as early as 1805 [41, 42]. Proper implementation of the Cooley-Tukey algorithm results in a time complexity of $O(n \log n)$. The algorithm is based on the assumption that a DFT of size $n = n_1 \cdot n_2$ can be expressed as a two-dimensional DFT of size $n_1 \times n_2$. The algorithm itself can be broken into three steps:

1. Perform n_1 DFTs of size n_2;
2. Multiply by some *twiddle factors*, which are a constant complex coefficient that is multiplied by the input data in order to recursively combine small DFTs;
3. Perform n_2 DFTs of size n_1.

When presented with this information, it becomes clear why the authors of FFTW decided to use a codelet-based design. An optimal solution to performing the FFT using the Cooley-Tukey algorithm allows for a codelet to calculate the DFT on a number of data structures of either size n_1 or n_2.

The difference between the forward and backwards transform is the sign of the exponent. `http://www.fftw.org/fftw2_doc/fftw_3.html`

$$Y_i = \sum_{j=0}^{N-1} X_j e^{-2\pi i j \sqrt{-1}/N} \tag{8.1}$$

$$Y_i = \sum_{j=0}^{N-1} X_j e^{2\pi i j \sqrt{-1}/N} \tag{8.2}$$

Implementation

In order to call FFTW rather than Temperton's FFT, a new Fortran module `fftw.F90` has been written, and preprocessor directives have been added to the existing `fft.F90` file to allow for the type of FFT used to be chosen at compile time. Compiling the model with the `-DFFTW3` preprocessor directive will compile the model to use FFTW instead of the default call to Temperton's FFT. Isambard, BCP3, BCP4 and BP have module files that allow for automatic linking to the FFTW library. If using the library on other systems, the FFTW library must be installed and linked to Isca manually. FFTW provides both a single and double precision version of its library, with subtle differences to the names of the interfaces it provides. Preprocessor directives must be used to chose which version of FFTW is linked to the Isca code.

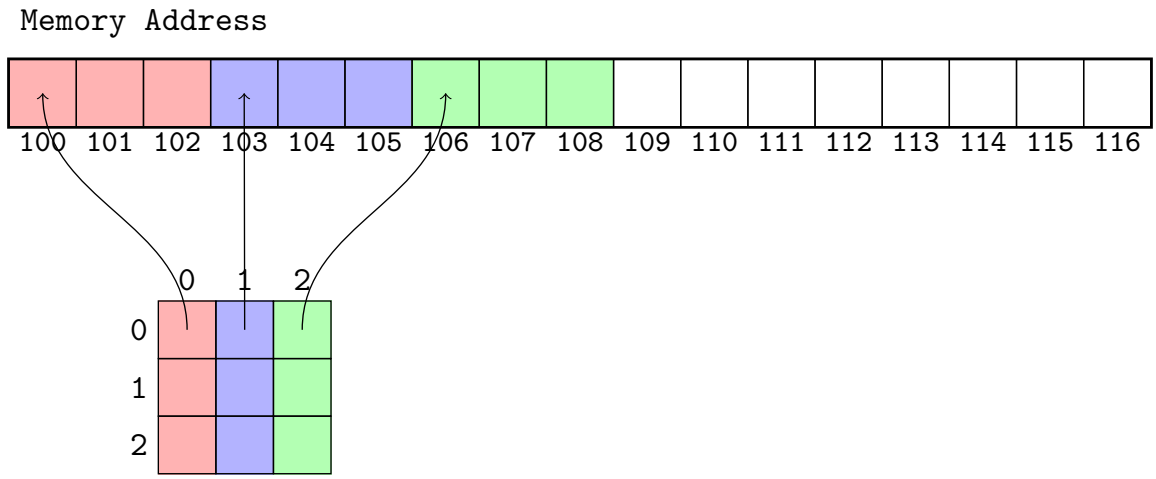


Figure 8.2: Contiguous data layout in memory for a two-dimensional array in Fortran.

To guarantee proper alignment for SIMD, the data structures on which the FFT is applied are allocated using the `fftw_malloc` subroutine, and deallocated using the `fftw_free` subroutine, both of which are provided by the FFTW library. These subroutines have the same behaviour as the `allocate` and `deallocate` subroutines found in the Fortran standard library, however they also call `memalign` to ensure that data structures are properly aligned. Figure 8.2 shows contiguous aligned memory for a two-dimensional array in the Fortran programming language, which uses a row-major order for multidimensional array storage.

Isca performs a transform from real to complex numbers and vice versa. A one-dimensional transform from a real array of size N results in a complex array of size $N/2$. When implementing this using FFTW, the same input and output arrays are re-used for multiple transforms in order to take advantage of FFTWs plans. As FFTW computes an unnormalised DFT, the result is multiplied by the number of items in the input sequence. This means that the result must be scaled by a factor of $\frac{1}{N}$ after the DFT is performed, which adds a small overhead to compute costs in addition to the cost of the DFT.

Verification

To ensure that FFTW computes the same values as Temperton's FFT, both forward and backwards transforms were tested on sequences of known data. The results of this test shows that both transforms compute the exact same DFT, and IDFT for 30 unique sequences of data. Additionally, computing both the DFT and IDFT of a sequence in succession yields the original sequence upon which the transforms were applied.

Methodology

To compare the performance of FFTW against the original Temperton FFT found in Isca, the time taken to complete a number of one-dimensional FFTs was measured for both FFT implementations. Each FFT implementation was tested on four different sizes of randomly initialised two-dimensional data structures. For example, a two-dimensional array of size 128 × 64 will compute the DFT of 128 one-dimensional arrays of size 64. The sizes of the arrays tested correspond to the different array sizes used for the T42 (128 × 64), T85 (256 × 128) and T170 (512 × 256) model resolutions. To emulate the Isca code, the test program was compiled using the same compiler flags used to compile Isca. To negate the error incurred by fluctuations in compute costs caused by the random data used in each array, the mean value for 100 transformations was calculated. 8.4.

