



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



0000064327

# **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

<b>I</b>	<b>Introduction and Background</b>	<b>1</b>
<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	High Performance Computing . . . . .	2
1.2	Aims and Objectives . . . . .	2
1.3	Contributions . . . . .	3
1.4	Heading . . . . .	4
<b>2</b>	<b>A background on climate modelling</b>	<b>5</b>
2.1	The Isca climate model . . . . .	5
2.1.1	Global Circulation Model (GCM) . . . . .	5
2.1.2	Domain decomposition . . . . .	6
2.1.3	Fast Fourier Transform (FFT) . . . . .	7
2.2	Software architecture . . . . .	7
2.2.1	General overview . . . . .	8
2.2.2	Dependencies . . . . .	9
2.2.3	Fortran Libraries . . . . .	9
2.2.4	Python Libraries . . . . .	10
<b>3</b>	<b>A background on HPC hardware and parallel processing</b>	<b>12</b>
3.1	Parallel processing . . . . .	12
3.1.1	Flynns Taxonomy . . . . .	12
3.1.2	Instruction-level parallelism (SIMD) . . . . .	12
3.1.3	Message passing . . . . .	14
3.1.4	Hardware . . . . .	14
3.2	HPC clusters . . . . .	14
3.2.1	BlueCrystal phase 3 (BCP3) . . . . .	14
3.2.2	BlueCrystal phase 4 (BCP4) . . . . .	15
3.2.3	BluePebble (BP) . . . . .	15
3.2.4	Isambard . . . . .	15
<b>4</b>	<b>A background on benchmarking</b>	<b>18</b>
4.1	Cluster benchmarks . . . . .	18
4.1.1	STREAM TRIAD . . . . .	18
4.1.2	High Performance Linpack (HPLinpack) . . . . .	19
4.2	Application benchmarking . . . . .	20
4.2.1	Performance metrics . . . . .	20
<b>5</b>	<b>Porting</b>	<b>22</b>
5.1	Compilers and MPI libraries . . . . .	22
5.1.1	GNU Compiler Collection (GCC) . . . . .	22
5.1.2	Intel Compiler Collection (ICC) . . . . .	22
5.1.3	Cray Compiling Environment (CCE) . . . . .	22
5.2	Libraries etc . . . . .	23
5.3	BlueCrystal phase 3 . . . . .	23
5.4	Discussion . . . . .	23
5.5	BluePebble . . . . .	24
5.5.1	Building modules . . . . .	24

5.5.2	Build Tools . . . . .	24
5.6	Issues . . . . .	24
5.7	Isambard . . . . .	24
5.7.1	GNU Compiler Collection . . . . .	24
5.7.2	Ambiguous arithmetic . . . . .	25
5.7.3	Modifications to codebase . . . . .	25
5.8	Verification of results . . . . .	25
5.8.1	Units of last place . . . . .	26
5.8.2	Rounding errors . . . . .	26
5.8.3	Application . . . . .	26
<b>II</b>	<b>Benchmarking, performance analysis and optimisation</b>	<b>27</b>
<b>6</b>	<b>Benchmarking and Performance Analysis</b>	<b>28</b>
6.1	Model configurations . . . . .	28
6.1.1	Held-Suarez test case . . . . .	28
6.1.2	Grey-Mars test case . . . . .	29
6.1.3	Domain decomposition . . . . .	29
6.2	Automated data collection . . . . .	30
6.2.1	Job submission . . . . .	30
6.3	Scaling study . . . . .	30
6.3.1	Results and discussion . . . . .	31
6.3.2	Conclusions . . . . .	35
6.4	Compiler comparison . . . . .	36
6.5	Communication analysis . . . . .	36
6.6	Performance projection . . . . .	37
<b>7</b>	<b>Optimisation</b>	<b>39</b>
7.1	Roofline model analysis . . . . .	39
7.1.1	Isca performance . . . . .	40
7.2	FFT optimisation . . . . .	40
7.2.1	FFTW . . . . .	42
7.3	Floating point precision . . . . .	45
7.3.1	Methodology . . . . .	46
7.3.2	Results . . . . .	46
7.3.3	Discussion . . . . .	47
7.4	Load imbalance . . . . .	48
7.4.1	Vector Instructions . . . . .	51
7.5	Memory Leaks . . . . .	51
7.6	Conclusions . . . . .	52
<b>III</b>	<b>Reflection and Critical Analysis</b>	<b>53</b>
<b>8</b>	<b>Reflection and Critical Analysis</b>	<b>54</b>
<b>A</b>	<b>Porting code changes</b>	<b>58</b>
A.1	Isambard . . . . .	58
A.2.1	Error 1 . . . . .	58
A.2.2	Error 2 . . . . .	58
<b>B</b>	<b>Code listings</b>	<b>59</b>

B.1	Grid to Fourier subroutine . . . . .	59
B.3	Program to time FFT . . . . .	59
<b>C</b>	<b>Job submission scripts</b>	<b>61</b>
C.1	BCP3 (PBS) . . . . .	61
C.3	BluePebble (PBS Pro) . . . . .	61
C.5	Isambard (PBS Pro) . . . . .	61
C.7	BCP4 (SLURM) . . . . .	61

---

## List of Figures

---

2.1	Grid-point and spectral domain decomposition . . . . .	6
2.2	Flowchart that illustrates the program flow of Isca, when run using its Python library. . . . .	8
2.3	Simple communication pattern; sections of compute are interrupted by calls to a halo exchange. . . . .	9
2.4	Visual output from the Isca model . . . . .	10
3.1	Flynns taxonomy . . . . .	13
3.2	Block diagram of the Cavium ThunderX2 server microprocessor . . . . .	16
4.1	STREAM TRIAD benchmark . . . . .	18
4.2	High performance Linpack benchmark . . . . .	19
6.1	Wallclock runtime of the Held-Suarez configuration running at T21 resolution across all processor architectures. The vertical purple bar on the $y$ plane indicates the maximum number of processors available on a node of BCP3. . . . .	31
6.2	Wallclock runtime of the Grey-Mars configuration across all processor architectures. Vertical coloured bars on the $y$ plane indicate the maximum number of processors available on a node for each cluster. . . . .	32
6.3	Wallclock runtime of the Held-Suarez configuration running at T42 resolution across all processor architectures. The vertical coloured bars on the $y$ plane indicate the maximum number of processor cores available on a node of each cluster. . . . .	33
6.4	Wallclock runtime of the Grey-Mars configuration across all processor architectures. Vertical coloured bars on the $y$ plane indicate the maximum number of processors available on a node for each cluster. . . . .	34
6.5	Wallclock runtime of the Held-Suarez configuration running at T85 resolution across all processor architectures. The vertical coloured bars on the $y$ plane indicate the maximum number of processors available on a node of each cluster. . . . .	35
6.6	Runtimes for the Held-Suarez and Grey-Mars configurations using different compilers. . . . .	37
6.7	Cost per grid point for the Held-Suarez and Grey-Mars configurations at T21 and T42 resolutions. . . . .	38
7.1	Roofline model of Isca on Intel hardware . . . . .	39
7.2	Contiguous and non-contiguous memory access . . . . .	41
7.3	Two-dimensional data layout in Fortran . . . . .	43
7.4	Performance comparison of FFTW and Temperton's FFT . . . . .	45
7.5	Roofline model comparing single and double precision arithmetic . . . . .	46
7.6	Performance comparison of varied precision of floating point numbers . . . . .	47
7.7	Communication matrices for Held-Suarez and Grey-Mars . . . . .	49
7.8	Comparison of the time spent in MPI . . . . .	50

---

## List of Tables

---

3.1	Hardware specifications of the target HPC systems . . . . .	15
5.1	Compilers and MPI libraries used for benchmarking . . . . .	22
6.1	Resolutions and their compatible core counts . . . . .	29
6.2	Number of processor cores used to measure the performance of different compilers at the T21 and T42 resolutions. . . . .	36
7.1	Runtime spent inside two compute kernels for both scalar and vector code . . . . .	40
7.2	TODO . . . . .	51

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**

### 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 been 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) that 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  $1.17\times$  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  $1.69\times$  the original code, and a speedup of  $1.78\times$  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

---

## A background on 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.

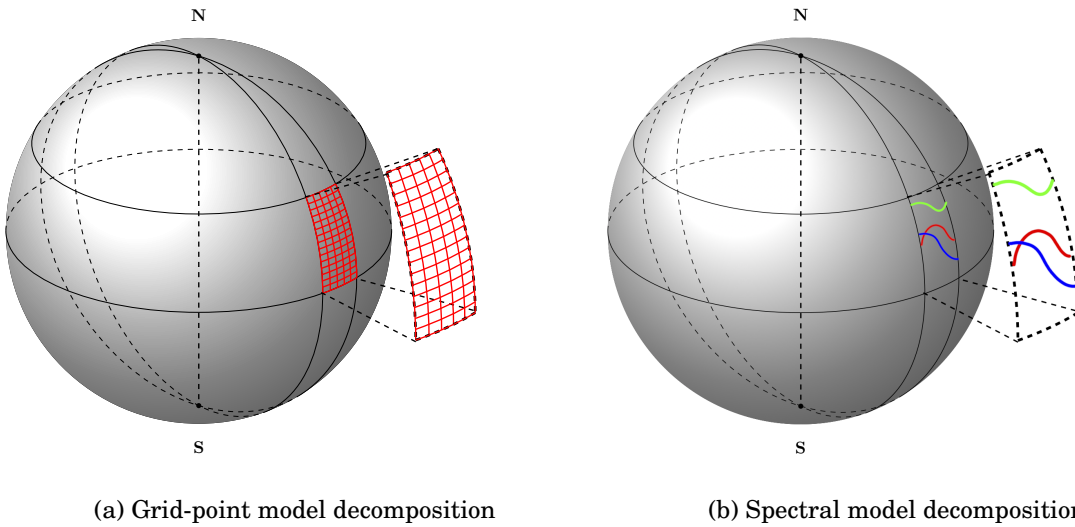


Figure 2.1: A simplified example of a grid-point and spectral model projected onto the sphere. This example uses just 3 waves, which implies a truncation of T3. Most spectral models use a resolution in excess of 21 waves (T21).

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 model uses 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  $\{x_n\} = \{x_0, x_1, \dots, x_{N-1}\}$  into another sequence of complex numbers  $\{X_k\} = \{X_0, X_1, \dots, X_{N-1}\}$ , defined in Equation 2.1.

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

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.

