University of Reading

Parallel Data Analysis for Atmospheric Science

PhD in Atmosphere, Oceans and Climate

Department of Meteorology

Matthew Jones

April 2018

Declaration

I confirm that this is my own work and the use of all material from other sources has been properly and fully acknowledged.

- Matthew Jones

Abstract

Data sizes are growing in atmospheric science, as climate models increase to higher resolutions to improve the representation of atmospheric phenomena, and larger numbers of ensemble members are used so as to better capture the variability in the atmosphere. New methods need to be developed to handle the increasing size of data – traditional analysis scripts often inefficiently read and process data, leading to excessive analysis times. Research into large data analysis often focuses on providing solutions in the form of software, or hardware, rather than providing quantitative results on what factors can reduce performance in an application. This thesis quantitatively investigates these factors in the software-hardware stack, in order to make decisions how to handle large data sizes during application development and data management. This is done in the context of an atmospheric science workflow in a high-performance computing environment.

A major bottleneck in analysis in atmospheric science is reading data. Two of the primary factors which are commonly known to affect the read time are the read pattern, and the read size. These factors are found in this work to reduce the read rate by up to 10-50 times for poor combinations. Other factors which could affect the read rate for atmospheric analysis include: the programming language, the libraries used, and the file layout.

NetCDF4 is one of the most commonly used data formats in atmospheric science, and the Python library netCDF4-python is one of the main interfaces used. As part of the NetCDF4 file format, there are options for chunking (multidimensional tiling), and inbuilt compression, which can be used to improve read and write performance from the files. It was found that at peak performance the netCDF4-python library performs 40% worse than the underlying C NetCDF4 library. With respect to chunking and compression, poor combinations of chunking, and inbuilt compression, were found to reduce the performance by over 100 times.

One solution to reduced performance, or a way to reduce analysis times on large datasets, is to run applications in parallel. It is important to understand how, on a particular platform, application relevant parallel reads will scale in order design an efficient application. The parallel scaling of the JASMIN super-data cluster was analysed. The investigation methodology, and conclusions from the investigation can be applied to other platforms.

A case study was used to apply the results from this work in a real atmospheric science workflow – a space-time spectral analysis technique. It confirmed that these results do indeed apply to real workflows.

Acknowledgements

A great deal of thanks is due to my supervisors: Bryan Lawrence, Jon Blower, and Annette Osprey. You have my deepest gratitude.

There are many other people who have supported me throughout this PhD, and without their support it would not have been possible. You know who you are.

Contents

Chapter 1

Introduction

The volume of data is growing in atmospheric science, as climate models increase to higher resolutions to improve the representation of atmospheric phenomena, and larger numbers of ensemble members are used so as to better capture the variability in the atmosphere. New methods need to be developed to handle the increasing size of data – traditional analysis scripts often inefficiently read and process data, leading to excessive analysis times. In order to make decisions on how to handle the large data sizes during application development and data management, quantitative analysis of the factors which affect data analysis throughput in atmospheric science in a high-performance computing environment are needed.

In general, there are four major stages to a workflow when analysing data in atmospheric science:

- First is accessing data. For smaller or medium sized datasets, it is often possible to download data to a local machine.
- Second is data exploration, important to assess suitability of a dataset for the investigation, and to plan the investigation. Usually this is only on a subset of the data in which, even with large datasets, the volume of the data often is not problematic.
- Third is the development of the analysis application. This involves testing, which is often on a subset of the dataset, and will involve many iterations in development, meaning a large number of reads. This stage could also be skipped if existing applications exist from previous investigations.
- Implementing the analysis application on the whole dataset and gathering results is the final stage. This will often include multiple different implementations, for example: mul-

tiple ensemble members, multiple time periods, or even multiple datasets (e.g. climate simulation output, and reanalysis data).

With very large datasets, transferring data to the user is not always an efficient option. Instead, running applications local to a data storage and processing facility is a more efficient solution. One such example of this is the JASMIN (Joint Analysis System Meeting Infrastructure Needs) super-data cluster.

Data exploration is inherently less data intensive than the final application, due to analysing only a subset of the data. However, any increase in duration due to inefficient reads or writes, is compounded with larger data sets.

Similarly to data exploration, because application development only requires a subset of data, it is not affected by inefficient reads and writes to the same extent as the final application. However, development and testing requires many repeated calculations, and read and write operations, meaning that any reduction in performance is compounded by larger datasets. With regards to the potential for existing applications being used, the processing algorithms may not be suitable for use with very large data, or could be significantly improved to provide acceptable run times. Bespoke applications or libraries, designed to analyse very large datasets are a good alternative to writing applications. However, analysis can often be too specific, or specialised for this approach to be valid, therefore, an application needs to be developed. Without prior quantitative knowledge on the factors which can affect analysis applications, providing acceptable performance can become a very time consuming task, and development can become constrained by trial and error. This can significantly lengthen the development process, taking up time which could otherwise be used for further analysis, or results interpretation.

Research into large data analysis generally focuses on providing a solution in the form of a library, software middleware, or hardware to improve performance. The motivation for these studies is that data analysis on large data is a problem and will become more of a problem, generally with little quantitative evidence. For those with experience handling and analysing large data, the problem is obvious, as are the implications for analysis as we progress into the exascale. The aforementioned prior quantitative knowledge on the factors which affect analysis is a gap in the research, and the focus of the work presented in this thesis. This knowledge is often gathered through experience, and trial and error, so the usefulness for this study is clear – circumventing pitfalls due to combinations of effects which provide very poor

performance, and eliminating a large proportion of trial and error, will reduce development time and enable more science to be done.

In atmospheric science, little work has gone into understanding, quantitatively, how the software-hardware stack can affect analysis workflows. Weather and climate models are complex and much work has gone into what can affect their performance (Section 2.1) – the performance of analysis applications could be investigated in a similar way.

Reading and writing data is a major bottleneck for analysing data in atmospheric science (Méndez et al., 2013; Gao et al., 2011). This problem is only going to become compounded as data gets larger, due to increasing temporal and spatial resolutions needed to more accurately represent the physics and dynamics in models; and increased ensemble sizes, needed to more accurately capture uncertainty in the chaotic systems (Balaji, 2015).

Traditionally, analysis of atmospheric science data has been done in serial, although that is now changing. Serial data analysis on very large datasets can be problematic for a number of reasons. Firstly, large datasets mean that the random access memory (RAM) may not be large enough to hold the data during calculations. Secondly, analysis can take a long time to complete, slowing down the workflows and meaning some tasks can become impractical. Many different factors can affect how long an analysis program takes to run, such as:

- the time to read or write data, which is heavily affected by the data size,
- the analysis algorithm,
- the organisation of the data,
- and the hardware and software used.

Running analysis in parallel can alleviate problems with long run times and running out of RAM. When using multiple nodes in parallel, the RAM from each node can be used, increasing the amount of RAM proportional to the number of nodes. The I/O and compute can also be split amongst the nodes, decreasing the total time for the parallel program to run. There are complications to consider when designing parallel programs, any of which can contribute to an efficient or inefficient program. These include:

- the parallel decomposition of the problem,
- the dimensional dependencies of the algorithm,

- the load balance for the I/O, computations, and RAM,
- the platform architecture and settings,
- and the inter-processor communication needed for the algorithm.

These factors can make writing an efficient parallel program difficult for many analysis tasks.

Large datasets in atmospheric science are often stored in an archive, or filesystem of a data centre. This data is generally produced by some form of model or simulation, so is written once and read from many times, by many different users. Typically, reading data dominates other factors for performance of applications (Childs et al., 2005). Factors which can specifically affect the read rate for an algorithm include:

- the layout and format of the data (including multidimensional tiling, and compression),
- the read pattern of the algorithm,
- and the language and libraries used.

The combination of these factors produces a large, complex, and interdependent problem. Although these factors are interdependent, it is necessary to understand the individual effects on performance before their combination can be understood. This thesis investigates the contributing factors which can affect the read rate for serial and parallel atmospheric science data analysis scripts. The work approaches the effects on the read rate from the point of view of a user of a high performance computing (HPC) platform, rather than analysing the peak performance of a system. The peak performance is not always useful to know when designing analysis programs, because the system performance is often quoted as a system wide performance, and the peak theoretical performance of a platform is often not representative of a real workflow. For example the theoretical I/O bandwidth may not be achievable. For climate and weather models, Balaji et al. (2017) discussed that, for example, flops (floating point operations per second) is not a useful metric for examining performance of earth-system models, instead they proposed more relevant domain specific metrics, such as simulated years per day. In a similar way, identification and quantification of realistic factors affecting data analysis in atmospheric science would be more relevant to users than system level benchmarks.