Results and discussion

The biggest performance improvement was observed on the Sandy Bridge processor when performing a FFT on a grid size representing the T170 resolution. For this configuration, the FFTW code ran 5.25× quicker than Temperton's FFT. Figure 8.3 shows that the performance gain was relative to the size of the data structure used, almost linearly in the case of the Intel processors.

Interestingly, the performance gain was only significant on the ThunderX2 for a problem of size 256×128 and greater.

Regardless of wether FFTW provides a significant improvement to Its better to use a library that is updated rather than relying on bespoke code that does not evolve with hardware.

Need to bear in mind that this test was performed on a single processor, so it is expected that the ThunderX2 will have the worst performance.

FFT only takes up around 5% of the total compute costs, so overall not a massive win. But in terms of the actual FFT its massively quicker.

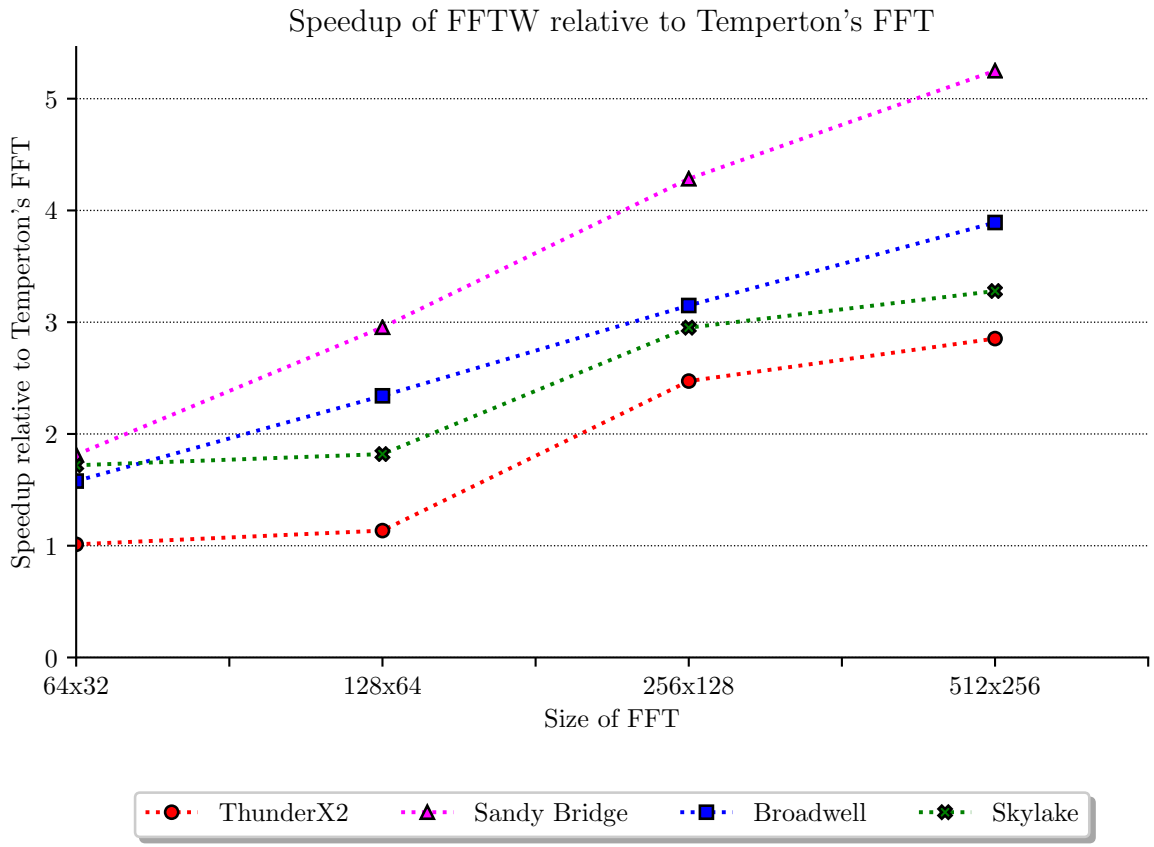


Figure 8.3: Speedup of FFTW relative to Temperton's FFT.