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

## 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.

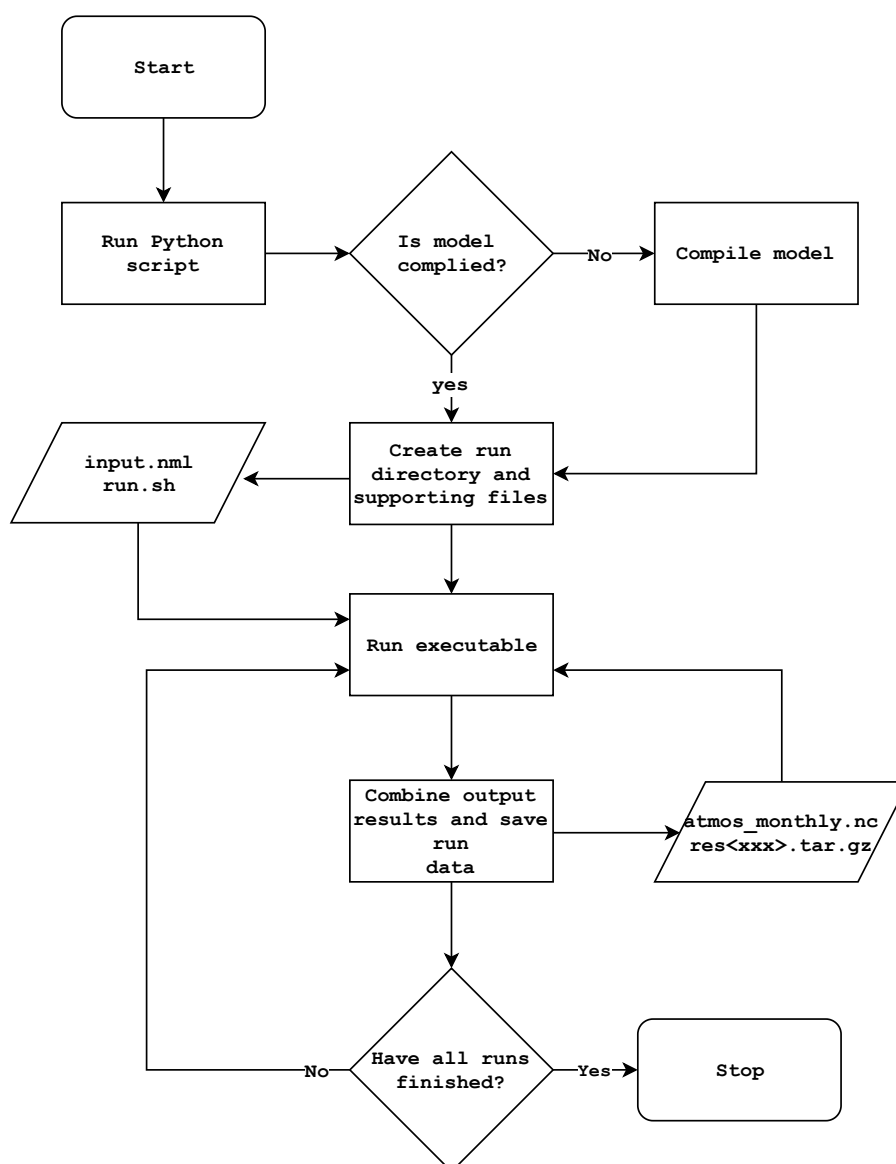


Figure 2.2: Flowchart that illustrates the program flow of Isca, when run using its Python library.