There are three investigations in this thesis: 1) the effect of the read pattern, software, and data type on the read rate, 2) the effect the layout of the data has on the read rate, and 3) the

scaling of the parallel read rate for multiple different situations. The final part of the investigation in this thesis is to implement the results from the first three investigations into a case study; affirming whether the results are valid in more realistic situations, and so applicable to the general users of HPC platforms. The JASMIN super-data cluster (Lawrence et al., 2012) is used throughout the thesis, as an exemplar HPC platform for data analysis (other platforms are used in sections of the thesis to provide a broader sample base).

1.1 Thesis structure

Chapter 2 contains the background required for the rest of the thesis, including: parallel computing concepts, factors affecting I/O performance, and discussion of parallel filesystems. The relation of these factors to atmospheric science is discussed, as well as alternative approaches to big data analysis. Chapter 3 contains the investigation of the performance of NetCDF4-python and the underlying libraries that it uses, as well as the effect read sizes and read patterns have on the read rate. Chapter 4 investigates how multidimensional tiling and compression affect the read rate. Chapter 5 investigates parallel scaling when reading NetCDF4 files on a parallel file systems. Chapter 6 combines work from the previous chapters in a case study, to ascertain whether the optimal parameters discovered the previous chapters are applicable to a real-life work flow. Then conclusions are drawn in Chapter 7, with discussion on how the thesis relates to other studies, and a discussion of channels for future work is included. Appendix A provides a more detailed look at the platforms used in the thesis than that in the background chapter. Appendix B describes the tools and libraries used for the testing scripts. Finally, Appendix C contains the main code for the testing scripts used.

1.2 A note on file sizes and read rates

Throughout this thesis, both base 10 (e.g. MB) and base 2 (e.g. MiB) units are used. Table 1.1 shows the size of each unit in bytes and the size difference. Where both are used care has gone into making sure the units are reported correctly. Rates in this thesis are always reported in base 10 units. Base 2 units are used for reporting file sizes where the files created for testing were in in base 2. So that the reported file sizes are easier to read on both graphs and in the text, the base 2 units are retained, as opposed to converting to base 10 and reporting the file sizes rounded to a number of decimal places (this also means the file sizes reported are exact

as opposed to rounded for the sake of reporting in base 10).

Table 1.1: The sizes in bytes of base 10 and base 2 units.

Base 10	Base 2	Difference (B)
1 KB = 10^3 B	1 KiB = 1024 B	24
1 MB = 10^6 B	1 MiB = 1024^2 B	48,576
1 GB = 10^9 B	1 GiB = 1024^3 B	73,741,824
1 TB = 10^{12} B	1 TiB = 1024^4 B	99,511,627,776

Chapter 2

Background

In this chapter, overarching concepts important to this thesis are discussed, along with alternative approaches to big data analysis.

A useful way of analysing the performance of a system is to categorise similar algorithmic patterns to predict performance (Section 2.2). I/O is a major bottleneck in atmospheric science so it is important to understand how to measure I/O performance (Section 2.3), as well as why data size is a problem in atmospheric science (Section 2.4). One way to improve performance of analysis applications in atmospheric science is to run them in parallel. This means that parallelism needs to be understood (Section 2.5), along with what factors affect I/O performance (Section 2.6). In order to understand the results from investigating application performance, the platforms which analysis is performed on need to be understood (Section 2.7). There are many other valid approaches to dealing with atmospheric big data, some of which are outlined in Section 2.8.

2.1 Atmospheric science model performance

Having useful measurements of performance is an important aspect of understanding performance. The performance of climate and weather models is an important area of research, due to not only the computationally intensive nature of the simulations – due to increasing resolution, complexity, and increasing ensemble sizes – but also the importance of the research providing scientific input to global policy decisions; for example the IPCC AR5 report (Stocker, 2014).

Having a useful set of realistic metrics to assess the cost of running a global model is in-

credibly useful. Balaji et al. (2017) propose a set of metrics that are representative of the actual performance of the models. For example, flops are not always a useful metric, whereas the number of simulated years per day is a much more relevant. Developing a similar set of performance metrics for data analysis would allow better profiling of data analysis applications. The first stage of this is to identify the areas where performance is affected.

2.2 Categorisation as a method to analyse performance

Characterisation of patterns in data can be used to gain an understanding of a system. Likewise, characterisation of different patterns in data analysis can help understand those applications. Similar algorithms can then be grouped by their characteristics, and then analysed for the best way to implement the applications for use with big data. Future applications which fit into the same groups could be solved in a similar way, making it easier to efficiently write those programs.

The Berkeley Dwarves provide a way of looking at parallel HPC applications on multicore processors (Asanovic et al., 2006). With a similar aim, but with a focus on data analysis, the Big Data Ogres were proposed (Jha et al., 2014). The Ogres provide four key facets to aid the understanding of big data analysis. These are:

1. the problem architecture facet,
2. the execution features facet,
3. the data source and style facet,
4. and the processing facet.

The problem architecture describes the way in which the problem has to be solved, for instance in a pleasingly (or ebarrassingly) parallel way, using shared memory, or machine learning (these will be discussed in Section 2.5.4). The execution features describe limiting factors, such as the I/O balance with the compute, and the limitations imposed by big data (discussed in Section 2.6). The data source and style facet describes the data itself, such as whether it is binary file format or a database, or the source of the data. The final facet, processing, describes the specifics of the algorithm, such as search and query, graph algorithms, and local and global machine learning.

The Ogre's facets can be more specifically applied to work in atmospheric science. Examples of how each facet could describe problems in atmospheric science:

1. Problem architecture, for example, could be a pleasingly parallel problem involved with retrieval over images, or could be a machine learning algorithm clustering atmospheric phenomena.

2. The execution features depend heavily on the algorithm, especially the balance between I/O and computation time (I/O-CPU balance), the data size, and the number of different fields required for the analysis.

3. The data source and style would typically be binary data for climate analysis or weather forecasting, but could be other data types, such as CSV files, images, or a proprietary data type.

4. The processing algorithm describes the specifics of the algorithm, for example, the specific clustering, or tracking algorithm.

In order for categorisation to be useful, an understanding and quantification is required for the factors which affect the performance. With analysis of large data sets in atmospheric science, typically I/O is a major bottleneck (Balaji, 2015).

2.3 Measuring I/O performance

Note that, throughout this thesis only reads will be discussed because data analysis on large data sets in atmospheric science is generally executed on data stored in an archive, which was produced by a numerical model. This means that reads are significantly more dominant than writes in most cases. Read performance can be quantified in different ways. Those discussed here are:

- read rate,
- bandwidth,
- I/O operations per second (IOPS),
- and throughput.

The read rate measures the speed of the operation and is defined as the amount of data transferred per second, for example if 600 MB were transferred in one minute the read rate would be 10 MB/s. The bandwidth is the maximum potential performance limited by the hardware being used. The minimum bandwidth in a system will constrain the total bandwidth available, for example if a filesystem has a bandwidth of 10 GB/s, but the local area network has a bandwidth of 1 GB/s, the network would limit the maximum rate. Note that this is a theoretical limit imposed by the hardware, which may not be achievable in reality. IOPS (I/O operations per second) describes the rate of which I/O operations can occur, including both reads and writes. This is also a theoretical limit, and hardware often quotes maximums for reads and writes, for sequential (where the next I/O operation is the neighbouring piece of data) and random (where the next I/O operation is not the neighbouring piece of data) – the random IOPS is lower than the sequential IOPS. Finally, the throughput is defined as the IOPS multiplied by the size of the I/O operation, meaning to improve performance either the read size, or the IOPS needs to increase.

Benchmarking applications can be used to assess the I/O performance of a system, and this can be used to help understand how an analysis application would perform. Four different applications will be discussed here, however, none of these quantitatively assess where in the software-hardware stack performance is being affected, only the overall results – this provides motivation for the work in this thesis.

On Linux based systems, a simple way to profile I/O is using the `dd` utility[1]. `dd` measures the read and write times and uses arguments to vary parameters, outputting the wall time (real-world time from start to finish for the application) and the respective read or write rate. It allows the size of the buffer to be varied. This controls the size of the individual reads and writes, for example a 1 GB read could have a buffer size of 1 MB, meaning 1000 reads were executed. The advantage and disadvantage of `dd` is its simplicity, as it only enables one-dimensional read and writes of binary data, and has no inbuilt way to assess parallel reads.

IOR[2] is a parallel benchmarking application, which allows much more control than `dd`, giving options including: the file type (including NetCDF, and HDF – see Section 2.4), the number of parallel tasks, the buffer size, and the number of files per task. This allows the performance on parallel file systems, and HPC clusters to be assessed, and so predict how

[1] https://linux.die.net/man/1/dd
[2] https://github.com/LLNL/ior/blob/master/doc/USER_GUIDE

parallel applications would perform. This is still a relatively simple benchmark however, and so may not be truly representative of the performance for scientific applications. An example of IOR results from JASMIN are shown in Figure 2.1.