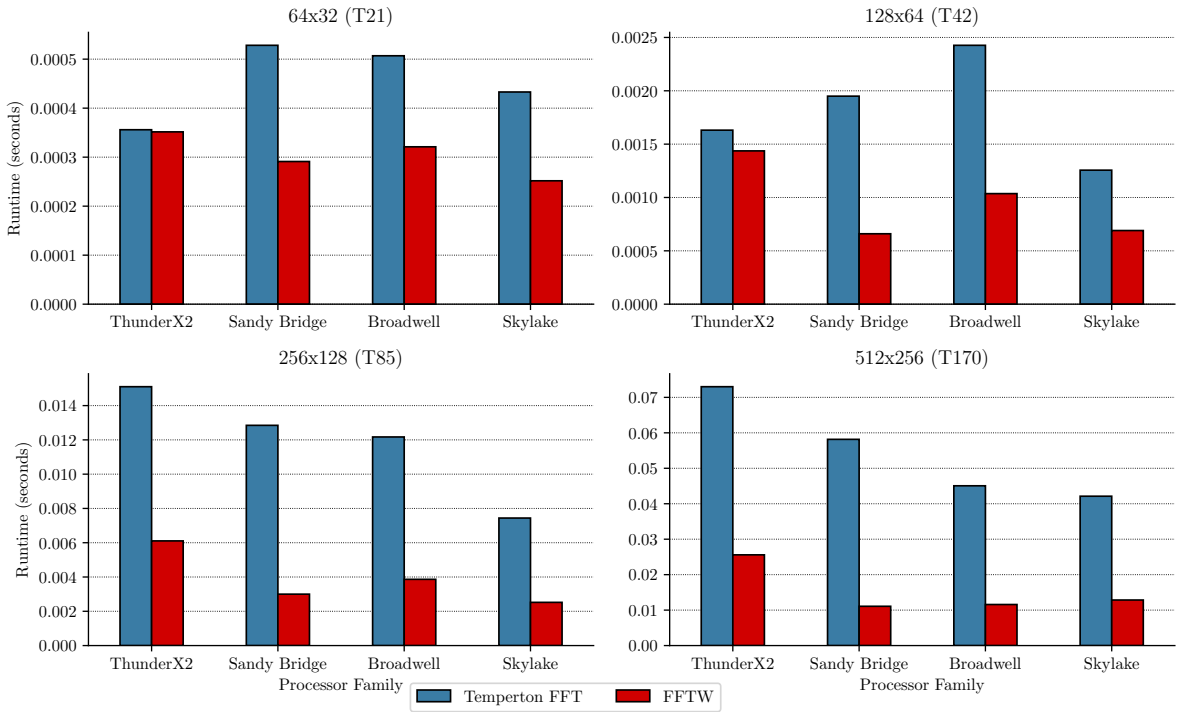


Figure 8.4: The performance of Temperton's FFT compared to the performance of FFTW across multiple domain sizes.

8.2 Floating point precision

On modern processors, the time taken to perform floating point arithmetic on a single data item is about the same for both single and double precision variables. At a larger scale, single precision arithmetic can improve performance by reducing memory bandwidth consumption, and by allowing for more data items to fit in a vector register.

To allow for SIMD instructions to be applied to complex data structures, Isca has been compiled to use single-precision floating point numbers. There is existing infrastructure to allow for this to be done as the codebase includes interfaces to both single and double precision versions of commonly used subroutines. To change the default memory usage of real and complex variables, a number of preprocessor directives and compiler flags can be specified at compile time.

8.2.1 Methodology

To measure the performance difference between single and double precision numbers in Isca, both the Held-Suarez and Grey-Mars configurations were rerun using the single precision configuration. Additionally, the operational intensity (FLOPS/Byte) and performance (GFLOPS/s) were remeasured for the entire run of the executable to allow for comparison with the double precision code.

I suspect that the speedup is caused by vectorisation... is this true?

- Measure runtime of both sp and dp code
- Look at roofline, how has it changed?
- plot vec and no-vec, see how the performance changes

8.2.2 Results

For the single precision code, approximately 36% of the program runtime was spent inside vectorised compute kernels, compared to just 28% of runtime for the double precision code. This suggests that more loops are able to use vector instructions when using single precision variables. Vector reports produced when compiling the code confirm this, as vector instructions are now used to iterate over complex data structures on all processor architectures.

The roofline model in Figure **??**

Figure 8.6 shows that the runtime is significantly reduced when using single precision arithmetic for all node types, and this result is consistent across both the Held-Suarez and Grey-Mars configurations. As the number of processor cores increases, the difference in runtime between the single and double precision code decreases. This is because of load balancing issues discussed in section x.

An additional benefit of using single precision arithmetic is that cache performance is improved and memory-bandwidth consumption is almost halved.

8.2.3 Discussion

In the Fortran 90 standard, single precision real numbers have 7 digits of accuracy, and double-precision real numbers have 15 digits of accuracy. Because of this, the units of last place numerical analysis technique cannot be used to compare single and double precision outputs. To verify

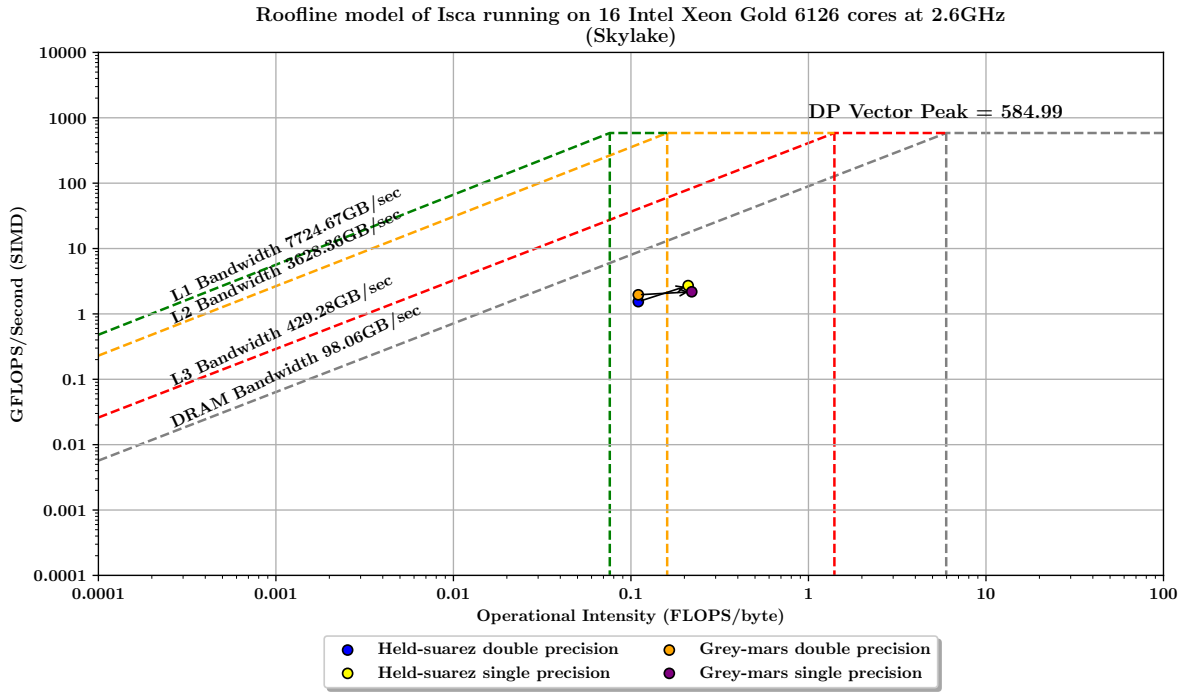


Figure 8.5: Comparison of the single and double precision versions of Isca using the Held-Suarez and Grey-Mars configurations.