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.

```

1 Time_next = Time + Time_step
2
3 if(idealized_moist_model) then
4     call idealized_moist_phys(...)
5 else
6     call hs_forcing(...)
7 endif
8
9 call spectral_dynamics(Time, ...)
10
11 if(dry_model) then
12     call compute_pressures_and_heights(x, z, ...)
13 else
14     call compute_pressures_and_heights(x, y, ...)
15 endif
16
17 call spectral_diagnostics(Time_next, ...)
18
19 previous = current
20 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.

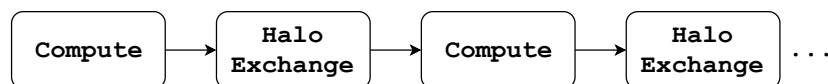


Figure 2.3: Simple communication pattern; sections of compute are interrupted by calls to a halo exchange.

## 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 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 shows an example of a NetCDF file produced by Isca.

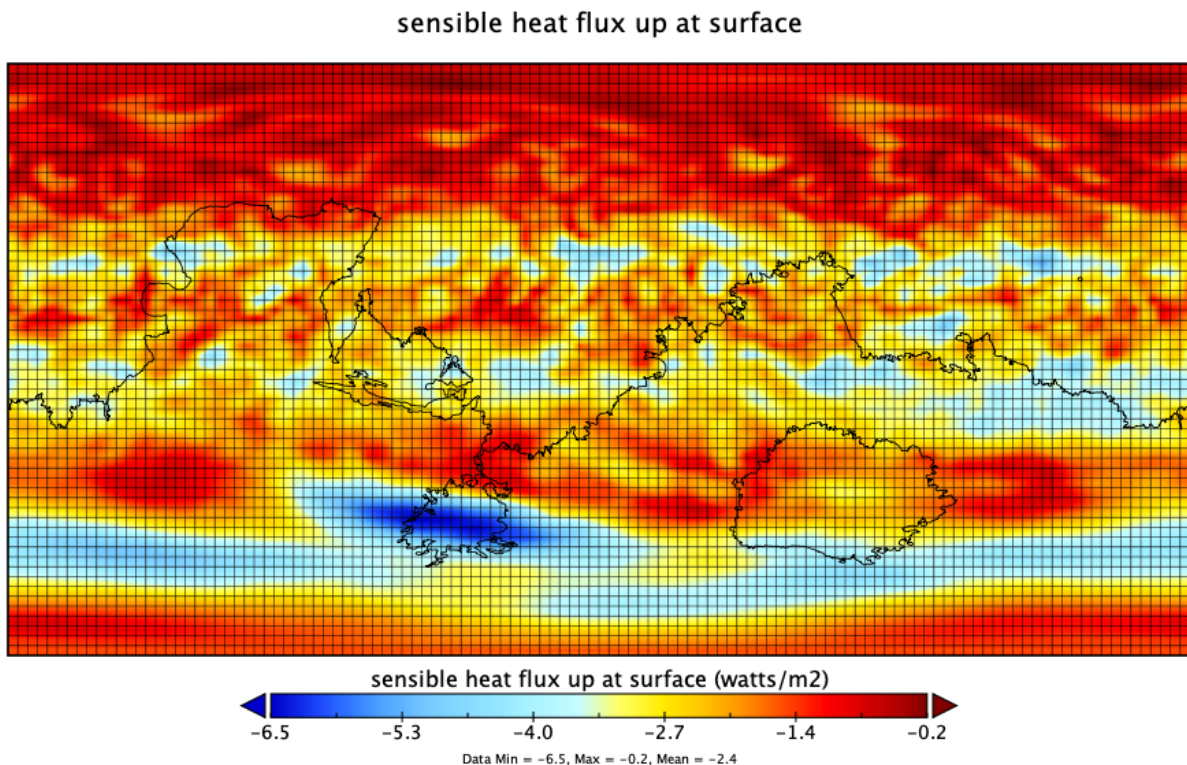


Figure 2.4: Visualisation of an output of the Grey-Mars configuration, showing the amount of heat transferred per unit area per unit time to the Martian surface after 690 days. This visualisation was generated using the Panoply NetCDF data viewer tool (citation).

### 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

---

### A background on 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 Flynn's Taxonomy

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

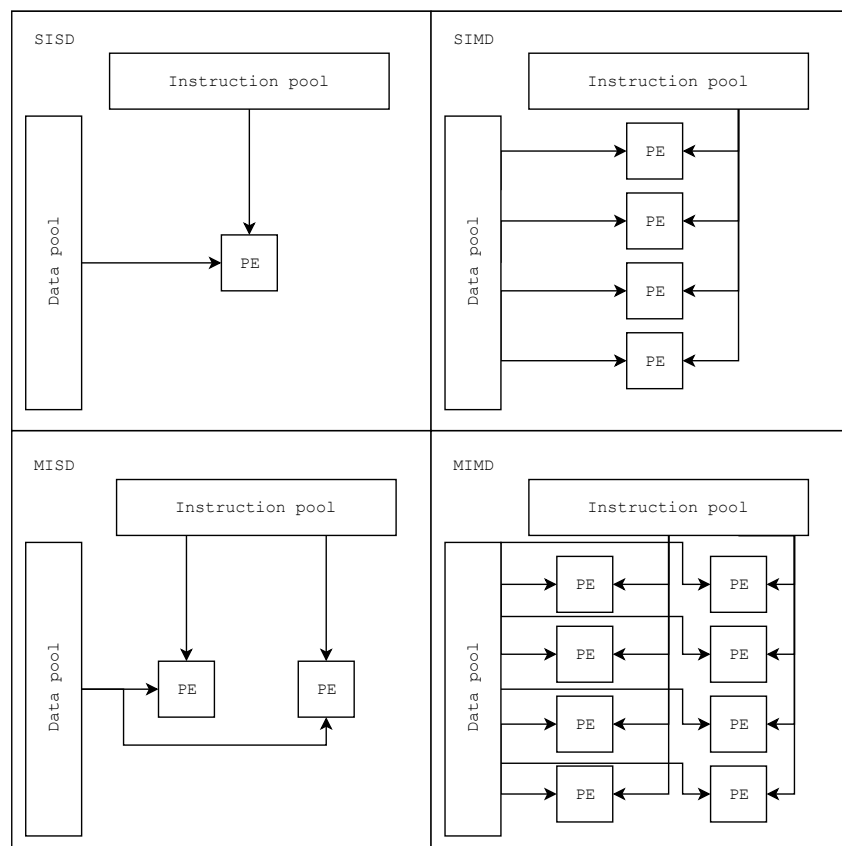


Figure 3.1: Flynn's taxonomy; instructions are applied to data by various processing elements (PE) in different ways.

performed operations on one-dimensional arrays of data, rather than single data items using a 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
	BC3	BC4	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	$2 \times 8$	$2 \times 14$	$2 \times 14$	$2 \times 32$
Memory / Node	64GB	128GB		256GB
Compute Cores	3,568	14,700	-	10,752
Interconnect				

### 3.2.2 BlueCrystal phase 4 (BCP4)

BCP4 has been the University of Bristol's 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 [24]. 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 [23, 21], finding that the processor delivers competitive levels of performance to Intel's line of Xeon processors.

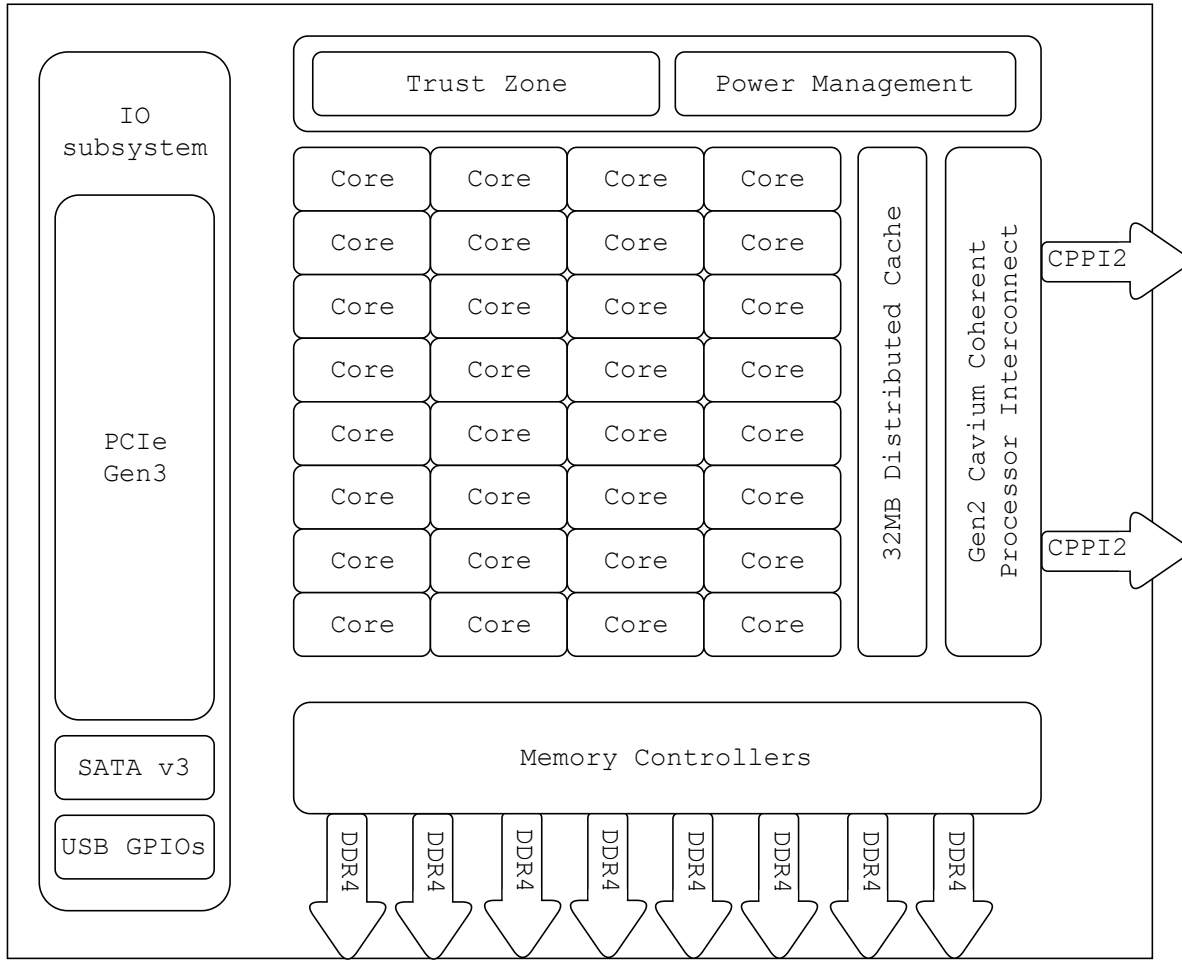


Figure 3.2: Block diagram of the Cavium ThunderX2 server microprocessor. Diagram redrawn from [20]

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 [26, 25]. 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

## A background on benchmarking

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).

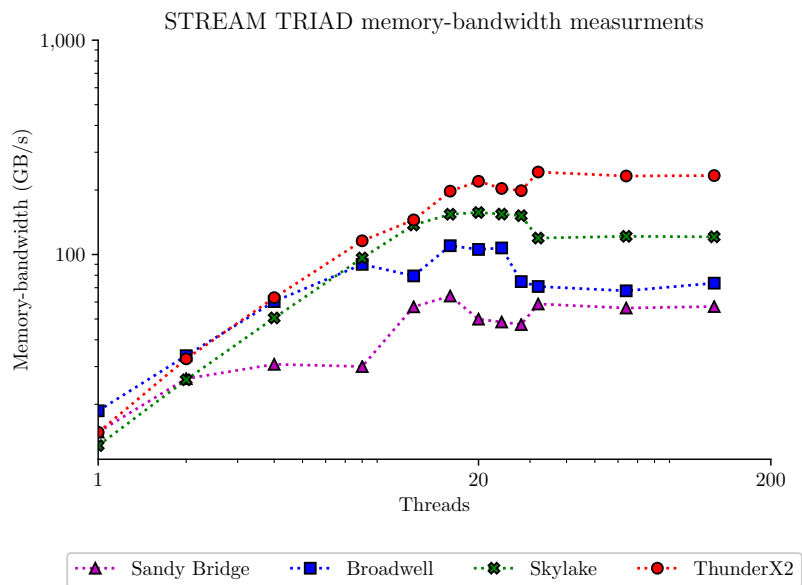


Figure 4.1: STREAM TRIAD results for all processor architectures that have been used as part of this research project.

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.

$$GFLOPS = c \cdot p \cdot i \cdot o \quad (4.1)$$

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.

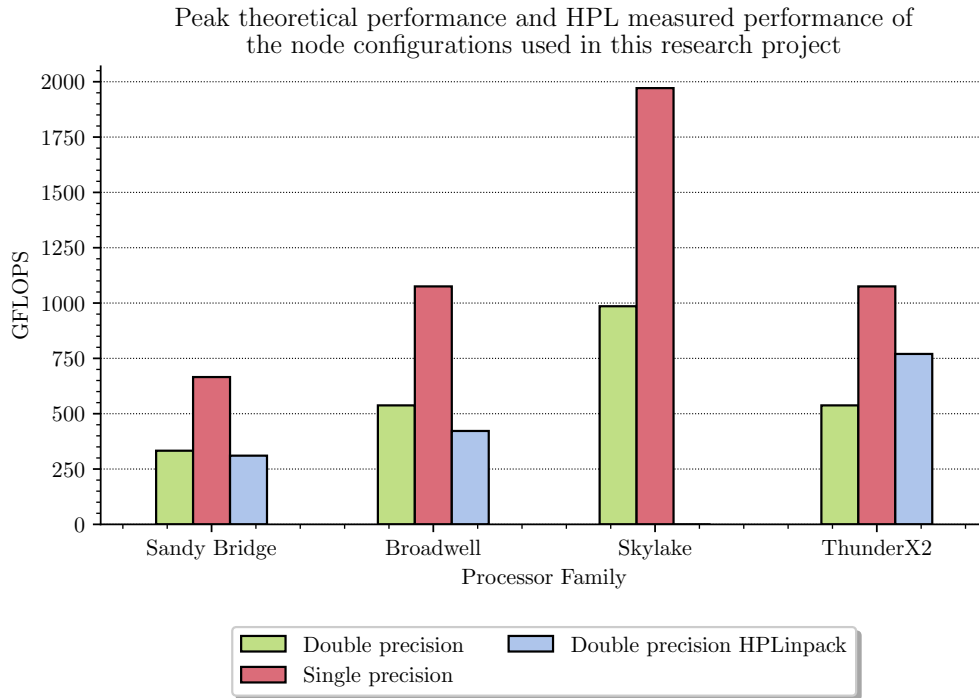


Figure 4.2: Comparison of the theoretical peak machine performance against the performance measured by HPLinpack

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  $R_2$ .

$$S = \frac{R_1}{R_2} \quad (4.2)$$

#### Cost per gridpoint

Other studies that have benchmarked parallel climate codes have used the cost per gridpoint 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.

$$T_p = t \cdot p \quad (4.3)$$

To provide a meaningful comparison between core counts, the cost per gridpoint must be calculated. Given the number of timesteps  $N_t$  and number of gridpoints  $N_g$ , we can calculate the total simulation cost per gridpoint per timestep  $C_{tg}$  as shown in Equation 4.4.

$$C_{tg} = \frac{T_p}{N_t \cdot N_g} \quad (4.4)$$

Although increasing MIMD parallelism by introducing additional processor cores decreases the overall runtime, a greater portion of the runtime is spent 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 determine formally defined in Equation 4.5.

$$I = \frac{W}{Q} \quad (4.5)$$

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 three 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 different tools used when porting Isca, and presents some of the challenges encountered in doing so. As there are too many code small changes to present in this thesis, this chapter...

### 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	OpenMPI
		Intel 13.0.1	OpenMPI
BCP4	Broadwell	GNU 7.2.0	OpenMPI
		Intel 18.0.3	Intel MPI
BP	Skylake	Intel 19.0.3	Intel MPI
Isambard	ThunderX2	CCE 8.7.9	Cray MPI
		GNU 8.2.0	Cray MPI

#### 5.1.1 GNU Compiler Collection (GCC)

The Gnu Compiler Collection (GCC) is a selection of compilers for numerous programming languages produced and maintained by the GNU project.

#### 5.1.2 Intel Compiler Collection (ICC)

The Intel Compiler Collection (ICC)

#### 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. However, the process of Porting for CCE

turned into a stringent debugging exercise that has improved the reproducibility of the code on different platforms. The following section describes some of the code changes required to compile and run Isca using the CCE compiler. A full list of code changes can be found in appendix TODO.

### **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 integers and reals 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].

## **5.2 Libraries etc**

The NetCDF library installed on BCP4 was incompatible with the latest GNU Fortran compiler, therefore the NetCDF library was installed in the `$HOME` directory in order to compile Isca using this configuration. On bcp4 the FFTW library was installed using easybuild, and was therefore slow, so it was reinstalled in my home directory.

## **5.3 BlueCrystal phase 3**

The NetCDF library installed on BCP3 was incompatible with the latest GNU Fortran compiler, therefore the NetCDF library was installed in the `$HOME` directory in order to compile Isca using this configuration.

## **5.4 Discussion**

A large part of the Isca code does not adhere to the Fortran standard.

Additionally, the Fortran standard has evolved over the past twenty years, and many features that were once commonplace are no longer supported by newer compilers. Some of the more popular compilers like the GNU Compiler Collection (GCC) and the Intel Compiler Collection (ICC) have been updated to allow for backwards compatibility with legacy code, however this is not the case for the Cray and Arm compilers that are available on Isambard and Catalyst.

The FMS was first released in 1998. It is a slowly evolving code that relies on

Isca is a strong scaling code. This means that the total program runtime varies with the number of processor cores for a fixed total problem size. The design of the code means that it is very limited in how it can scale.

## 5.5 BluePebble

### 5.5.1 Building modules

Had to build:

- Python 3.7 on Catalyst
- NetCDF on BluePebble
- Git on BluePebble

### 5.5.2 Build Tools

BluePebble only has access to the Intel Fortran and C compilers, and only allows for the use of the Intel MPI implementation. This was a purposeful decision made when designing the cluster with the aim of reducing the number of libraries, and therefore the number of dependencies for each code base. This meant that the Isca codebase had to be compatible with the latest 2019 version of the Intel compiler.

As BluePebble is still in its Beta phase, there are only a limited number of modules on the system. In order to compile Isca, the NetCDF, Git, and Anaconda Python libraries had to be built and made into module files.

## 5.6 Issues

The default stack size on Bluepebble 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 64k.

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 to 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 runtimes for BluePebble were collected using this technique.

## 5.7 Isambard

Although many of Isambard's compute nodes use the Cavium ThunderX2 processor, the machine itself is a Cray XC50. This means that there are a wide range of tools and compilation environments available on this cluster that are unique to Cray systems.

### 5.7.1 GNU Compiler Collection

The GNU compiler collection is commonly used...



This implicit conversion allowed by the GNU and Intel compilers caused many errors for the CCE compiler. To resolve this issue, additional preprocessor directives had to be included in the `mpp_domains_reduce.inc` file.

### **Namelist read errors**

Isca uses Fortran namelist files to read large numbers of parameters into existing variables and data structures at runtime.

The code exhibited different behaviour when compiled using CCE, which suggests that the code relies on some legacy compiler features that are not part of the original Fortran standard, and have remain in the Intel and GNU compilers for backwards compatibility issues.

## **5.7.2 Ambiguous arithmetic**

The Cray compiler required brackets around some arithmetic.

### **Arm HPC Compiler**

The Arm HPC compiler....

## **5.7.3 Modifications to codebase**

On Cray machines, jobs submitted using the PBS pro workload manager must launch MPI applications using the `aprun` utility, which is synonymous to the `mpirun` utility provided by Intel MPI. Prior to this research project, Isca had never been compiled and run on a Cray machine, and the script template used to execute the compiled binary (`run.sh`) only allowed for the use of `mpirun`.

To allow for the code to run using `aprun`, a clause was added to the `run.sh` template script that looks for an environment file called `EXECUTION_TYPE`, which must be set the value `APRUN` if running on a cray system. To allow for backwards compatibility, the script defaults to using `mpirun` if no environment file is defined.