IOR can be used to synthetically and accurately model applications (Shan et al., 2008). Theoretically, the arguments of IOR can be used to replicate the I/O of an application to measure how it would perform on a system without having to run the application, saving time. However, the opinion of some studies is that it can be difficult to relate the IOR arguments to real applications (Borrill et al., 2007; Lawrence, 2014).

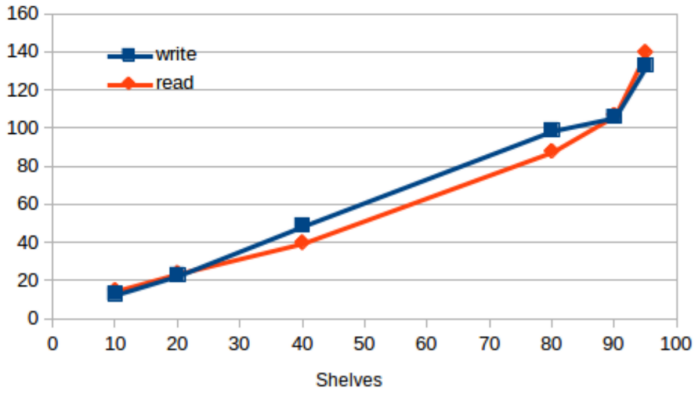


Figure 2.1: I/O scaling results from JASMIN using IOR measuring total bandwidth (y-axis, in GB/s) scaling with number of shelves (See Appendix A). Figure from Lawrence (2014)

Benchio[3] was designed to give a more flexible benchmark which is more applicable to scientific analysis than IOR (Lawrence et al., 2017). Firstly, it provides more control of the parallel decomposition to more accurately represent read patterns in applications. Secondly, it is less opaque than IOR, enabling more conclusions to be drawn about what is affecting the I/O rate. Finally, benchio is able to use three-dimensional data, meaning it is much more relevant for scientific applications.

MADbench2 (Borrill et al., 2009) is an application derived benchmark, developed from analysis of the cosmic microwave background (electromagnetic radiation which can provide information on the early universe). It provides more information than the previous benchmarks discussed above, including the time spent in CPU based calculations, and the I/O time. It allows adjustment of the scale of the benchmark, parallelisation, and system specific settings. The aim of this benchmark was to measure and compare the performance of systems rather than predict quantitatively how other applications would perform.

[3]https://github.com/EPCCed/benchio

Instead of measuring the performance of applications directly, statistical modelling can be used to predict the performance of the applications by identifying important elements of the access path. Schmid and Kunkel (2016) showed that artificial neural networks can more accurately predict performance than a linear model, where linear models are not a valid method of performance analysis.

2.4 Atmospheric science data

For a given analysis problem, the data size affects the number of floating point and I/O operations, each of which can increase the run time. Higher fidelity models and observations, and larger ensemble sizes produce larger amounts of data, all of which needs to be analysed. For example CMIP6 is expected to produce 10-20PB of data (Cinquini et al., 2014), compared to CMIP5 which produced around 3 PB (Liu et al., 2015). As the data size becomes larger, the analysis problem becomes more difficult.

Higher spatial resolutions provide a number of benefits for atmospheric models. Firstly, high horizontal and vertical resolutions mean that atmospheric phenomena and topography are more accurately represented (demonstrated in Figure 2.2), improving the accuracy of models. Secondly, with higher resolutions less parametrisation (representing phenomena with parameters when they are not resolvable) is needed, potentially improving the accuracy of models. For instance, at higher resolutions (around 1km) convection is resolvable.

Higher temporal resolutions are often required to ensure stability with higher resolution models. Also, higher temporal resolutions can resolve phenomena more accurately. Quite often, not all timesteps are outputted from the model – either timesteps are skipped when saving the data, or an average is calculated over a time period, and the result saved. Higher temporal resolutions are needed when analysing models if fast moving phenomena are being studied, such as gravity waves.

Ensembles are used to capture variability in chaotic systems. Each ensemble member has slightly different initial conditions, so each will have a different representation of the atmosphere. The variability in the atmosphere is more accurately represented by having a larger number of ensemble members, but this also means there is more data to analyse.

The volume of data in atmospheric science is a problem since I/O is a major bottleneck (Méndez et al., 2013; Gao et al., 2011). With traditional, serial analysis programs which are

often either made for a very specific task, or are legacy programs which were used for analysis on smaller data sets, the program may not complete the analysis in a timely manner. The latter is an obvious problem, but the former is a problem from a productivity standpoint – having to wait a long time for analysis to finish degrades productivity. Analysis programs taking less time, and producing results faster also means that more science can be done.

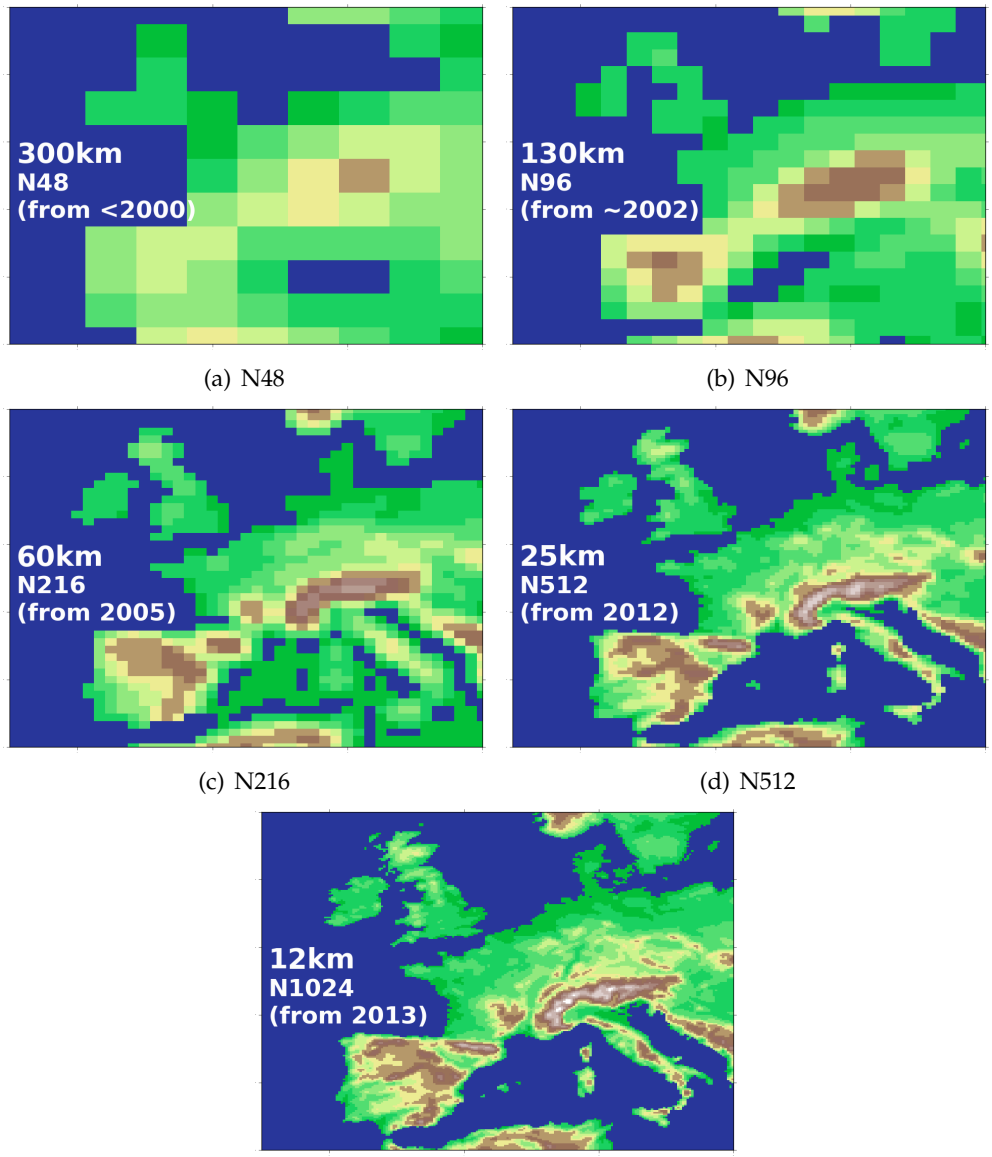
Before discussing an example of why big data is a problem in atmospheric science, the nomenclature of the resolution of atmospheric models needs to be described. Figure 2.2 shows an example of the nomenclature used with climate models to describe the resolution (this nomenclature is used for the MetOffice Unified Model, other models may use different nomenclature). Nx describes the horizontal resolution according to the formula $[2x, 3x/2]$, for instance N216 has 532 longitude points, and 324 latitude points, N512 has 1024 longitude points and 768 latitude points, and N2048 has 4096 longitude points and 3072 latitude points – the number of points for all the resolutions in Figure 2.2 is shown in Table 2.1 (plus N2048). The number of levels in the model is then describe by an L followed by a number, i.e. L80 has 80 vertical levels, and L180 has 180 vertical levels. This is combined to give a spatial description of the model, e.g. N512L180 would have dimensions (in height, latitude, longitude order) of [180,768,1024] – this order is used in the thesis to describe the slowest to fastest varying dimension.[4]

Table 2.1: Number of latitude and longitude grid points for each resolution.