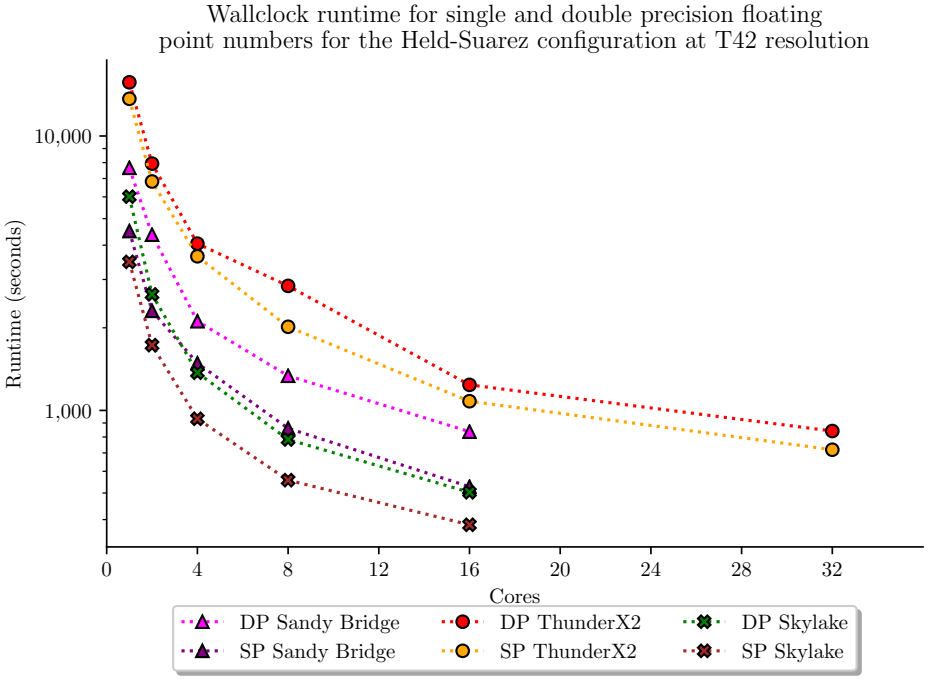


Figure 8.6: Comparison of the wallclock runtimes for single and double precision floating point numbers for the Held-Suarez configuration running at T42 resolution. Single precision arithmetic is significantly faster at all core counts.

the scientific validity of the results, both single and double precision output files were given to domain experts for comparison.

Although Isca is not stochastic, it is chaotic, which means that small changes to variables can lead to drastically different results. However, Isca is mainly used for the estimation of climate behaviour on exoplanets, and therefore outputs results as the averages of many numbers, so less precise variables are okay.

Single precision configurations could be applied when running a perturbed physics ensemble, which is a brute-force approach to model parameter selection. The process involves running many different simulations with a range of parameters. Single precision numbers could be used to find interesting parameter configurations, and promising results can be re-run at higher resolution using double precision.

Look at the time taken to do loops. Plot another roofline model -> how has the oi changed?

Halves memory-bandwidth consumption. Therefore why isn't ThunderX2 performing better?

8.3 Load imbalance

Slowest functions/subroutines as a result of MPI_recv

trans_spherical_to_grid_3d Fields are transformed from spherical harmonics to a grid layout. Communication is required between all ranks to enable this to happen. Maybe we can do some rank reordering, or try to hide the cost of the comms between some compute? Will have to take a deeper look at the function.

area_weighted_global_mean Returns the area weighted global mean of a 2-d field input as a field of values local to the processor.

Further down the call stack, both of these issues are caused by `MPI_wait`. This suggests that there is an imbalance in the MPI. Both these functions are something to do with the FFT, although the root cause of the problem is the MPI wait.

8.4 Memory Leaks

Memory leaks occur when allocated memory that is no longer needed is not released. Isca exhibits a large number of memory leaks whenever a new namelist is read in

There are loads of memory leaks caused by incorrectly reading of namelists. There previous method was also incompatible with the Cray compiler -> maybe incorrect data was being read into the model previously? Was the model even correct? This code has been replaced at x locations throughout the code. Although there was no measurable performance benefit, this makes the code better.

```
    read(input_nml_file, nml=transforms_nml, iostat=io)
    ierr = check_nml_error(io, 'transforms_nml')
```

```
    namelist_unit = open_namelist_file()
    read (namelist_unit, transforms_nml, iostat=io)
    close_file(namelist_unit)
    ierr = check_nml_error(io,'transforms_nml')
```

8.5 Conclusions

Although the FFT was improved, there is very little scope for optimisation without addressing the deeper rooted problem of the MPI imbalance.

Moral of the story is that bespoke code does not age well. It is best to use a popular library that is regularly updated to support the newest hardware.

Part III

Reflection, Critical Analysis and Conclusion

CHAPTER 9

Conclusion and Critical Analysis

This project aimed to present a comprehensive performance analysis and optimisation of the Isca climate model on multiple processor architectures. The following chapter assesses whether this has been achieved, and discusses the limitations of the work presented.