```
1  if [[ -z "${EXECUTION_TYPE}" ]] || [ "${EXECUTION_TYPE^^}" = "MPIRUN" ]; then
2      exec nice -{{nice_score}} mpirun {{mpirun_opts}} -n {{ num_cores }} {{
        execdir }}/{{ executable }}
3
4  elif [ "${EXECUTION_TYPE^^}" = "APRUN" ]; then
5      eexec nice -{{nice_score}} aprun {{mpirun_opts}} -n {{ num_cores }} {{
        execdir }}/{{ executable }}
6  fi
```

A full list of code changes can be found in appendix ??.

## **5.8 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 changes must be non-intrusive, and configurations must default to the original behaviour. In the case of Isca, the code changes that have been made to run on the new machine must not stop the code from working on an old one.

### 5.8.1 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 computer memory, there is a minimum distance 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, although as the model is not stochastic in any way, the same simulation configuration ran on two separate machines should produce identical results, assuming that the compilers use the same amount of memory of each floating point variable.

### 5.8.2 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 than 1 ULP suggests an inconsistent result.

### 5.8.3 Application

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

---

### Benchmarking and Performance Analysis

---

This chapter presents an extensive performance analysis of the Isca model running two unique configurations. The aim of this is to characterise the code on different hardware configurations, aid in the identification of performance optimisations and to provide a comparison of the processors themselves. To do this, the following experiments were performed:

**Scaling study** The runtimes of two model configurations have been measured on four different compute nodes. This involved varying the core count and spatial resolution on each system.

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

**Performance projection** Using the results from the scaling study, 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 SIMD registers of up to 512-bits.

### 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  $23.5^\circ$  tilt, and Mars’ axis is at  $25^\circ$  (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.

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	$32 \times 64$	1, 2, 4, 8, 16
T42	$64 \times 128$	1, 2, 4, 8, 16, 32
T85	$128 \times 256$	1, 2, 4, 8, 16, 32, 64
T170	$256 \times 512$	1, 2, 4, 8, 16, 32, 64, 128

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, 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 define a strict method of data collection. When benchmarking high performance applications, data collection can be a timely 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 (citation).

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.

In order to collect reliable data, a full node was used for each experiment. This means that 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 variation 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. The results presented in the following section are the consequence of over 2,000 hours of experimental runtime.

### 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 then 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 C.

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 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 to model its performance at various levels of complexity and realism.

Whilst the majority of the results presented in this section were to be 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.

### 6.3.1 Results and discussion

For the Held-Suarez configuration, each spatial resolution was run up to the maximum.

The Grey-Mars simulation can only be run on a maximum resolution of T42.

**Held-Suarez T21:** 1, 2, 4, 8, 16

**Held-Suarez T42:** 1, 2, 4, 8, 16, 32

**Held-Suarez T85:** 1, 2, 4, 8, 16, 32, 64

#### T21 resolution

Why are lower resolutions good?

The scaling curve as shown in Figure 6.1 shows how the runtime varies as a function of processor cores for the Held-Suarez configuration at the T21 resolution, and can be described as a sublinear plateau for all node configurations. 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.

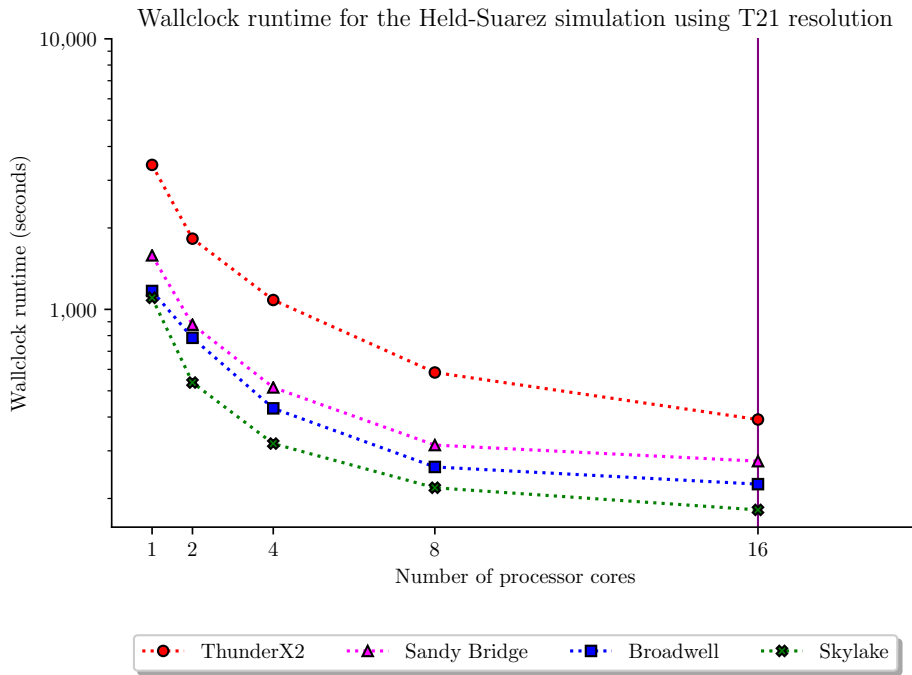


Figure 6.1: Wallclock runtime of the Held-Suarez configuration running at T21 resolution across all processor architectures. The vertical purple bar on the  $y$  plane indicates the maximum number of processors available on a node of BCP3.

Interestingly, when increasing the number of processor cores from 8 to 16 in the Held-Suarez case, there is only an approximate  $1.15\times$  performance gain for the Intel processors. Results like these are typical of many parallel codes, whereby the performance benefit of additional compute resources decreases as more processor cores are utilised. This effect is similar to the law of diminishing returns observed in Economics. As this problem size is small, more time is spent in communication relative to compute when a greater number of processor cores are used.

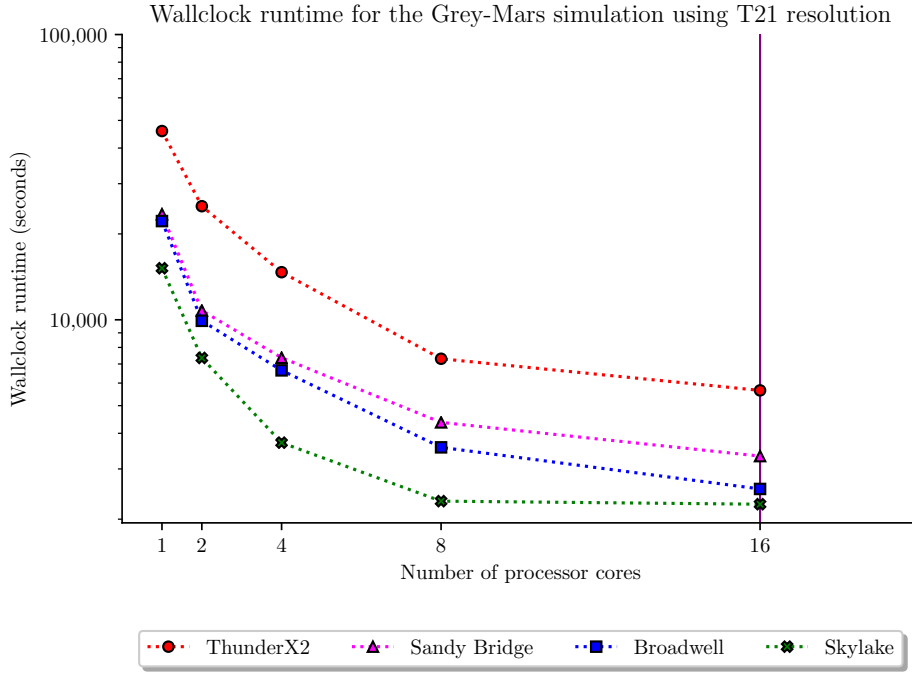


Figure 6.2: Wallclock runtime of the Grey-Mars configuration across all processor architectures. Vertical coloured bars on the  $y$  plane indicate the maximum number of processors available on a node for each cluster.