Resolution	Longitude	Latitude	Size in MB
N48	96	72	0.055
N96	192	148	0.22
N216	512	324	1.33
N512	1024	768	6.3
N1024	2048	1536	25
N2048	4096	3072	100

As an example of why data volume is a problem for atmospheric analysis, consider the following when analysing stored data (most analysis in atmospheric science is done post runtime, when the data is stored in an archive, or analysis platform). In the first case consider results from an atmospheric simulation run for 1 year with a 6 hourly timestep and a middling resolution of N216L80, meaning the dimensions are [1440x80x324x512]. This means the dataset has a size 152 GB with each element an 8 byte floating point numbers. At peak read rate of 1GB/s (the approximate theoretical maximum for a serial process on JASMIN (Lawrence et al.,

[4]Longitude values for the first latitude value and height value are contiguous on disk, followed by the next set of longitude values for the next latitude value and first height value, as shown in Figure 2.7.



(a) N48 (b) N96

(c) N216 (d) N512

(e) N1024

Figure 2.2: Representation of Europe in the different resolution models as part of a global model. Blue denotes sea, and the other colours denote the elevation of land.

2012)), this read would take around 152 seconds. At a realistic inefficient read rate of approximately 10MB/s (from the investigation into read performance in Chapter 3), this would increase to 253 minutes – a significant increase but not unreasonable.

Now consider the same analysis on high resolution N512L180 hourly data, with dimensions of [8640x180x768x1024] – 9.8TB in size. At the 1GB/s read rate it would take 163 minutes to read the data, however at 10MB/s it would take 11.3 days to read the data - obviously impractical. At even higher resolution this problem would be compounded. As well as resolution, when analysing ensembles this must be multiplied by the number of ensemble members.

There are a number of ways to potentially improve the read rate: identifying where there

are factors influencing the read rate (Section 2.6), implementing the analysis in parallel (Section 2.7), along with other techniques discussed in Section 2.8.

One of the most widely used data formats in atmospheric science is the Network Common Data Format (NetCDF)[5]. Figure 2.3 shows an example of the proportion of data stored in the JASMIN workspaces. NetCDF4 is built on HDF5 (Hierarchical Data Format version 5). (Note, NetCDF3 is an older format of NetCDF not built on HDF5, and does not allow chunking (see Section 2.4.1) so is not investigated in this thesis, for a comparison between NetCDF3 and 4 performance see Welch et al. (2010)).

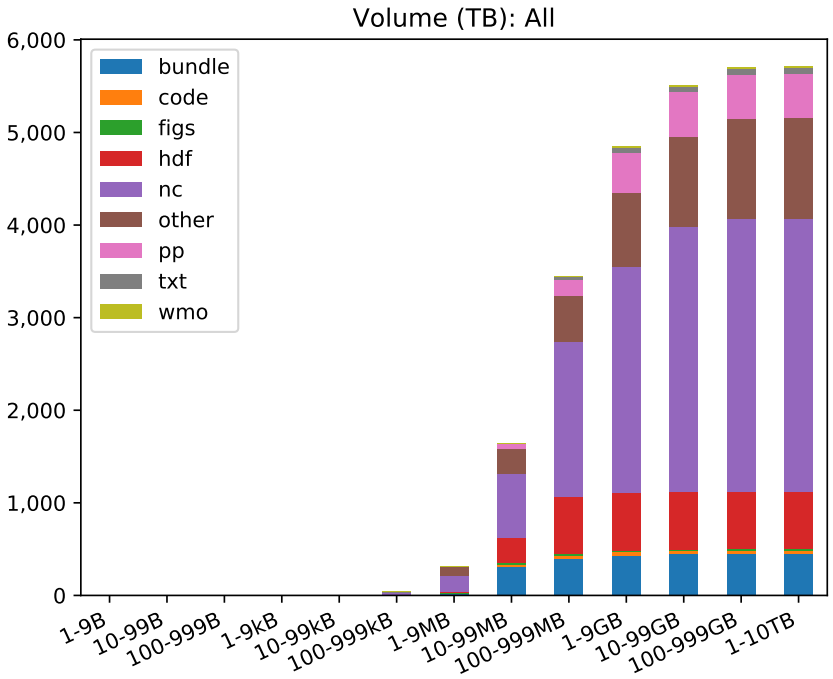


Figure 2.3: Data stored at JASMIN on the group work spaces (June 2017), split into the file size(x-axis), total volume (y-axis), and the composition from different file types. (B.Lawrence, personal communication) Note, the histograms are stacked.

2.4.1 HDF5

HDF5 is a versatile, portable data format which can be used to store large data sets, and it allows the use of chunking and compression[6]. Chunking (an implementation of multidimensional tiling) allows files to be reorganised on disk to improve read rates for different access patterns (Welch et al., 2010) – discussed in Section 2.6.3. Chunking can improve the performance of partial I/O (when only sections of the file are being read) (Rew, 2013). This is be-

[5]https://www.unidata.ucar.edu/software/netcdf/
[6]https://support.hdfgroup.org/HDF5/

cause, in order for partial I/O to be most efficient the data selected must be contiguous on disk, which chunking allows. The chunks can be compressed to reduce the size of data on disk – compression is discussed in Section 2.6.4. Chunking and compression can have a large effect on the read rate (Bartz et al., 2015).

2.4.2 NetCDF4

NetCDF4 is popular in the scientific community because it is a self describing, platform independent binary file format. NetCDF4 allows the use of zlib compression (Lee et al., 2008) which allows sections of the file to be compressed and accessed individually (in this case the sections are chunks) – compression is discussed in more detail in Section 2.6.4. zlib compression provides a good balance between compression ratio and compression speed (Liu et al., 2015) without losing accuracy in the data (lossless compression). The data size is reduced so less data is read from disk, meaning the time to read the file could decrease (Miller, 2015), but obviously this data will then need to be uncompressed on the processing node, meaning there is a balance between increased read rate due to fewer bytes being read, and the extra computation required to uncompress the data. The compression of NetCDF4 requires the data to be chunked so the interplay between chunking and compression could have a significant effect on the read rate. The author is not aware of any literature which looks at how the interplay between compression and chunking affects the read rate.

2.5 Parallelisation

Running an algorithm in parallel means splitting an algorithm into sections and running these sections on multiple processors. If these section are run simultaneously they are concurrent. This may be done because of the volume of work (calculations or I/O) in the algorithm, or because a single node does not having enough memory to process the data required (utilising multiple nodes can increase the effective memory available for the algorithm). Parallelising an algorithm into multiple tasks can reduce the time for an algorithm to complete. Parallelisation on multiple nodes also means utilising more network interface cards (NICs), increasing the available I/O bandwidth; so that if the I/O is run in parallel, the I/O time can be reduced. This last point is important as it is well known that I/O is a major bottleneck for analysis in atmospheric science, serial data processing is no longer an optimum solution (Balaji, 2015),

and concurrent reads are crucial for high speed reads (Lofstead et al., 2011).

2.5.1 Strong and weak parallel scaling

Strong and weak scaling is an important concept for analysing results with parallel application performance. Strong scaling is where the size of the problem stays fixed, but the number of concurrent tasks is increased. Weak scaling is keeping the size of the problem per task the same, but the total size of the problem increases with the number of tasks. An example of the difference can be demonstrated when reading data to test the I/O scalability of a system. In the strong scaling case a single task could read from a 1 GB file, then two tasks would either read half the file each, or two 500 MB files – keeping the total size the same. In the weak scaling case, with two tasks, each would read from a 1 GB file – the total size of the problem has doubled.

2.5.2 Amdahl's Law and Gustafson's Law

In its simplest sense, parallel computing is purely using more than one processor to complete a task. However, the proportion of a workflow which must be executed in series limits the available speedup (the parallel algorithm's time to complete divided by serial algorithm's time to complete). With a fixed problem size this is described by Amdahl's law, which gives an upper, theoretical, limit on how much a task can be sped up, depending on how much must be executed in serial. The maximum speedup of a task is described by:

$$R(P)/R(1) = \frac{1}{S + (1 - S)/P} \tag{2.1}$$

where $R(P)$ is the rate of completion of a task using P processors and S is the fraction of the time spent doing tasks in serial (Amdahl, 1967). In an idealised situation using an infinite number of processors, the maximum speedup is maximum $1/S$, so for example for a process in which 10% must be done in serial, the maximum speed up is 10 times.