9.1 Benchmarking and performance analysis

9.1.1 Hardware benchmarking

Although this research project presents a grounded performance analysis of numerous high-performance processors, Isca is just one program and the performance of said processors cannot be judged based on this alone. Isca also has a relatively low computational intensity for a high-performance code, and did not push any of the processors tested to their limits. Perhaps more computationally intensive programs like TeaLeaf, CloverLeaf, or other synthetic benchmarking codes would provide a better platform for performance comparison (citation).

9.1.2 Application benchmarking

The results presented in Chapter 7 disproportionately represents the performance of the T42 resolution. This was not intentional, however providing benchmarks across all different combinations of processor, configuration and resolution was challenging to manage, therefore the T42 resolution was chosen as the default for many experiments. The time restrictions of the three month project meant that it was simply not possible to collect results for all possible scenarios.

Although the Grey-Mars and Held-Suarez model configurations are vastly different, they only represent two narrow use-cases of the Isca model. Isca can be used for the simulation of countless other scenarios, ranging from the benignly simple to complex realistic global simulations. Further work could look at other use-cases included in the Isca Github repository, including a 'hot Jupiter', 'bucket hydrology' and realistic Earth simulation using topography (citation).

The work presented in this thesis measured resolutions up to and including T85 only. Isca officially supports resolutions up to T170, however it would be trivial to modify the Python library to allow for simulations of arbitrary granularity. Larger resolutions would allow for the model to be run across 10's of nodes, rather than just the maximum of three used in this project. Although this is non-standard and does not represent how the model is used by researchers, it would be interesting to study the underlying . Higher resolutions would also allow for the ThunderX2 processor to be tested in a mutlinode configuration.

9.2 Load imbalance

It would have been good to have a proper go at fixing the load imbalance -> couldnt do because time restrictions. Also berkley guy said to focus on the 95%, not possible in this case.

Didnt look at resolutions higher than T85, and therefore did not measure the multinode performance of the ThunderX2

9.3 Optimisation

9.4 Further work

- It was difficult to find things to optimise. The biggest time-sink was the MPI, but that was due to unavoidable load-balancing issues that cannot be resolved through asynchronous message passing.
- Prior to starting the project, there was no guarantee that the code could be optimised in any way. A lot of time was spent looking for stuff to optimise.
- I spent a long time at the beginning of the project getting the code to work with the Cray compiler. This was maybe a waste of time.
- I had never used Fortran before, and it was difficult to learn the language with such a massive code base. The codebase contains over 250,000 lines of code and a considerable amount of time was spent getting familiar with the different parts.
- Couldn't do 3 runs for all test cases as there simply wasn't enough time. The model compiles into a special directory, and this means that only one model can be tested at one time.
- Code crashes all the time because it's very delicate.
- Although the Python library helped, it was still very difficult to coordinate all the testing as there were many experiments to do.
- Only benchmarked one code, therefore can't be a good reflection of the ThunderX2
- What do the results mean?
- The codebase takes a long time to compile. The Cray compiler can take upwards of half an hour to compile the Grey-mars configuration. Sometimes a single code change can take half an hour to check if it works.
- As the number of cores per processor increases there is less point in using MPI for smaller codes like Isca. Perhaps this was unavoidable 20 years ago when processors commonly had only 1/2 cores per node but this is not the case anymore. Especially at low resolutions

CHAPTER 10

Reflection

10.1 Challenges

The initial challenge was learning the Fortran programming language, having never used it before this project. This was complicated further by the size and complexity of the Isca codebase, which made it difficult to learn by example. Fortran behaves very differently from other more familiar programming languages like C and Python and a long time was spent learning the intricacies of the language before any real development could take place.

Following this, simply compiling the model was a non-trivial process, and it took 4 weeks before Isca was compiled and run on each cluster. There were many small problems with the codebase that were unique and difficult to find information on, and often required an obscure fix in the form of a compiler flag or environment variable. The BluePebble and Isambard clusters caused many issues. Towards the beginning of the project, a large amount of time was spend trying to compile the model using the Arm HPC compiler, which was not used in the rest of the project. Perhaps this was a waste of time.

Isca can sometimes take up to half an hour to compile. This made it difficult to test changes made to the codebase as minor mistakes required a total recompilation of the model.

Isca is a very fragile codebase, and even very minor changes cause the model to crash. The main source of crashes throughout this project occurred as a result of atmospheric variables exceeding their expected range. The chaotic nature of the model meant that this was a frequent occurrence, as small changes to parameter values would cause vastly different results.

underestimated the amount of time it would take to Compiling the model was non-trivial, and finding

Bibliography

[1] G. K. Vallis, G. Colyer, R. Geen, E. Gerber, M. Jucker, P. Maher, A. Paterson, M. Pietschnig, J. Penn, and S. I. Thomson, "Isca, v1.0: a framework for the global modelling of the atmospheres of earth and other planets at varying levels of complexity," *Geoscientific Model Development*, vol. 11, pp. 843–859, Oct. 2018.

[2] J. Penn and G. K. Vallis, "The thermal phase curve offset on tidally and nontidally locked exoplanets: A shallow water model," *The Astrophysical Journal*, vol. 842, no. 2, p. 101, 2017.

[3] S. I. Thomson and G. K. Vallis, "Atmospheric response to sst anomalies. part i: Background-state dependence, teleconnections, and local effects in winter," *Journal of the Atmospheric Sciences*, vol. 75, no. 12, pp. 4107–4124, 2018.

[4] R. Geen, F. Lambert, and G. Vallis, "Regime change behavior during asian monsoon onset," *Journal of Climate*, vol. 31, no. 8, pp. 3327–3348, 2018.

[5] V. Balaji, "The fms manual: A developer's guide to the gfdl flexible modeling system," tech. rep., Princeton, NJ, USA, 2002.