The trend observed for the Grey-Mars configuration in Figure 6.2 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. The Skylake processor massively outperforms all other processors when run on 8 cores, however it quickly tapers off when run on 16 cores, providing just a  $1.02\times$  speedup.

To determine the importance of memory bandwidth at this resolution, the 16 core configuration was rerun on the Broadwell processor. However, the cores were split across two nodes. Figure x shows that...

### T42 resolution

The T42 resolution in Figure 6.3 presents a similar scaling curve to that observed for the T21 resolution. 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. However for the Intel processors, running on more than 16 cores requires multiple nodes, and this has a dramatic impact on the performance of the Sandy Bridge and Skylake processors.

In comparison to the ThunderX2, which still runs within a single socket when using 32 processor cores, we can determine that this issue is most likely caused by internode communication rather than a problem with the model itself. Interestingly, this behaviour is not apparent on the Broadwell processor, which improves its performance as expected when increasing the number of processor cores used from 16 to 32, even though two nodes are used in this case. To determine the cause, will need to to a mpi trace.

The worst performance observed for the multinode configuration is the Sandy Bridge processor, which takes  $82\times$  longer to complete the same job than when running on a single node. Both BCP3 and BluePebble are intended to be used for smaller node jobs on a single node, so this result could



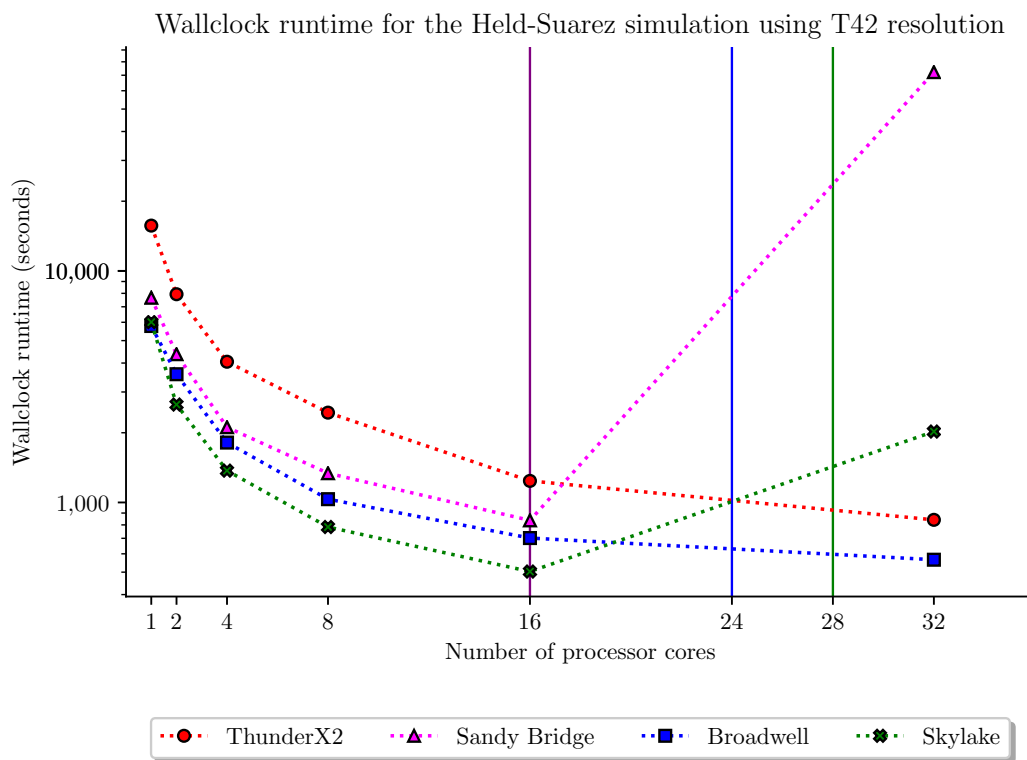


Figure 6.3: Wallclock runtime of the Held-Suarez configuration running at T42 resolution across all processor architectures. The vertical coloured bars on the  $y$  plane indicate the maximum number of processor cores available on a node of each cluster.

be explained by a slower interconnect.

Generally, running on a greater number of nodes improves the performance of codes that are bound by memory bandwidth, as a greater number of nodes implies a larger memory bandwidth.

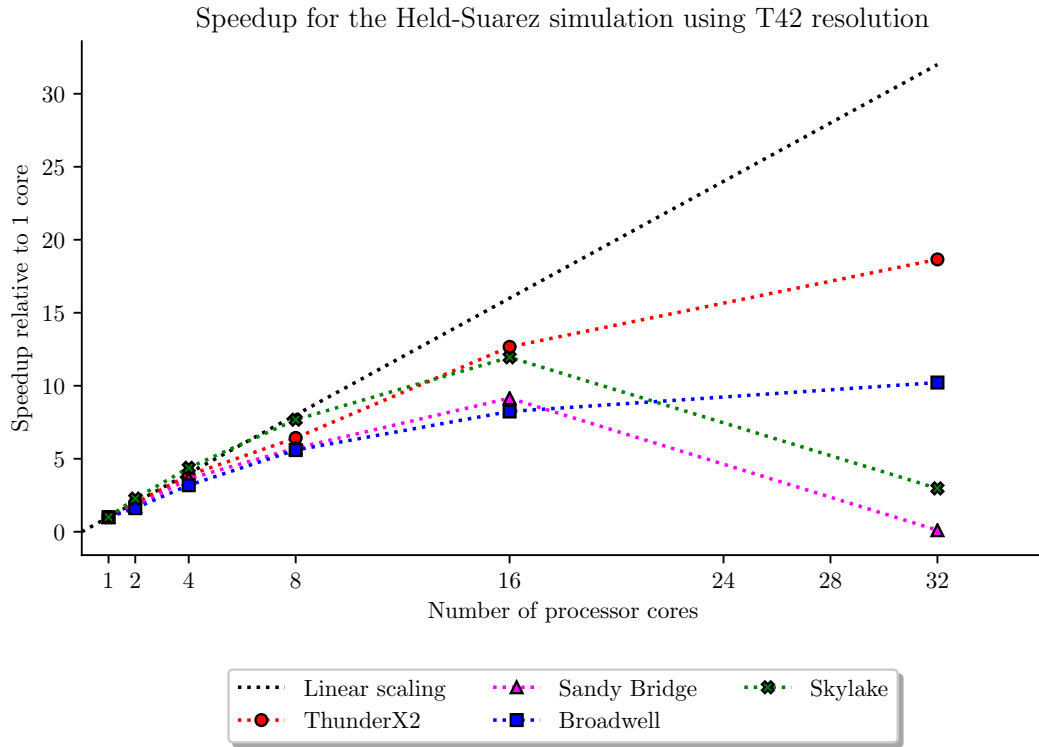


Figure 6.4: Wallclock runtime of the Grey-Mars configuration across all processor architectures. Vertical coloured bars on the  $y$  plane indicate the maximum number of processors available on a node for each cluster.

### T85 resolution

Due to restrictions imposed by domain decomposition, the Grey-Mars configuration cannot be run at a resolution 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 longer than 360 hours.

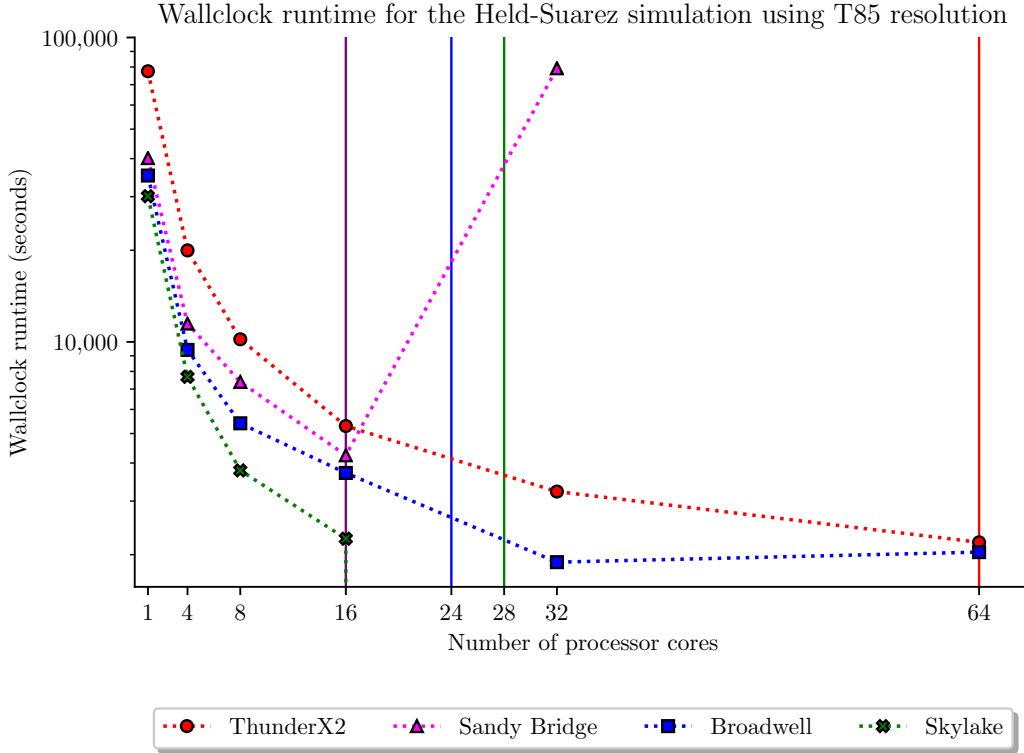


Figure 6.5: Wallclock runtime of the Held-Suarez configuration running at T85 resolution across all processor architectures. The vertical coloured bars on the  $y$  plane indicate the maximum number of processors available on a node of each cluster.

Plotting the scaling graph as a function of speedup in relation to the serial performance, we can see that... When run on 64 cores across four nodes, the performance of the Broadwell processor

The results obtained for the T85 resolution are similar to the T42 resolution. A similar behaviour is observed for the largest problem size of T85.

One possible reason for this phenomenon is that the MPI library is using shared memory for communication.

The BluePebble and BCP3 supercomputers were designed to be used for single node jobs, and larger communication times may be as a result of the interconnect?

### 6.3.2 Conclusions

This scaling study highlighted the differences in processor architecture between Arm and Intel. Arms approach to processor design relies on a greater number of simpler cores, in comparison to Intel's fewer more complex cores. This design benefitted the performance of Isca when run on the

ThunderX2 by keeping communication inside a single node for all resolutions tested and allowed for competitive runtimes to that of the Intel processors.

An interesting issue has been uncovered on the BCP3 (Sand Bridge) and BP (Skylake) supercomputers, whereby Isca performs worse when run in a multi-node configuration. Further analysis of this problem has been discussed in section x. Although the different run configurations have demonstrated

BluePebble is slow because it just uses ethernet.

Although the ThunderX2 has the largest memory bandwidth, it does not have the best runtime at all resolutions.

It would be interesting to run the model at much higher resolutions to see how it reacts to processors at very high core counts.

## 6.4 Compiler comparison

To find the optimal compilation settings for each processor, both the Held-Suarez and Grey-Mars model configurations were compiled 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. 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. This means that the model was ran up to the maximum number of cores available on each node for the given model configuration as shown in Table 6.2. This was done to provide a comparison of the compilers in relation to the performance available on other processors.