Gustafson's Law does not assume a fixed problem size (strong scaling), and instead assumes that as the computational power of a system increases, more work will be done (weak scaling) (Gustafson, 1988). This gives no limit to the possible speedup when the problem size increases.

2.5.3 Flynn's Taxonomy

Flynn's taxonomy is a classification scheme for central processing units (CPUs). There are four classifications, which are described by the number of instruction or data streams are being used, this is shown in Table 2.2. SISD (single instruction multiple data) is where a single processing unit (PU) receives a single instruction stream from memory and a single data stream, giving a single operation at a time; an example of this would be old single core personal computers (PCs). MISD (multiple instruction single data) describes when multiple instructions streams operate on the same data. MISD is generally used for fault tolerance, where all results from the instruction stream must agree. SIMD (single instruction multiple data) describes a computer in which a single instruction stream works on different streams of data working in parallel to process data faster with no parallelisation of the instruction stream. An example of SIMD would be a graphics processing unit (GPU). MIMD (multiple instruction multiple data) on the other hand, parallelises the instruction stream and the data stream, an example of which would be a modern PC or cluster. (Flynn, 1972)

Table 2.2: The four types of computer architecture according to Flynn's taxonomy.

	Single instruction stream	Multiple instruction streams
Single data stream	SISD	MISD
Multiple data streams	SIMD	MIMD

MIMD can be further divided into SPMD (single program multiple data), and MPMD (multiple program multiple data). These two roughly describe different techniques to parallelise workflows: either as a single program executing on different sections of data, to process the data faster, which could also be described as data parallel or domain decomposition; or multiple different programs where sections of the workflow are split between processors, this could also be called task parallel (Blank and Nickolls, 1992). Typically, data parallelisation is the dominant form of parallelisation on modern computers (MIMD type) (Hager and Wellein, 2010). Typically, most applications, however, will use a combination of data and task parallelism.

2.5.4 Parallel computing models

During the discussion on Big Data Ogres in Fox et al. (2014)(the key facets were discussed in Section 2.2), there were five computing models described (the problem architecture facet). It is important to discuss the relevance of these computing models for the work in this thesis. These are:

- classic MapReduce,
- iterative Map-Collective,
- iterative Map-communicative,
- pleasingly (or embarrassingly) parallel,
- and shared memory .

The MapReduce and pleasingly parallel models are shown in Figure 2.4. All of these in terms of Flynn's taxonomy would, for most workflows, be classed as SPMD, or data parallel.

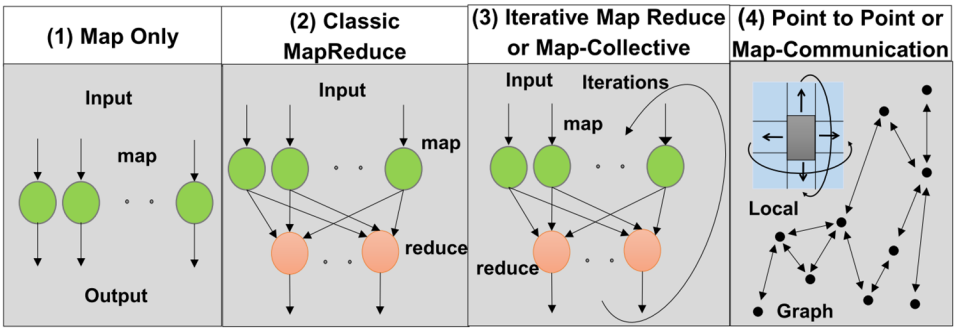


Figure 2.4: Representation of the three families of MapReduce, plus pleasingly parallel (or map only). From Fox et al. (2014) (Figure 2).

The first family of MapReduce is commonly known as classic MapReduce; it was the original version of MapReduce introduced by Google (Dean and Ghemawat, 2008). In this version of MapReduce the input data or input files are split and sent to map functions which then perform some algorithm, which, for example, could be a sort or search algorithm. The results from the map stage are then sent to the reduce stage where a function typically will summarize the results from the map stage, or could merge values to give a smaller set of results. Typically MapReduce is used for analysis tasks such as web based big data, but there has been some work on how to extend this to scientific data (see Section 2.8 for a discussion on some approaches).

The other two families of map reduce are iterative Map-Collective and iterative Map-Communication. The former uses a collective stage to gather data and recast it out for another iteration of the algorithm. A common implementation of this would be in clustering, where data is collected into regions over a number of iterations, becoming more accurate after more iterations of the algorithm. Map-Communication has the communication included as part of

the processing algorithm (before the reduce) and is also iterative. An example of the use of this would be for solving graph algorithms.

Pleasingly (or embarrassingly) parallelism is in many ways similar to MapReduce, but with no reduce stage. Like the map stage of MapReduce, the input data is split between processors and these processors implement an algorithm. There is no communication between processors.

Shared memory is a technique that allows multiple processors to access a large pool of memory, negating the need for explicit communication between the processors, because they can all access data in the memory.

2.6 Factors affecting I/O performance

Many factors can affect the read performance of analysis. Figure 2.5 depicts factors which can affect the read performance, and from what area of the software-hardware stack they originate. The I/O time is affected by the size of the data, the software, the bandwidth defined by the hardware, and the access pattern to the data. The size of the data and the system settings are generally not adjustable by the user implementing the analysis – the system options, such as stripe width, are sometimes configurable but often users do not configure them. It is well known that access patterns and data organisation have a large impact on I/O rate and can dominate other costs in analysis (Childs et al., 2005). How the software and hardware, and the tunable parameters chosen either by the user or the system administration, interact can also have a large impact on the I/O rate and therefore affect the workflow.

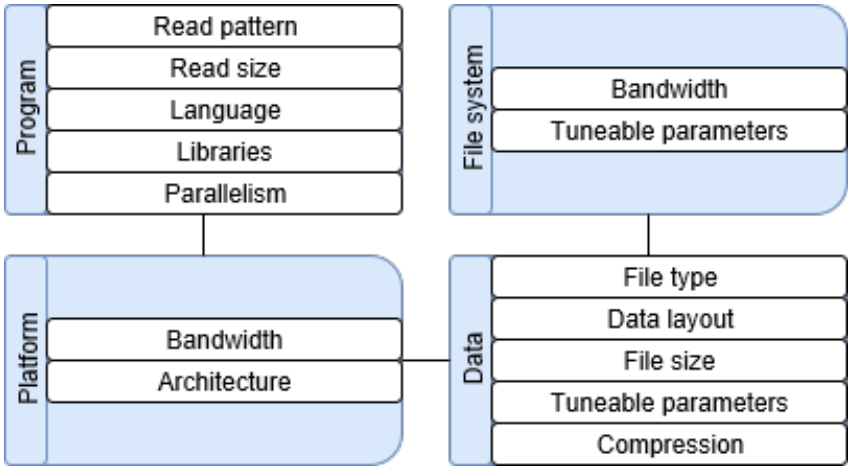


Figure 2.5: Factors which can affect the read rate for a program. The blue boxes show in which section of the software-hardware stack each factor is contained.

The relationship between the indexing in the logical data array and the data layout on disk

can have a significant effect on the read rate. Before going into detail about access patterns, it is useful to discuss how data is accessed on a hard disk drive (HDD).

2.6.1 Data access from a HDD

Data is stored on disk as a one-dimensional stream of bytes. The relation between the logical order in array space and file space is depicted in Figure 2.6. When the file system requests data from a storage disk, the read head is moved to the correct distance from the centre of the disk so that the correct track on the disk is selected – the track is the path covered by the read head as the disk rotates. The time for this movement of the read head to the track is called the seek time. Once the head is on the correct track, it needs to wait for the data to move under the head, this time is known as rotational latency. The total time to access the data on the disk is the sum of these two times. The initial part of data read from the sector is the header (a sector being the smallest amount of data that can be read or written to disk) and the last part of the sector is the trailer. A header defines the start of the sector and contains metadata about the sector. The trailer defines the end of the sector and contains error correcting code for the sector. This is not to be confused with the header for a file, which can contain metadata on the file. A file is composed of many sectors. The mapping between files and sectors is controlled by the file system. A file is composed of many of these sectors, which, in an idealised situation or clean disk, are stored in order (in practice files will be fragmented, or spread, throughout the disk). (Silberschatz et al., 2013)

Figure 2.6: Relationship between the logical layout of the data in array space (left), and file space (right). The fastest varying dimension is labelled 'Dim 1'.