[6] L. J. Donner, B. L. Wyman, R. S. Hemler, L. W. Horowitz, Y. Ming, M. Zhao, J.-C. Golaz, P. Ginoux, S.-J. Lin, M. D. Schwarzkopf, *et al.*, "The dynamical core, physical parameterizations, and basic simulation characteristics of the atmospheric component am3 of the gfdl global coupled model cm3," *Journal of Climate*, vol. 24, pp. 3484–3519, Jul. 2011.

[7] R. Farneti and G. K. Vallis, "An intermediate complexity climate model (iccmp1) based on the gfdl flexible modelling system," *Geoscientific Model Development*, vol. 2, pp. 73–88, Jul. 2009.

[8] GFDL, "Stream memory-bandwidth benchmark," Jul. 2019.

[9] V. Bjerknes, J. W. Sandström, and O. D. Devik, *Dynamic meteorology and hydrography*. No. 88, Carnegie, 1910.

[10] P. N. Edwards, "History of climate modeling," *Wiley Interdisciplinary Reviews: Climate Change*, vol. 2, pp. 128–139, Dec. 2011.

[11] C. L. Godske and V. Bjerknes, *Dynamic meteorology and weather forecasting*, vol. 605. American Meteorological Society, 1957.

[12] B. Geerts, "Coordinate systems of numerical weather prediction models," Apr. 1998.

[13] Execlim, "Isca: Idealized gcm from the university of exeter," Dec. 2018.

[14] N. Rajovic, P. M. Carpenter, I. Gelado, N. Puzovic, A. Ramirez, and M. Valero, "Supercomputing with commodity cpus: Are mobile socs ready for hpc?," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), pp. 41–53, ACM, Nov. 2013.

[15] G. Conte, S. Tommesani, and F. Zanichelli, "The long and winding road to high-performance image processing with mmx/sse," in *Proceedings Fifth IEEE International Workshop on Computer Architectures for Machine Perception*, pp. 302–310, IEEE, Sept. 2000.

[16] C. Lomont, "Introduction to intel advanced vector extensions," tech. rep., Santa Clara, CA, USA, 2011.

[17] I. Corporation, "Intel architecture instruction set extensions and future features programming reference," tech. rep., Santa Clara, CA, USA, Apr. 2019.

[18] I. Corporation, "Intel xeon processor e5-2680 v4 product specification," Jan. 2019.

[19] J. D. McCalpin *et al.*, "Memory bandwidth and machine balance in current high performance computers," *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, vol. 2, Dec. 1995.

[20] Cavium, "Thunderx2 cn99xx product brief," tech. rep., San Jose, CA, USA, 2018.

[21] S. McIntosh-Smith, J. Price, T. Deakin, and A. Poenaru, "A performance analysis of the first generation of hpc-optimized arm processors," *Concurrency and Computation: Practice and Experience*, p. e5110, May. 2018.

[22] V. Kindratenko and P. Trancoso, "Trends in high-performance computing," *Computing in Science & Engineering*, vol. 13, p. 92, Apr. 2011.

[23] N. Rajovic, A. Rico, F. Mantovani, D. Ruiz, J. O. Vilarrubi, C. Gomez, L. Backes, D. Nieto, H. Servat, X. Martorell, J. Labarta, E. Ayguade, C. Adeniyi-Jones, S. Derradji, H. Gloaguen, P. Lanucara, N. Sanna, J.-F. Mehaut, K. Pouget, B. Videau, E. Boyer, M. Allalen, A. Auweter, D. Brayford, D. Tafani, V. Weinberg, D. Brömmel, R. Halver, J. H. Meinke, R. Beivide, M. Benito, E. Vallejo, M. Valero, and A. Ramirez, "The mont-blanc prototype: An alternative approach for hpc systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, (Piscataway, NJ, USA), pp. 38–50, IEEE Press, 2016.

[24] E. Calore, F. Mantovani, and D. Ruiz, "Advanced performance analysis of hpc workloads on cavium thunderx," in *2018 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 375–382, IEEE, Jul. 2018.

[25] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, *et al.*, "The arm scalable vector extension," *IEEE Micro*, vol. 37, pp. 26–39, Apr. 2017.

[26] A. Rico, J. A. Joao, C. Adeniyi-Jones, and E. Van Hensbergen, "Arm hpc ecosystem and the reemergence of vectors," in *Proceedings of the Computing Frontiers Conference*, (New York NY, USA), pp. 329–334, ACM, ACM, May. 2017.

[27] G. E. Moore *et al.*, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, pp. 33–35, Apr. 1965.

[28] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent ram," *IEEE micro*, vol. 17, pp. 34–44, Apr. 1997.

[29] J. Hammond, "Stream memory-bandwidth benchmark," Jul. 2019.

[30] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary, "Hpl – a portable implementation of the high-performance linpack benchmark for distributed-memory computers," Jan. 2008.

[31] M. Schmidt, "A benchmark for the parallel code used in fms and mom-4," *Ocean Modelling*, vol. 17, pp. 49–67, Dec. 2007.

[32] A. N. S. Programming, *Fortran 90 ISO/IEC 1539 : 1991*. Carnegie, 1991.

[33] GNU, "Implicit conversion between logical and integer," Jul. 2019.

[34] I. M. Held and M. J. Suarez, "A proposal for the intercomparison of the dynamical cores of atmospheric general circulation models," *Bulletin of the american Meteorological society*, vol. 75, no. 10, pp. 1825–1830, 1994.

[35] G. Lancaster, "A collection of scripts for automating runtime and trace data collection for the isca climate model," Aug. 2019.

[36] X. Guo, C. Morales, O. Saastad, A. Shamakina, W. Rijks, and V. Weinberg, "Best practice guide - arm64," tech. rep., Germany, 2019.