Figure 6.6 shows that for all configurations tested on Intel nodes, the Intel compiler outperformed the GNU compiler. This is consistent with what was expected, as Intel has developed their compiler alongside their processors and it therefore produces well-optimised instructions. For the ThunderX2, the GNU compiler outperformed the Cray compiler. Neither compiler is developed specifically for the ThunderX2, therefore this performance difference can be explained by the fact that the GNU compiler is older and therefore more mature than the Cray compiler, however this is just speculation and has not been investigated. To confirm these observations, the test was repeated for the T21 resolution, which yielded similar results.

## 6.5 Communication analysis

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.

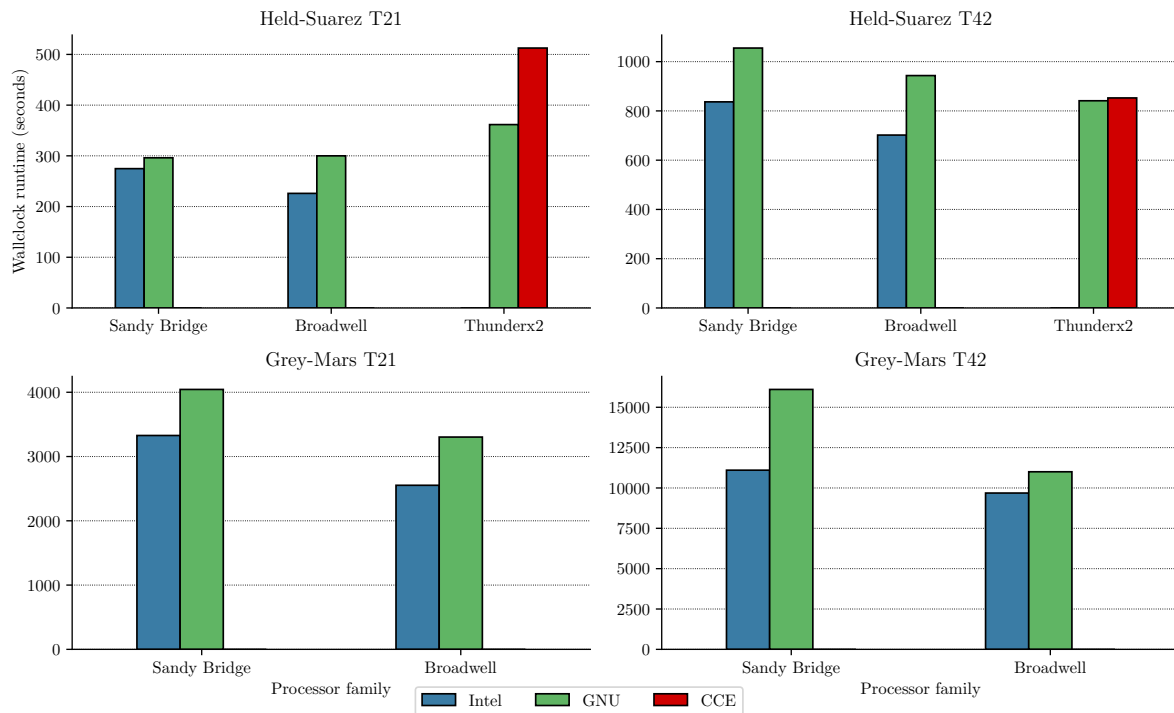


Figure 6.6: Runtimes for the Held-Suarez and Grey-Mars configurations using different compilers.

The rate of computation is defined by the cost of computing a single grid cell per time step of the simulation, relative to the number of processors used. When run in serial, 100% of the program runtime is spent on compute as there is no interprocessor communication.

The measured rate of computation for the three model resolutions is shown in Figure 6.7.

## 6.6 Performance projection

The A64FX processor

- list processor specs
- compare specs to Thunderx2
- Does the A64fx clock down?

Using these results, how do I think that the code will perform on the new processors?

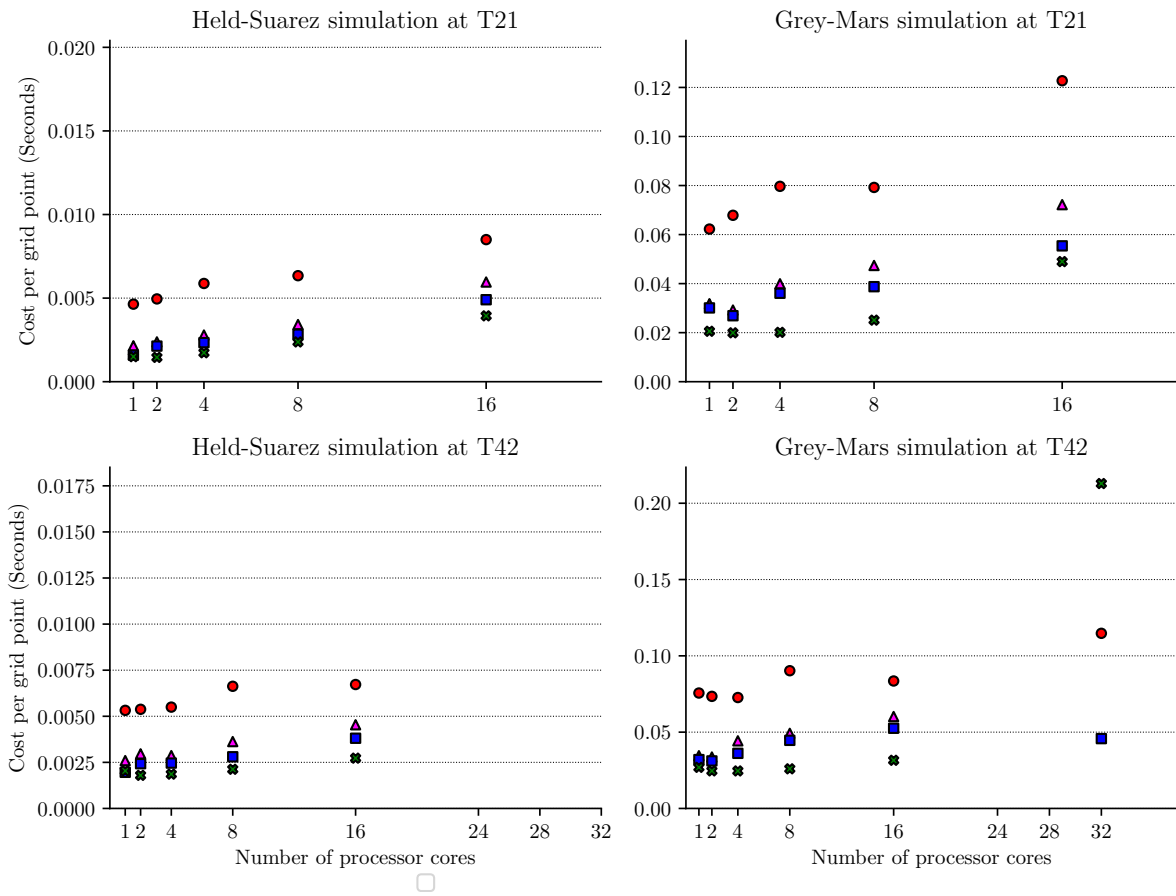


Figure 6.7: Cost per grid point for the Held-Suarez and Grey-Mars configurations at T21 and T42 resolutions.

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.

### 7.1 Roofline model 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.

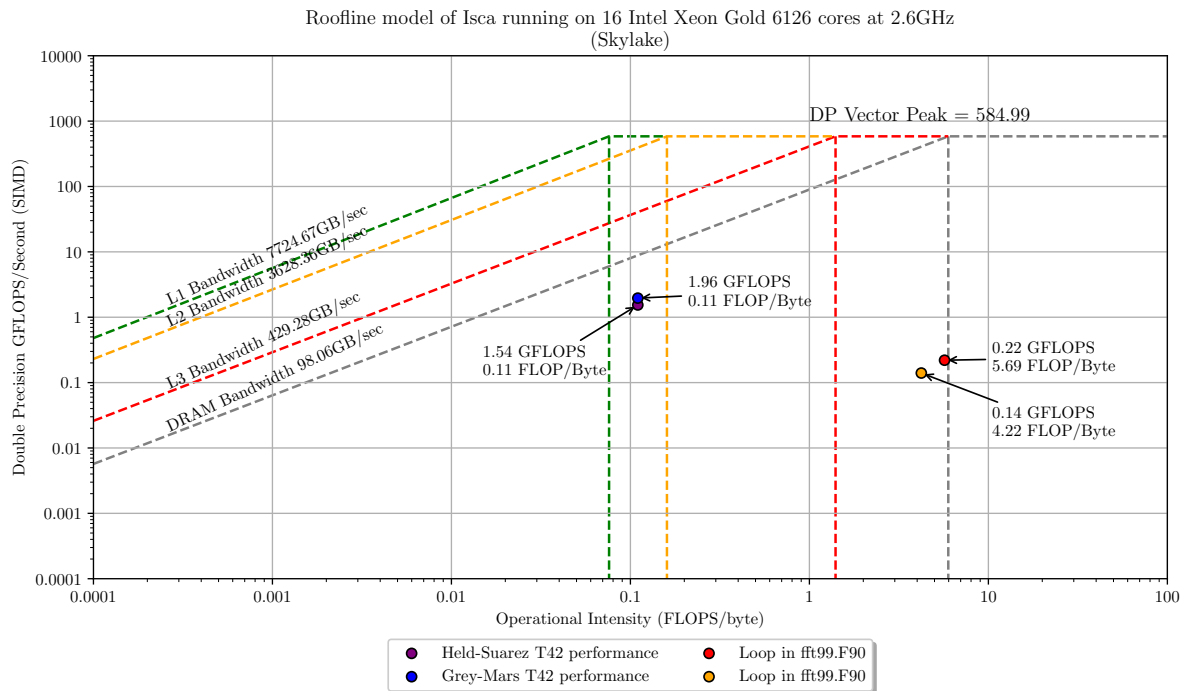


Figure 7.1: Roofline model for 16 cores of two Intel Xeon Gold processors at 2.6 GHz.

### 7.1.1 Isca performance

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. The roofline model in Figure 7.1 indicates that the configuration is limited by memory-bandwidth, as the program total performance is located underneath the DRAM ceiling. Intel Advisor suggests that the theoretical peak double-precision 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 to the hardware.

In addition to quantifying the performance of Isca, Figure 7.1 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. The following section investigates the limitations of these kernels and describes how they have been improved.

## 7.2 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 7.2.

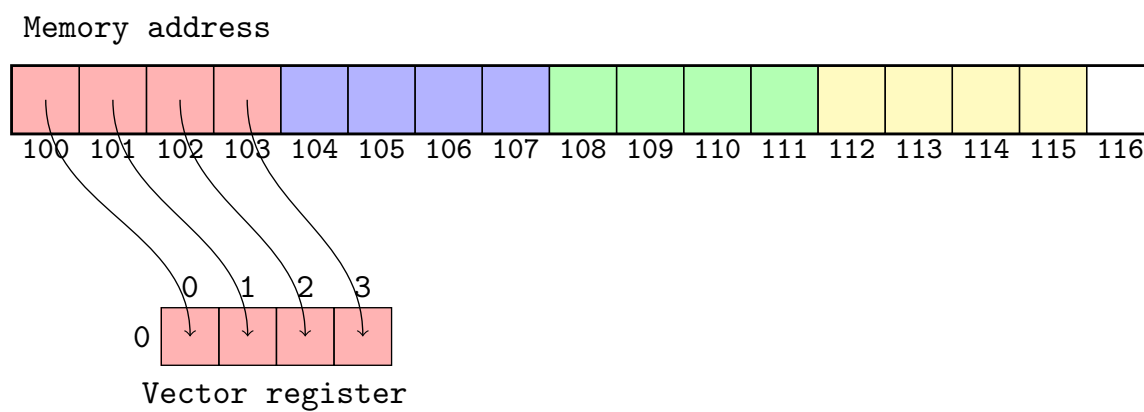
When forcing the compiler to use vector instructions by using the `!DIR$ VECTOR ALWAYS` preprocessor directive, there is only a marginal improvement over the scalar code, as shown in Table 7.1. This provides evidence to support that the code has not been written to vectorise on modern hardware. Also, 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 7.1: Runtime spent inside two compute kernels for both scalar and vector code

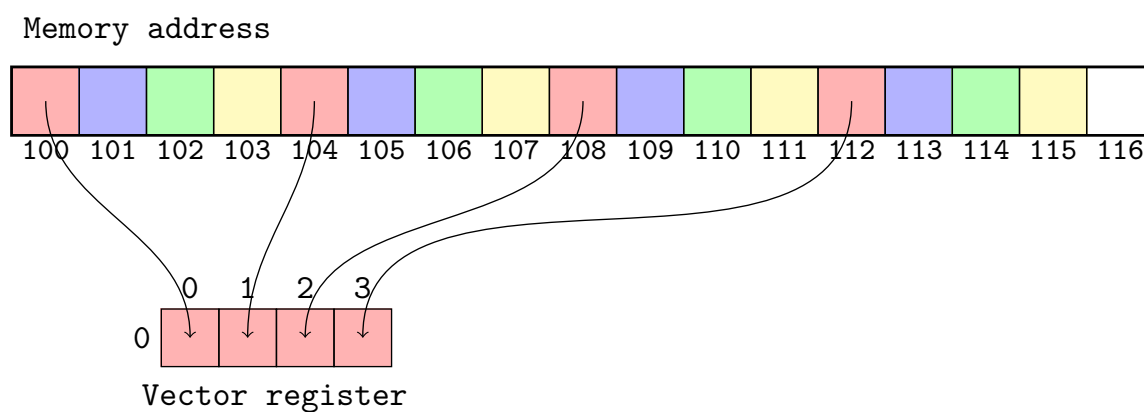
Loop	Scalar time (milliseconds)	Vector time (milliseconds)
(a) <code>vpassm:1081</code>	822.5	649.8
(b) <code>vpassm:1049</code>	793.6	459.8

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





(a)



(b)

Figure 7.2: (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.

important in the late 80's and early 90's when memory was in short supply, however modern 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 be 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.

### 7.2.1 FFTW

FFTW is an implementation of a DFT that aims to adapt to the hardware on which it is run [35]. 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 [35]. 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 [35]. 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 successively 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 [36]. 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 [37, 38]. 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](http://www.fftw.org/fftw2_doc/fftw_3.html)

$$Y_i = \sum_{j=0}^{N-1} X_j e^{-2\pi i j \sqrt{-1/N}} \quad (7.1)$$

$$Y_i = \sum_{j=0}^{N-1} X_j e^{2\pi i j \sqrt{-1/N}} \quad (7.2)$$

### FFTW applied to Isca

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.

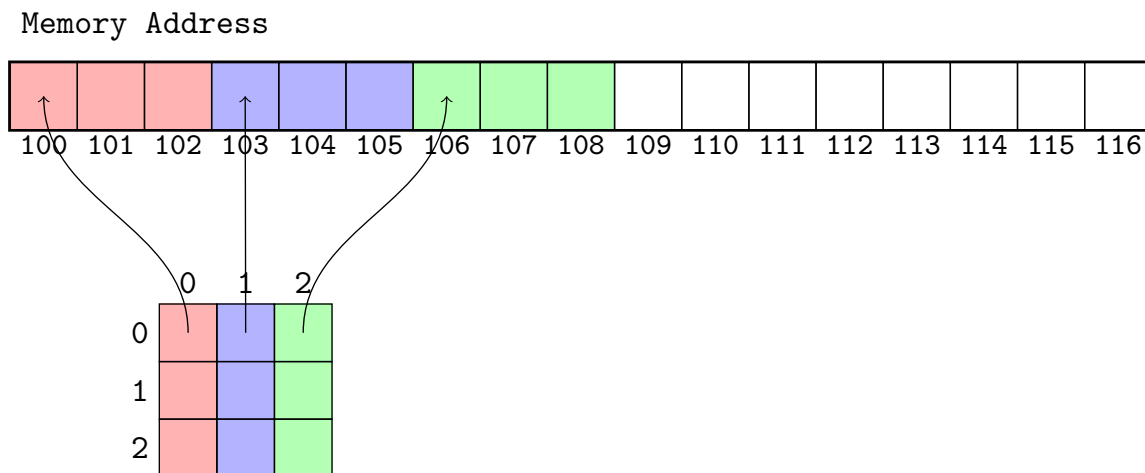


Figure 7.3: Contiguous data layout in memory for a two-dimensional array in Fortran.

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 7.3 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 FFTW's 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.

## Methodology

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.

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  $128 \times 64$  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 ( $128 \times 64$ ), T85 ( $256 \times 128$ ) and T170 ( $512 \times 256$ ) 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. 7.4.

## Results

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  $5.25\times$  quicker than the Temperton FFT. Interestingly, a greater performance gain was measured when the FFT was applied to the largest data structure. why?

Interestingly, the performance gain was only marginal for the T42 resolution when run on the ThunderX2 processor. Interestingly, the T170 resolution gave the greatest performance boost, which suggests that the actual computation is much quicker...

Regardless of whether FFTW provides a significant improvement to it's 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 it's massively quicker.

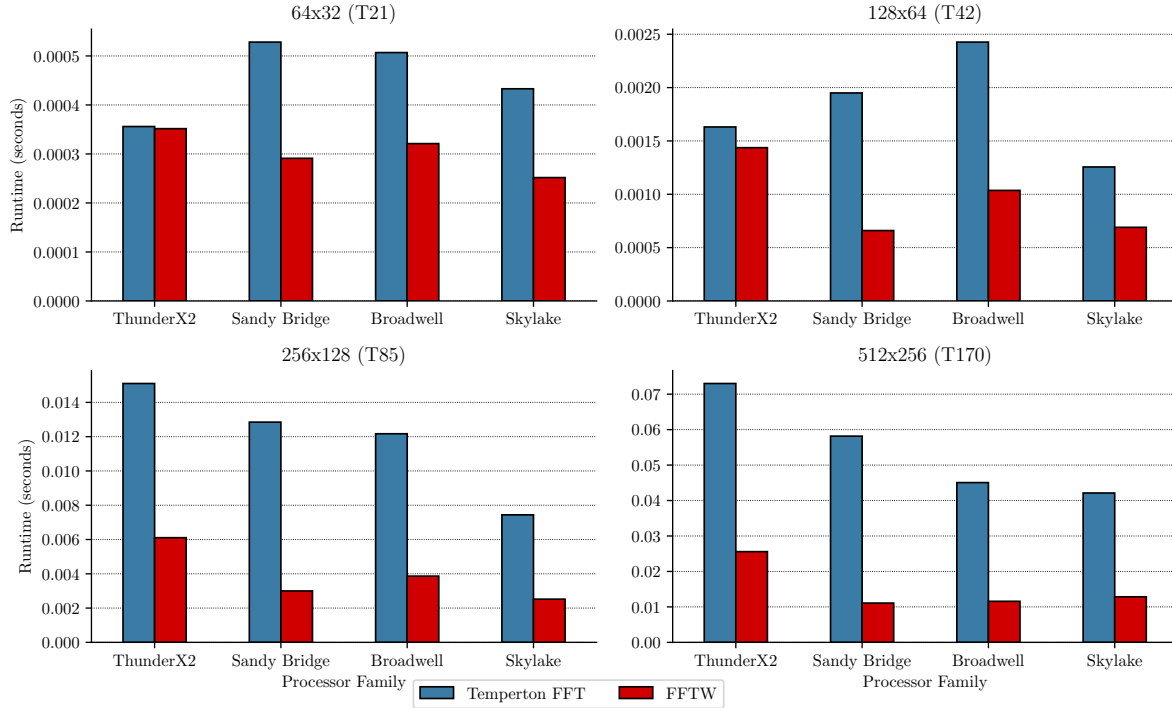


Figure 7.4: The performance of Isca’s bespoke Temperton FFT compared to the performance of FFTW across multiple domain sizes.