Data is stored as a one-dimensional stream of bytes, so understanding how this translates to multidimensional data is important. In the array order, the fastest varying dimension (x in the examples here) sits contiguous on disk for the first value of y, z, and t. The next set of x values, for the second y value and first z and t, lies next on disk. Once y has reached its maximum value then z will increase, and so on until the end of the data array. This is shown

in Figure 2.7(a) (in this example there are 4 x, 2 y, 2 z, and 2 t values).

2.6.2 Read patterns with multidimensional arrays

The read pattern can have a significant effect on the read rate. Figure 2.7, and 2.8 shows an example of how the read pattern affects the read rate. With the 'normal' laid out file, the x-t read would be a non-sequential read, which severely reduces the read rate. One technique to avoid this would be to rearrange the data in the file, either by changing the layout of the files (Dong et al., 2013), or by using multidimensional tiling (Section 2.6.3). By optimising the read rate for one read pattern, the read rate for other read patterns can be handicapped. One strategy to avoid this is to store multiple versions of the data. This is not an ideal solution, however, because it increases data storage costs, although compression can reduce this cost by reducing the data volume. Compression of data could also increase the read rate because of less data being read from disk (Baker et al., 2014).

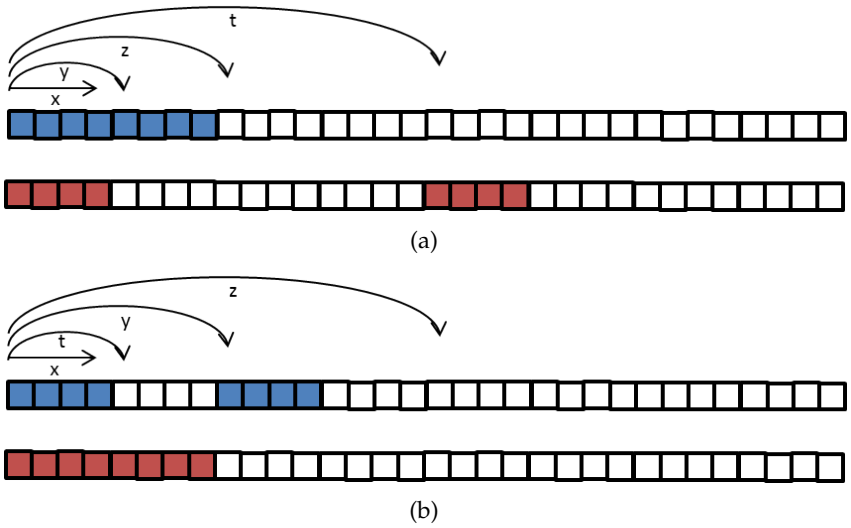


Figure 2.7: Demonstration of the effect of multidimensional tiling to the logical layout of a file in 1D. The blue colour shows an x-y read and red shows an x-t read. (a) shows the normal layout of the file with no chunking applied – x varying first, then y, z, finally t until the end of the array. (b) shows a layout with a chunk shape that contains all x and t – x varies, then t, then y, z until the end of the file. This means that a read in x-t space will read significantly faster in (b) than (a), but the x-y read will be slower.

2.6.3 Multidimensional tiling

Multidimensional tiling is where the logical order of the array is maintained, but the order of bytes on disk is changed, depicted in Figure 2.9. An example of this would be in Figure 2.7

Figure 2.8: Same as Figure 2.6, but with the red box showing how a slice in array space relates to the access pattern in file space.

– when the tiling has changed the order of the file to [z,y,t,x], the arrays are still indexed as [t,z,y,x]. This is a trivial example, but keeping the logical array indexing the same is important in cases where the chunking does not just change the order of the dimensions. For instance, the tiles could be small hypercubes a tenth of the size of each dimension; if this changed the way to index the array it would significantly complicate analysis scripts. The flexibility of multidimensional tiling allows the performance of many different read patterns to improve, but at the cost of the performance of others. An implementation of multidimensional tiling is HDF5's chunking.

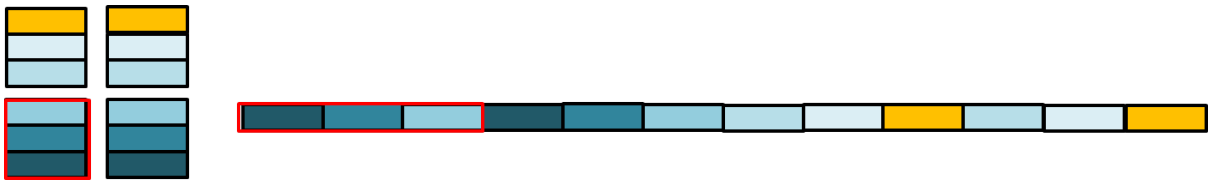


Figure 2.9: Same as Figure 2.6, but with multidimensional tiling applied to the file, giving four chunks. The layout is changed so the access pattern in the red box is contiguous in file space. The yellow boxes show where the file size has been increased because of the mismatch between the tile shapes and the array dimensions.

2.6.4 Compression

Compression reduces the size of data by using an algorithm to discover patterns in the data, so that the file can be represented as repetitions of patterns in the file; reducing the amount of bytes it takes to represent it. There are many different types of compression algorithm, which fall into two main groups, either lossy, or lossless compression. Lossless compression compresses the data without losing any accuracy in the data. Lossy compression however does reduce the accuracy of the data: it identifies unimportant information and removes it from the data (Huang et al., 2016).

Compression can be broadly grouped in to three categories based on how it is implemented. The first method treats the data as a one-dimensional stream of bytes with no knowl-

edge of the data itself. The algorithm then identifies redundancy in the byte stream. This method is generally not very effective with floating point numbers. The second method introduces some prior knowledge about the data type, so is more effective with floating point numbers. However, the floating points are still treated as a one-dimensional array, so it does not perform well with scientific data, where redundancy is often in higher dimensions. The third method applies more prior knowledge about the data, including the relationships between the dimensions. This allows redundancy to be identified along multiple dimensions, which allows greater levels of compression for scientific data. However, this method is slower.

It is well known that floating points do not compress particularly well with lossless compression (Hübbe et al., 2013). An example of a lossy compression method could reduce the accuracy of the data, removing non-scientific, essentially random, data by reducing the significant digits of the data and so reducing the number of bits needed to store the data when converted to integers using a scale and offset (Zender, 2016) (reducing the accuracy of the floating point numbers is not suitable for every work flow, for example, budget studies). Bit grooming is an example of this method, and is similar to the more well known method, 'bit shaving'. Bit shaving converts the floats to integers by multiplication by the number of decimal places, then sets numbers beyond the number of significant digits (nsd) to 0 (Caron, 2014); introducing errors as an underestimation of the true values. Setting the bits to 1 instead overestimates the true values. Bit grooming alternates between 1s and 0s giving a better estimation of the true value (Zender, 2016). The data is still stored as floats in the NetCDF4 file, so not reducing the size of the file – it needs to be compressed in order to gain the benefit of reducing the number of significant digits. This method can improve the effectiveness of NetCDF4 compression.

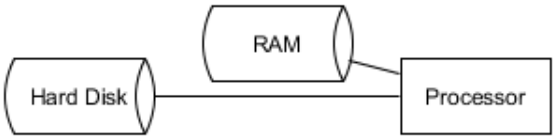
2.7 Parallel data analysis in a HPC environment

A HPC data analysis cluster is composed of, generally, two components: a processing cluster, and a parallel file system. Each are connected via a high-speed network, and can consist of homogeneous (all the same), or heterogeneous (not all the same) nodes.

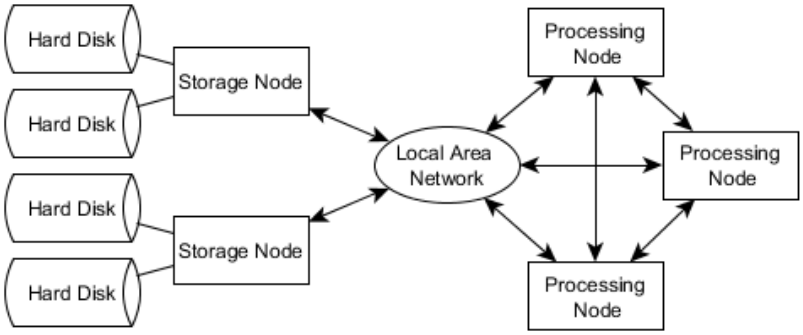
2.7.1 Compute cluster

Processing nodes in a HPC compute cluster are connected to each other via a network, and to a filesystem. Each node consists of one or more multicore processors with memory which

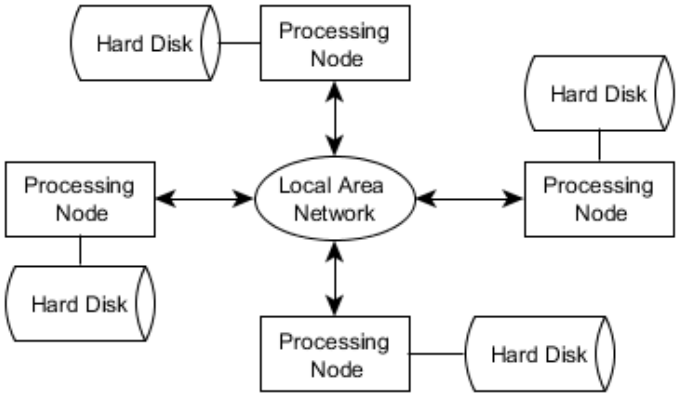
all cores can access. Comparing a personal computer (PC) and a HPC cluster (Figure 2.10), a PC consists of typically one multicore processor, and HDD (or other drive), which are directly mounted, whereas the connections to the storage system in a HPC cluster are over a network. Access to a HPC cluster is usually by a batch queuing system, allowing jobs to be submitted to be processed.