[37] S. Behling, R. Bell, P. Farrell, H. Holthoff, F. O'Connell, and W. Weir, "The power4 processor introduction and tuning guide," tech. rep., Armonk, NY, USA, 2001.

[38] C. Carvalho, "The gap between processor and memory speeds," in *Proc. of IEEE International Conference on Control and Automation*, 2002.

[39] M. Frigo and S. G. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[40] M. Frigo, S. G. Johnson, *et al.*, "Fftw for version 3.3.8," May 2018.

[41] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[42] M. T. Heideman, D. H. Johnson, and C. S. Burrus, "Gauss and the history of the fast fourier transform," *Archive for history of exact sciences*, vol. 34, no. 3, pp. 265–277, 1985.

APPENDIX A

Porting code changes

A.1 Isambard

A.2.1 Error 1

```
ftn-356 crayftn: ERROR MPP_DO_GLOBAL_FIELD2D_L8_3D, File = mpp_do_global_field.
    h, Line = 123, Column = 12   Assignment of a INTEGER expression to a LOGICAL
     variable is not allowed.
```

This issue was caused by a variation of the Fortran standard between the Cray, GNU and Intel compilers. GNU and Intel allow for the implicit conversion between logical, integer and real variables, but the Cray compiler does not. To resolve this issue, a new macro was defined to set a default value depending on the type of variables it was being used for.

If `MPP_TYPE_` is of type integer or real, then

```
#define MPP_DEFAULT_VALUE_ 0
```

If `MPP_TYPE_` is of type complex, then

```
#define MPP_DEFAULT_VALUE_ .false.
```

A.2.2 Error 2

```
ftn-1725 crayftn: ERROR COMPUTE_LC1, File = ../../../lustre/home/br-glancaster/
    Isca/src/atmos_spectral/init/polvani_2007.F90, Line = 356, Column = 26
  Unexpected syntax while parsing the assignment statement : "operand" was
    expected but found "-".
```

Cray Fortran requires brackets around all values denoted as negative, for example

```
Tr = T0 + lapse/(zt**-alpha + z(k)**-alpha)**(1./alpha)
```

becomes

```
Tr = T0 + lapse/(zt** (-alpha) + z(k)** (-alpha))**(1./alpha)
```

APPENDIX B

Code listings

B.1 Grid to Fourier subroutine

```
subroutine grid_to_fourier_double_2d_fftw(num, leng, lenc, grid, fourier)

integer(kind=4),             intent(in)    :: num
integer(kind=4),             intent(in)    :: leng
real(C_DOUBLE),              intent(in)    :: grid(leng, num)
complex(C_DOUBLE_COMPLEX),   intent(out)   :: fourier(lenc, num)
real                                       :: fact
integer                                    :: i, j

fact = 1.0 / (leng - 1)

do j = 1, num
  do i = 1, leng - 1
    real_input(i) = grid(i,j)
  enddo

  call dfftw_execute_dft_r2c(real_input_pointer, real_input, complex_output)

  do i = 1, lenc
    fourier(i, j) = complex_output(i) * fact
  enddo
enddo
return
end subroutine grid_to_fourier_double_2d_fftw
```

Listing B.1: Code used to perform an FFT using the FFTW library. This subroutine can be found in the new `fftw.F90` module, and transforms a 2D data structure from the spacial domain to frequency domain.

B.3 Program to time FFT

```
subroutine time_fft()
use fft_mod

real(kind=8)    , allocatable  :: ain(:,:), aout(:,:)
complex(kind=8), allocatable  :: four(:,:)
integer    :: i, j, m, n, k, h, iter, lot
integer    :: ntrans(3) = (/ 128, 256, 512 /)
integer    :: lots(3) = (/ 64, 128, 256 /)
real       :: start_time = 0, stop_time = 0, mean_time_iter = 0, mean_time_full
    = 0, append_time = 0, time_3d_start = 0, time_3d_stop = 0

iter = 100

! test multiple transform lengths
  do m = 1, 3

  ! set up input data
    n = ntrans(m)
    lot = lots(m)
```

```
19
20      allocate(ain(n+1,lot),aout(n+1,lot),four(n/2+1,lot))
21
22      call fft_init(n)
23
24      do k = 1, iter
25          call random_number(ain(1:n,:))
26          four = fft_grid_to_fourier(ain)
27          call cpu_time(start_time)
28              aout = fft_fourier_to_grid(four)
29          call cpu_time(stop_time)
30          append_time = append_time + (stop_time - start_time)
31      enddo
32
33      mean_time_iter = append_time / iter
34
35      append_time = 0.0
36      start_time = 0.0
37      stop_time = 0.0
38
39      do k = 1, iter
40          call random_number(ain(1:n,:))
41          four = fft_grid_to_fourier(ain)
42          call cpu_time(time_3d_start)
43          do h = 1, 25
44              aout = fft_fourier_to_grid(four)
45          enddo
46          call cpu_time(time_3d_stop)
47          append_time = append_time + (time_3d_stop - time_3d_start)
48      enddo
49
50      mean_time_full = append_time / iter
51
52      call fft_end()
53      deallocate (ain,aout,four)
54
55      print *, '( ',n,' x ' ,lot ,' ), mean_iteration_time: '
56      write (*,'(f15.9)') mean_time_iter
57      print *, '( ',n,' x ' ,lot ,' ), mean_full_time: '
58      write (*,'(f15.9)') mean_time_full
59   enddo
60
61  end subroutine time_fft
62
63  end program test
```