### 7.3 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.

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 AVX2 and AVX512 respectively.

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.

### 7.3.1 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.

- Measure runtime of both sp and dp code
- Look at roofline, how has it changed?

### 7.3.2 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 ??

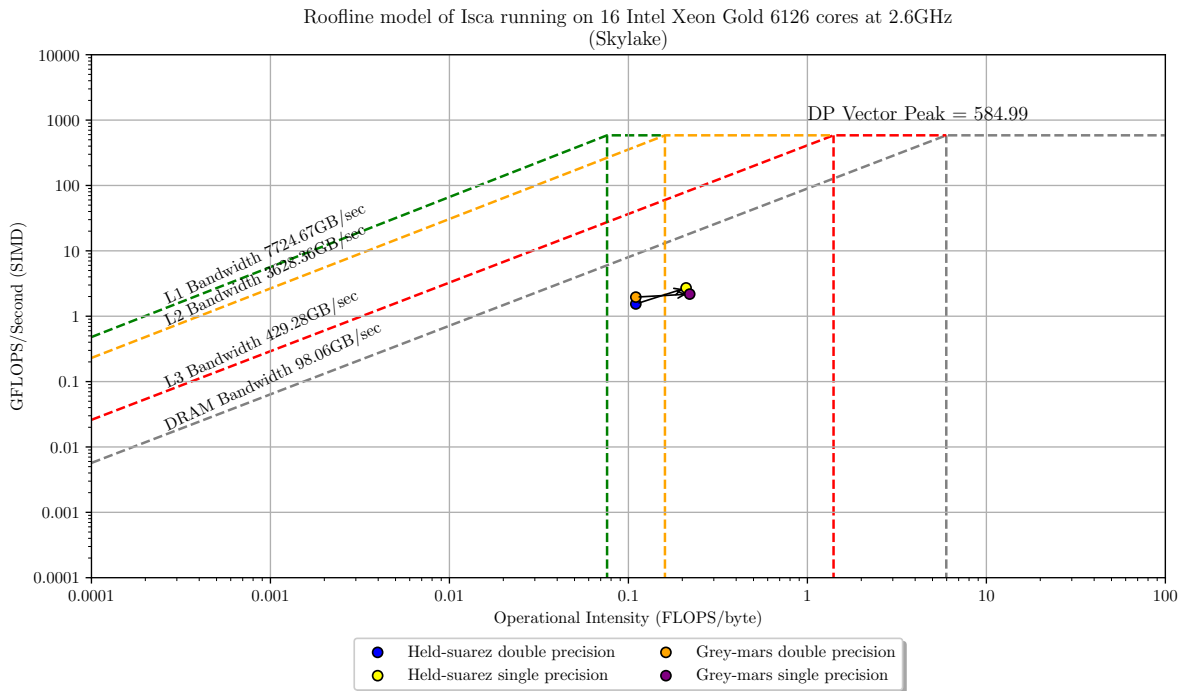


Figure 7.5: Comparison of the single and double precision versions of Isca using the Held-Suarez and Grey-Mars configurations.

Figure 7.6 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.

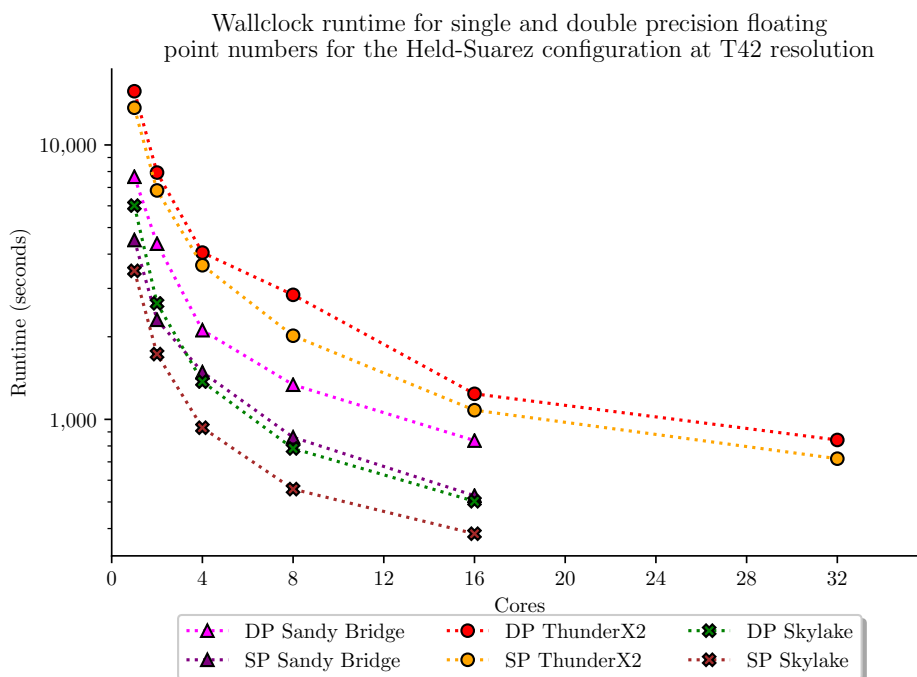


Figure 7.6: Comparison of the wallclock runtimes for single and double precision floating point numbers for the Held-Suarez configuration running at T42 resolution. Single precision arithmetic is significantly faster at all core counts.

### 7.3.3 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 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?

## 7.4 Load imbalance

Isca exhibits a significant load-balancing issue whereby over 30% of the program runtime is spent inside calls to `mpi_recv`. It is not possible to improve upon this code due to the deep rooted imbalance throughout the code. This is just a necessary point of synchronisation that the code requires in order to operate. In order to make the code run faster it would require a complete re-write. Perhaps it may benefit from using some shared memory here, as

Graph of communication pattern

Some special purposes require more expensive communication. For bit-reproducible global sums over 2D fields each processor gathers the pieces from all other processors to construct the global field. Examples for such expensive operations are surface flux corrections to maintain the total volume or salinity budget. However, operations of this kind are rare in MOM-4 and can be avoided completely for regional models.

Switched to an asynchronous call but this led to segmentation faults. use some kind of graph to prove that there is a deeper rooted issue here.

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.



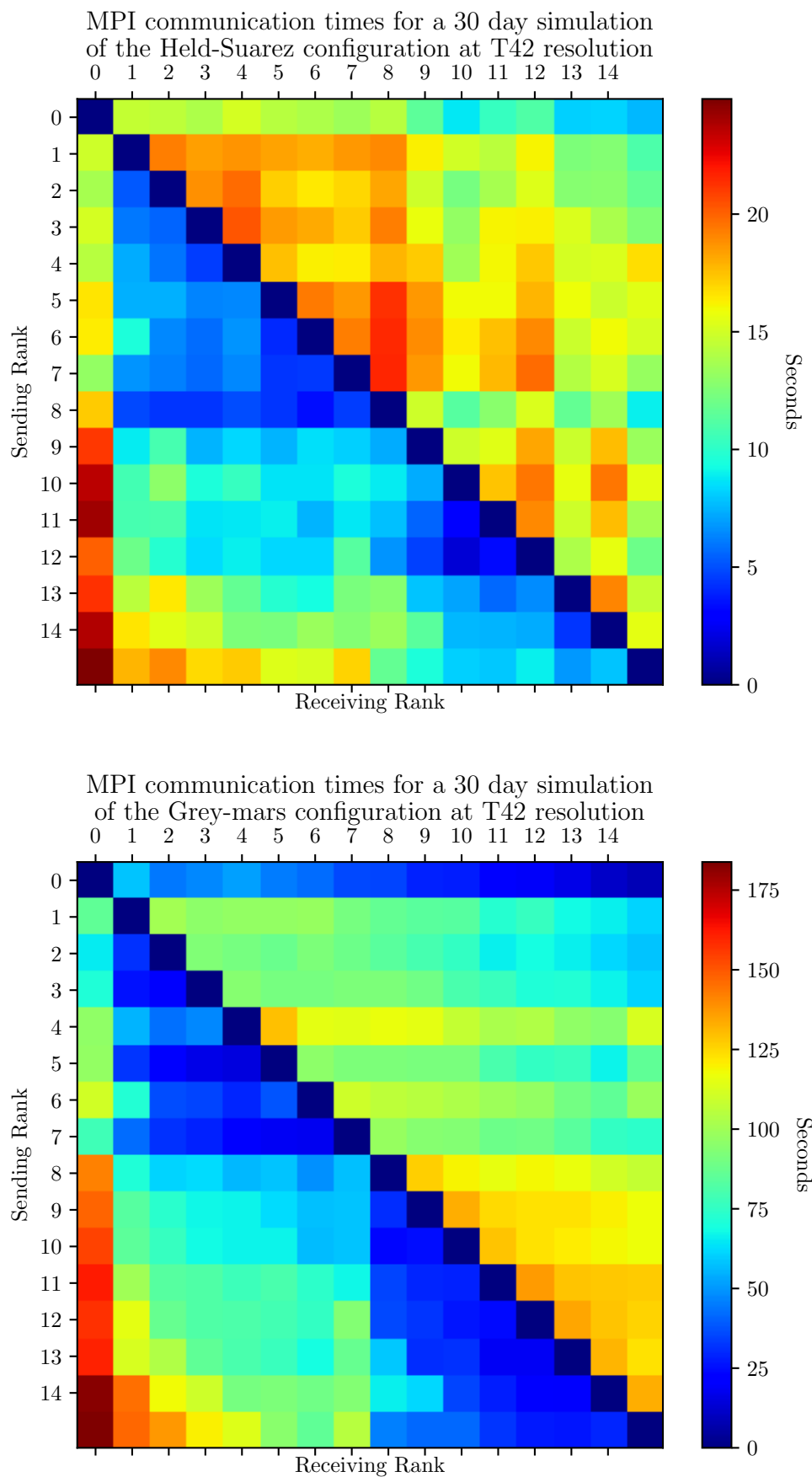


Figure 7.7: Communication matrices for both the Held-Suarez and Grey-mars model configurations when run at a resolution of T42. Communication time has been measured as the sum of all time spent inside the MPI library

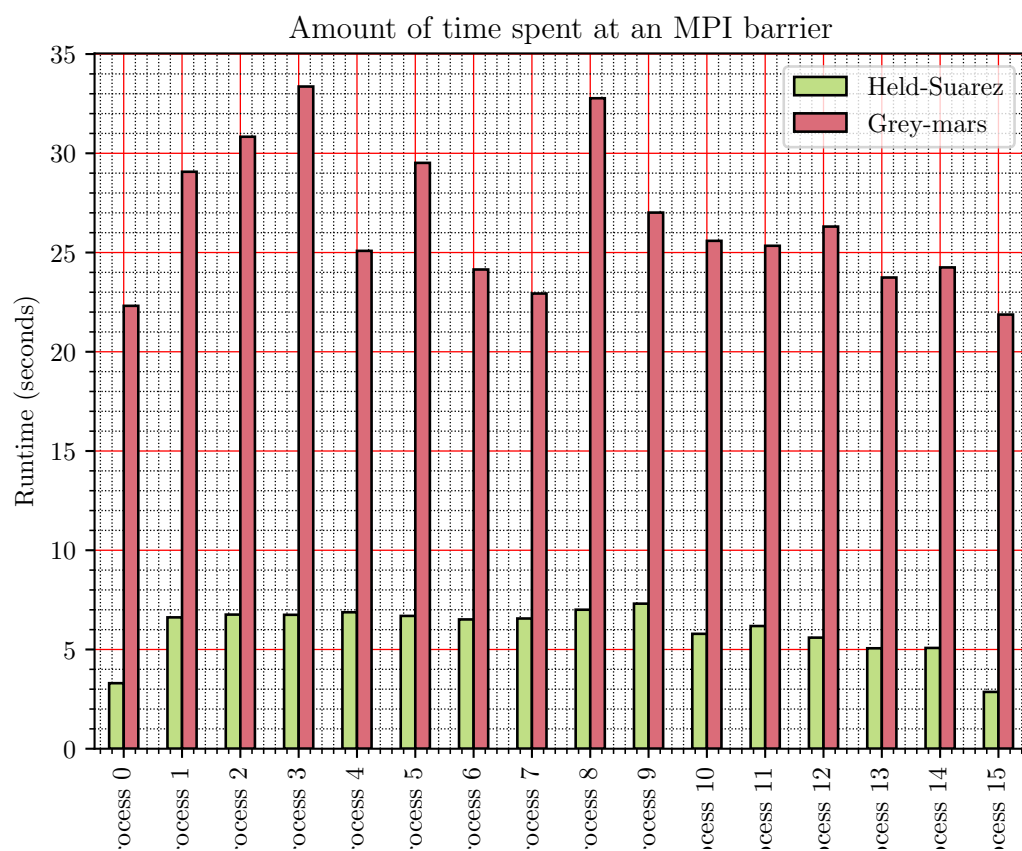


Figure 7.8: Sum of total time spent at an MPI barrier for both the Held-Suarez and Grey-mars model configurations

Table 7.2: TODO

File	Line	Performance Issue
fft99.F90	1081	Inefficient memory access patterns
fft99.F90	1049	Inefficient memory access patterns

### 7.4.1 Vector Instructions

Intel advisor was run for both the Grey-Mars and Held-Suarez case. Running the Grey-Mars case, we find that there are five top time-consuming loops that are performing scalar operations, when vector operations may be able to be used.

Additionally, many of the vector instructions that are being used are only using SSE. Why are there not many AVX2/512 operations being performed? Can we force/coax the use of these wider registers to improve performance?

Inefficient memory access patterns may result in significant vector code execution slowdown or as in this case, block automatic vectorisation by the compiler.

The top five slowest functions.

At `spherical_fourier.F90:312` and `spherical_fourier.F90:315`, there is a similar problem due the nature of how the DFT to spherical harmonics transform works. The transform splits data into even and odd

To do this, it uses two fast Fourier transforms to convert between states. The first being `trans_spherical_to_fourier` and the second being `trans_fourier_to_spherical`. This is the slowest part of the code, lasting for xxx% of the runtime.

`trans_spherical_to_grid` and `trans_grid_to_spherical` both call the `trans_fourier_to_spherical` subroutines that take ages to compute. This means that it would be best to swap out `trans_spherical_to_grid` for something else...

TODO: find a library that does what we want. Will need to check the outputs from both the FMS code, and the new library to make sure that they are the same.

There are some parallels between `fftw3` and the implementation in `Isca`. `fftw3` has two subroutines called `fftw_plan_many_dft_r2c` and `fftw_plan_many_dft_c2r`, that looks like they are used to perform many one-dimensional fft's over a two-dimensional array. Additionally, the `n` variable from `fft99` is used in the same way as `L` from `fftw3` when allocating the memory for the complex type array in the Fourier domain ( $n/2+1$  or  $L/2+1$ ). This means that these arrays are probably used in the same way.

To ensure that the results are the same, need to compare the results of the regular FFT to the results obtained using `fftw`. Sure that this will reveal the root of the problem.

Tested the results of the FFT by using a number of test scenarios using `fft991` and `fftw`. The results were identical using both methods.

## 7.5 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.

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

1 namelist_unit = open_namelist_file()
2 read (namelist_unit, transforms_nml, iostat=io)
3 close_file(namelist_unit)
4 ierr = check_nml_error(io, 'transforms_nml')
```