(a) Simplified PC (personal computer) architcture



(b) Simplistic schematic of a HPC cluster with parallel file system. The processing cluster is on the right composed of processing nodes, and the file system is on the left composed of storage nodes and hard disks. NB a processing node consists of RAM and a processor.



(c) Hadoop type cluster

Figure 2.10: Comparison between a typical PC, and a HPC cluster (a) and (b), and a parallel filesystem and distributed filesystem (b) and (c). Lines denote direct mounting, and arrows denote a network.

2.7.2 Parallel file systems

A parallel file system consists of multiple storage nodes connected to a network, designed to store large volumes of data, with fast (often parallel) access, while also providing a global shared namespace for directories and files. The shared global namespace allows access of the data without the user needing information about which specific nodes or disks the data is stored on – this is handled via metadata in the filesystem. A distributed filesystem (an example of which is Hadoop, shown in Figure 2.10) is also designed to store large volumes of data for fast access. The differences are that:

- On a distributed file system, each file is stored on a single node, whereas on a parallel file system a file can be split over multiple nodes.
- With a distributed file system, the storage is often, but not always, directly mounted to the processing nodes, whereas a parallel file system will be separate from the processing nodes.

To determine how a file is split between disks and nodes, a RAID (redundant arrays of independent disks) algorithm is often used to 'stripe' the data (see Figure 2.11) – note RAID algorithms are also used in non-parallel filesystems where multiple disks are used. Both RAID and parallel filesystems can be used to improve I/O performance.

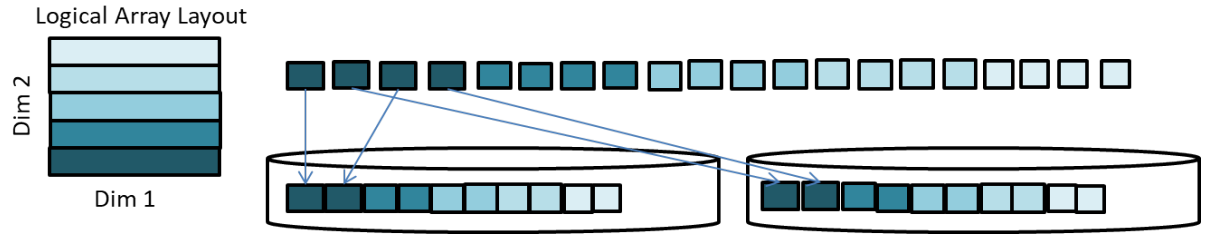


Figure 2.11: Same as Figure 2.6, but with striping applied to the file. In this example, the file is split between two disks in a round-robin style, where alternate stripes are stored on each. (Note, this example shows RAID0 because no redundancy is included.)

A RAID algorithm can offer improved access speed or redundancy for data, depending on the type. The most important types of RAID algorithms to discuss here are RAID 0, RAID 1, and RAID 5. RAID 0 is where the files are 'striped' across multiple disks to improve the transfer rate – striping is where the blocks (small elements of the file) are spread across multiple disks. RAID 0 does not provide any redundancy. RAID 1 is where the disks have a mirrored copy. This provides strong redundancy, and increased access speed, but is expensive for large

volumes of data. RAID 5 uses the same striping idea as RAID 0, but with redundancy provided by distributed parity. Parity is a system which can be used to detect errors in data and reconstruct it (Silberschatz et al., 2013).

The way the data is distributed in a parallel file system can have a large effect on I/O performance. The data distribution will depend upon the parallel file system settings, including RAID, stripe width (size of the stripes), and a minimum file size for striping. A logical file is a stream of bytes, in a parallel file system this is split amongst multiple nodes, and where each section of the file is stored is controlled by a manager node, transparently to the user. The distribution is often a round-robin distribution as shown in Figure 2.11. Using multiple storage nodes enables greater maximum achievable bandwidth to the data, however, the performance could also be affected by the relationship between the read and the stripe width – performance is best when the read matches the stripe width – and likewise, multidimensional tiling will interact with the striping.

More specific details on the platforms used in the thesis are included in Appendix A.

2.8 Alternative approaches to handling big data

The aim of this thesis is not to compare different approaches to parallel data analysis, but to provide quantitative knowledge about factors which affect the I/O performance in a workflow; to provide more general information for developers of software and users of HPC clusters. However, there are many projects looking to improve parallel analysis. Some are outlined here.

Parallel libraries, particularly in Python, attempt to implement parallelism with minimal effort for the user. For example, DistArray[7] aims to have a similar API to the Python library NumPy (popular array based data processing library), but to parallelise operations by using distributed arrays. Another library, Dask[8], represents parallel computations as a task graph, with each operation represented as a node, and the vertices describing how the operations interact. A task scheduling algorithm then implements the graph in parallel.

Middleware (software which sits between the application and operating system), can be used to improve the I/O speed either by implementing the I/O in parallel (MPI-IO [9]), continu-

[7] http://docs.enthought.com/distarray/
[8] https://dask.pydata.org/en/latest/
[9] http://beige.ucs.indiana.edu/I590/node86.html

ously characterising I/O patterns to improve the parallel access (Darshan [10]), or separating the I/O from the application allowing the same script to be run on different platforms efficiently (ADIOS [11]).

ADIOS is middleware which allows an application to decouple from the platform I/O (Lofstead et al., 2008). ADIOS provides a system to describe the data and platform specific factors outside the application through a simple configuration file, meaning that a single application can be effective on multiple platforms. ADIOS also has the ability to use BP (binary packed) files, which gives beneficial performance for reads along multiple dimensions (not normally the case, see Section 2.6) for data with greater than two dimensions with large files (Lofstead et al., 2011). This could solve many of the problems for large data analysis in atmospheric science, if a similar technique was applied to NetCDF4 files.

MapReduce and Hadoop are commonly used in BigData analysis in fields other than atmospheric science. Their advantage is that they easily allow large scale parallelisation across clusters (Dean and Ghemawat, 2008). However, for scientific data, the MapReduce model may not work well because of the mismatch between the logical layout of the data in dimension space, and the physical layout on disk. Buck et al. (2011) aimed to provide metadata about the dimensionality of the data to the Hadoop framework in order to improve its performance with scientific data. Another approach to using Hadoop for processing scientific data is to provide HPC levels of performance to Hadoop-like workflows (Jha et al., 2014). Spark[12] is an adaptation of Hadoop which reduces I/O operations by keeping data in memory. SciSpark is an extension of Spark to scientific data (Palamuttam et al., 2015), which could give significant benefits to big data analysis in atmospheric science. H5Spark (Liu et al., 2016) is another similar approach of using Spark with atmospheric data.

Along with the more general tools and approaches above, another way of providing good performance for big data analysis is to create bespoke applications, or software suites. This allows any complicated programming to be implemented by software engineers, in order to free up scientists to focus on scientific analysis. An example of this is OODT (Mattmann et al., 2006). Some other technical solutions are described in Schnase et al. (2016) and Crichton et al. (2012).

In the concluding chapter of this thesis, the results from Chapters 3-6 are compared with

[10] http://www.mcs.anl.gov/research/projects/darshan/
[11] https://www.olcf.ornl.gov/center-projects/adios/
[12] https://spark.apache.org/

other relevant approaches from literature, some of which are not discussed here.

2.9 Summary

There are many factors which can affect the performance of an application. The key bottleneck to analysis application performance for atmospheric big data is the read rate. Generally, the work which has been discussed in this chapter assesses how an application will perform on a specific system, or provide some ways of dealing with large data sets. What is not quantitatively investigated is how each layer of the software-hardware stack would affect an application. This is crucial in order to make application design decisions before development and assess how categories of analysis would perform. This thesis provides the first steps to this: when considering the analysis which needs executing, what factors affect performance, and so what design decisions should be made to avoid poor performance. Figure 2.12 shows which factors affecting read performance are being analysed in each chapter of the thesis.