APPENDIX C

Compile environment scripts

C.1 Isambard GNU

```
# template for the  gfortran compiler
# typical use with mkmf
# mkmf -t template.ifc -c"-Duse_libMPI -Duse_netCDF" path_names /usr/local/include
CPPFLAGS = -I/usr/local/include
NETCDF_LIBS = `nf-config --fflags  --flibs`

# FFLAGS:
#  -cpp: Use the fortran preprocessor
#  -ffree-line-length-none -fno-range-check: Allow arbitrarily long lines
#  -fcray-pointer: Cray pointers don't alias other variables.
#  -ftz: Denormal numbers are flushed to zero.
#  -assume byterecl: Specifies the units for the OPEN statement as bytes.
#  -shared-intel:  Load intel libraries dynamically
#  -i4: 4 byte integers
#  -fdefault-real-8: 8 byte reals (compatability for some parts of GFDL code)
#  -fdefault-double-8: 8 byte doubles (compat. with RRTM)
#  -O2: Level 2 speed optimisations

FFLAGS = $(CPPFLAGS) $(NETCDF_LIBS) -cpp -fcray-pointer \
          -O2 -ffree-line-length-none -fno-range-check \
          -fbacktrace -target-cpu=arm-thunderx2 -fdefault-real-8 -fdefault-double-8

#         -ftree-vectorize -fopt-info-vec-missed

#FFLAGS = $(CPPFLAGS) $(NETCDF_LIBS) -cpp -fcray-pointer \
#          -O2 -ffree-line-length-none -fno-range-check \
#          -fdefault-real-8 -fdefault-double-8 -fbacktrace \
#              -target-cpu=arm-thunderx2

FC = $(F90)
LD = $(F90) $(NETCDF_LIBS)

LDFLAGS = -lnetcdff -lnetcdf
CFLAGS = -D__IFC
```

C.3 BCP3 Intel

```
# template for the Intel fortran compiler
# typical use with mkmf
# mkmf -t template.ifc -c"-Duse_libMPI -Duse_netCDF" path_names /usr/local/include
CPPFLAGS = -I/usr/local/include
NETCDF_LIBS = `nc-config --libs`

# FFLAGS:
#  -fpp: Use the fortran preprocessor
#  -stack_temps:  Put temporary runtime arrays on the stack, not heap.
#  -safe_cray_ptr: Cray pointers don't alias other variables.
```

```
#   -ftz: Denormal numbers are flushed to zero.
#   -assume byterecl: Specifies the units for the OPEN statement as bytes.
#   -shared-intel:  Load intel libraries dynamically
#   -i4: 4 byte integers
#   -r8: 8 byte reals
#   -g: Generate symbolic debugging info in code
#   -O2: Level 2 speed optimisations
#   -diag-disable 6843:
#        This suppresses the warning: ‘warning #6843: A dummy argument with an explicit INTENT(OUT) declaration is not given an explicit value.‘ of which
#        there are a lot of instances in the GFDL codebase.
FFLAGS = $(CPPFLAGS) -fpp -stack_temps -safe_cray_ptr -ftz -assume byterecl -shared-intel -i4 -r8 -g -O2 -diag-disable 6843
#FFLAGS = $(CPPFLAGS) -fltconsistency -stack_temps -safe_cray_ptr -ftz -shared-intel -assume byterecl -g -O0 -i4 -r8 -check -warn -warn noerrors -debug variable_locations -inline_debug_info -traceback
FC = $(F90)
LD = $(F90) $(NETCDF_LIBS)
#CC = mpicc

LDFLAGS = -lnetcdff -lnetcdf -lmpi -shared-intel
CFLAGS = -D__IFC
```

APPENDIX D

Job submission scripts

D.1 BCP3 (PBS)

```
#!/bin/sh

#PBS -n held_suarez_benchmarking
#PBS -V # export all environment variables to the batch job.
#PBS -d . # set working directory to .
#PBS -q long # submit to the long queue
#PBS -l nodes=1:ppn=16
#PBS -l walltime=72:00:00 # Maximum wall time for the
#PBS -m e -M qv18258@bristol.ac.uk

source activate isca_env
python $BENCHMARK_ISCA/src/main.py -codebase grey_mars -mincores 4 -maxcores 4
    -r T21 -r T42 -r T85
```

D.3 BluePebble (PBS Pro)

```
#!/bin/sh
#PBS -l select=1:ncpus=28:mem=20GB
#PBS -l walltime=72:00:00

module load tools/git/2.22.0
source activate isca_env
python $BENCHMARK_ISCA/src/main.py -mincores 16 -maxcores 16 -r T21 -r T42 -
    codebase grey_mars -fc kind_4
```

D.5 Isambard (PBS Pro)

```
#!/bin/sh

#PBS -q arm
#PBS -l select=1:ncpus=64
#PBS -l walltime=23:00:00
#PBS -M =qv18258@bristol.ac.uk

source ~/isca_env/bin/activate
python $BENCHMARK_ISCA/src/main.py -mincores 4 -maxcores 4 -r T21 -r T42 -r T85
     -codebase held_suarez -fc cray_temp
```

D.7 BCP4 (SLURM)

```
#!/bin/bash

#SBATCH --job-name=benchmark_held_suarez_two_cores
#SBATCH --partition=cpu
#SBATCH --time=4-00:00:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=24
#number of cpus (cores) per task (process)
```

```
#SBATCH --cpus-per-task=1
#SBATCH --output=held_suarez_two_cores_%j.o
#SBATCH --mail-type=ALL
#SBATCH --mail-user=qv18258@bristol.ac.uk

echo Running on host `hostname`
echo Time is `date`
echo Directory is `pwd`

module purge
source $HOME/.bashrc
source $GFDL_BASE/src/extra/env/bristol-bc4
source activate isca_env

$HOME/.conda/envs/isca_env/bin/python $BENCHMARK_ISCA/src/main.py -mincores 2 -
    maxcores 2 -r T21 -r T42 -codebase held_suarez -fc gcc
```