## 7.6 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 and Critical Analysis**

## CHAPTER 8

---

### Reflection and Critical Analysis

---

This project aimed to present

- 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.

- [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] 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.
- [24] 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.
- [25] 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.
- [26] 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.
- [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] M. Frigo and S. G. Johnson, “The design and implementation of fftw3,” *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.



- [36] M. Frigo, S. G. Johnson, *et al.*, “Fftw for version 3.3.8,” May 2018.
- [37] 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.
- [38] 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

```
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

```
1 #define MPP_DEFAULT_VALUE_ 0
```

If MPP\_TYPE\_ is of type complex, then

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

#### A.2.2 Error 2

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

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

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

becomes

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

# APPENDIX B

## Code listings

### B.1 Grid to Fourier subroutine

```
1 subroutine grid_to_fourier_double_2d_fftw(num, leng, lenc, grid, fourier)
2
3 integer(kind=4),          intent(in)      :: num
4 integer(kind=4),          intent(in)      :: leng
5 real(C_DOUBLE),           intent(in)      :: grid(leng, num)
6 complex(C_DOUBLE_COMPLEX), intent(out)    :: fourier(lenc, num)
7 real                      :: fact
8 integer                   :: i, j
9
10 fact = 1.0 / (leng - 1)
11
12 do j = 1, num
13   do i = 1, leng - 1
14     real_input(i) = grid(i,j)
15   enddo
16
17   call dfftw_execute_dft_r2c(real_input_pointer, real_input, complex_output)
18
19   do i = 1, lenc
20     fourier(i, j) = complex_output(i) * fact
21   enddo
22 enddo
23 return
24 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

```
1 subroutine time_fft()
2 use fft_mod
3
4 real(kind=8), allocatable :: ain(:,:), aout(:,:)
5 complex(kind=8), allocatable :: four(:,:)
6 integer :: i, j, m, n, k, h, iter, lot
7 integer :: ntrans(3) = (/ 128, 256, 512 /)
8 integer :: lots(3) = (/ 64, 128, 256 /)
9 real :: start_time = 0, stop_time = 0, mean_time_iter = 0, mean_time_full
10      = 0, append_time = 0, time_3d_start = 0, time_3d_stop = 0
11
12 iter = 100
13
14 ! test multiple transform lengths
15 do m = 1, 3
16
17   ! set up input data
18   n = ntrans(m)
19   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

## Job submission scripts

### C.1 BCP3 (PBS)

```
1 #!/bin/sh
2
3 #PBS -n held_suarez_benchmarking
4 #PBS -V # export all environment variables to the batch job.
5 #PBS -d . # set working directory to .
6 #PBS -q long # submit to the long queue
7 #PBS -l nodes=1:ppn=16
8 #PBS -l walltime=72:00:00 # Maximum wall time for the
9 #PBS -m e -M qv18258@bristol.ac.uk
10
11 source activate isca_env
12 python $BENCHMARK_ISCA/src/main.py -codebase grey_mars -mincores 4 -maxcores 4
   -r T21 -r T42 -r T85
```

### C.3 BluePebble (PBS Pro)

```
1 #!/bin/sh
2 #PBS -l select=1:ncpus=28:mem=20GB
3 #PBS -l walltime=72:00:00
4
5 module load tools/git/2.22.0
6 source activate isca_env
7 python $BENCHMARK_ISCA/src/main.py -mincores 16 -maxcores 16 -r T21 -r T42 -
   codebase grey_mars -fc kind_4
```

### C.5 Isambard (PBS Pro)

```
1 #!/bin/sh
2
3 #PBS -q arm
4 #PBS -l select=1:ncpus=64
5 #PBS -l walltime=23:00:00
6 #PBS -M =qv18258@bristol.ac.uk
7
8 source ~/isca_env/bin/activate
9 python $BENCHMARK_ISCA/src/main.py -mincores 4 -maxcores 4 -r T21 -r T42 -r T85
   -codebase held_suarez -fc cray_temp
```

### C.7 BCP4 (SLURM)

```
1 #!/bin/bash
2
3 #SBATCH --job-name=benchmark_held_suarez_two_cores
4 #SBATCH --partition=cpu
5 #SBATCH --time=4-00:00:00
6 #SBATCH --nodes=1
7 #SBATCH --ntasks-per-node=24
8 #number of cpus (cores) per task (process)
```

```
9 #SBATCH --cpus-per-task=1
10 #SBATCH --output=held_suarez_two_cores_%j.o
11 #SBATCH --mail-type=ALL
12 #SBATCH --mail-user=qv18258@bristol.ac.uk
13
14 echo Running on host `hostname`
15 echo Time is `date`
16 echo Directory is `pwd`
17
18 module purge
19 source $HOME/.bashrc
20 source $GFDL_BASE/src/extra/env/bristol-bc4
21 source activate isca_env
22
23 $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
```