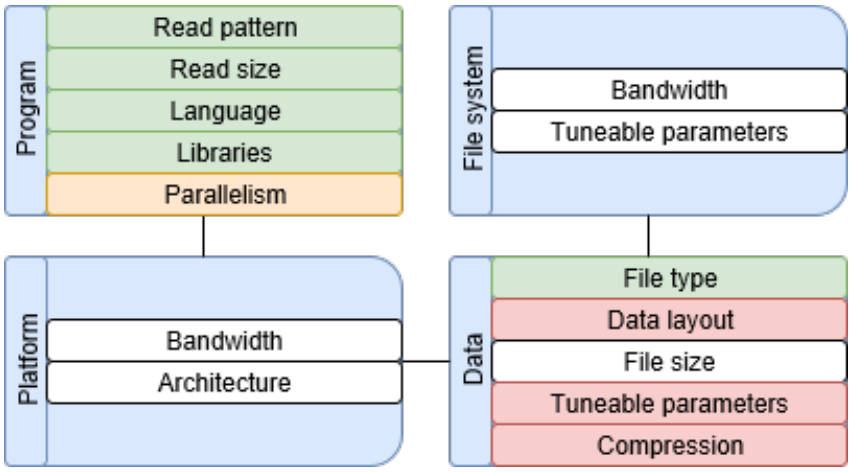


Figure 2.12: Different factors that can affect read performance, with the different colours showing what chapter in the Thesis they are investigated in: green is Chapter 3, red is Chapter 4, and orange is Chapter 5. White boxes are not investigated.

Chapter 3

NetCDF4 Performance on HPC Parallel File Systems

NetCDF4 (see Section 2.4.2) is frequently used in atmospheric science (See Figure 2.3), as is Python. The NetCDF4-python library is a very commonly used interface to NetCDF4 files with Python, and other libraries are often built using it. An important aspect of NetCDF4, which also makes it popular, is the ability to chunk and compress the data (see Section 2.4.2, and Chapter 4). The performance studies investigating NetCDF4 mainly focus on the performance compared to NetCDF3 (Lee et al., 2008), or as part of another library. An important first step in investigating the effect on read rate from the software stack, is to establish the baseline performance (meaning the best possible serial performance) when using netCDF4-python and NetCDF4 files. As far as the author is aware there are no published studies which investigate the performance of NetCDF4 in a serial implementation on HPC parallel file systems.

Analysing the performance of NetCDF4 for typical scientific workflows is critical, and is the first stage in determining the effect on the read rate of chunking, compression, and parallel reads.

The aims of this chapter are to:

- Evaluate the performance effect of using NetCDF4-python, and assess the reasons for any reduction in performance.
- Evaluate the effect on the read rate for different read patterns and read sizes.

Knowledge of any effect on performance from factors outlined in the aims will enable decisions to be made on what combination of factors gives the best compromise for read rate for

large data sizes, and for different read patterns. Also, this knowledge will enable the identification of bottlenecks in the I/O pipeline and whether the effect can be mitigated, or the problem solved. The results from this chapter will provide a comparison for investigations into chunking and compression (Chapter 4), parallel reads from NetCDF4 files (Chapter 5), and the effect on read rate for realistic workflows (Chapter 6).

Literature on the performance of NetCDF4 generally looks at the performance compared to previous versions of the library, with performance examples of new features (Lee et al., 2008). They do not assess the performance which would be seen from a typical user's perspective. The reason for this is that each work flow is very different and there is not a general solution for every work flow and machine architecture.

The contributions from this chapter are:

- Quantitative read rate analysis from NetCDF4 files under a typical HPC analysis environment for atmospheric science.
- Comparison between the performance of the NetCDF4 C library and the netCDF4-python library.
- An evaluation of serial NetCDF4 performance on three UK atmospheric science HPC platforms: JASMIN, ARCHER, and the RDF, each of which has a different type of parallel file system (see Appendix A for details).

This chapter has been published in Jones et al. (2016), copies of which are available on request.

3.1 Methodology

Performance is affected by different layers of the software stack shown in Figure 3.1. The yellow shaded boxes show the variables being investigated in this chapter, and the grey boxes show the variables tested in Chapter 4. The variables for each test are summed up in Figure 3.2. These variables are ones which a scientist wanting to do analysis can easily change, or are factors which could affect the performance further up the stack.

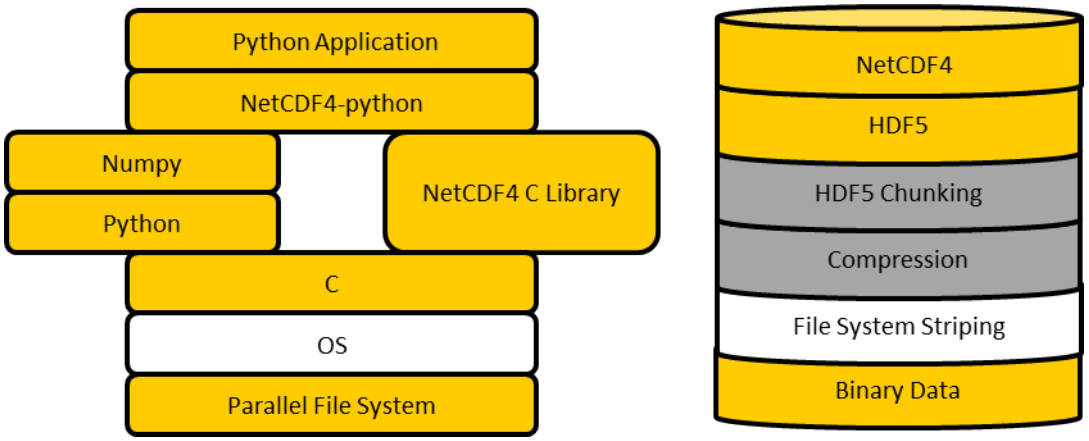


Figure 3.1: The left image shows the software stack for applications built on NetCDF4-python. The netCDF4-python library relies on python (some of which is written in C) and the C NetCDF library. C interfaces with the operating system (OS) which interfaces with the parallel file system to access the NetCDF4 data files. The right image depicts the NetCDF4 data format. NetCDF is built on HDF5, which is a type of binary file. Boxes shaded in yellow are being tested in this chapter, and areas shaded in grey will be covered later in the thesis.

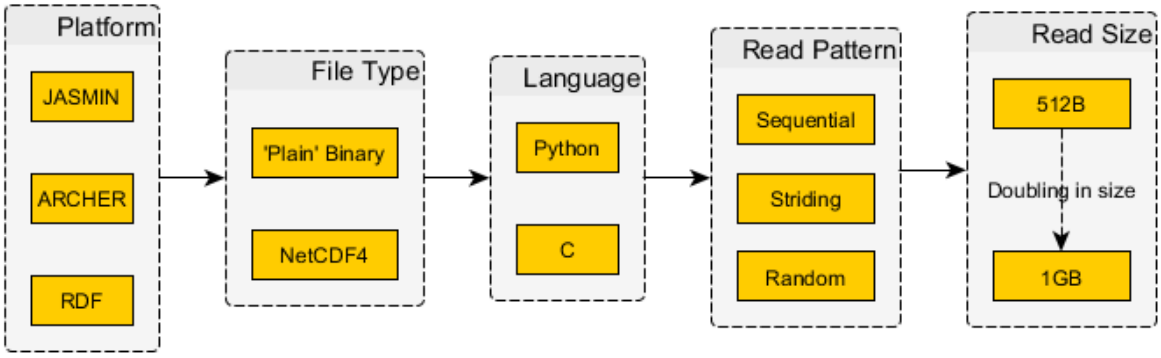


Figure 3.2: Testing domain for this chapter, each box showing the variables, and each group showing the category they lie in.

3.1.1 Testing Domain

The main investigation in this chapter is into the effect using netCDF4-python has on the read rate. In addition, the file type and language were varied in order to identify any reduction in performance in the software stack, this being necessary to localise the cause of any impact on performance. Multiple file systems were used (by execution on different platforms) not as a comparison between platforms, but to provide a wider test base to strengthen conclusions. However, it is also a useful comparison to ascertain whether there is any difference in performance between the platforms.

There are factors which can be varied in the OS and parallel file system related to how the file system handles the data (storage location, and strip width for example), and how the OS

caches data. This is a complex optimisation problem requiring specialist knowledge not within the skill set of a typical user, and not within the scope of this thesis – also, the user may not always have permission to change these parameters, or if they do, they may not want to.

The most basic level in the software stack was used to quantify the baseline performance for each system, which was tested so that when more complex layers are added the effect on performance can be attributed to a specific layer. This means that the cause of any change in the performance can be discerned. The most basic test from the software stack (Figure 3.1) is to read the plain binary file using C using the POSIX read functions, `fseek()` and `fread()`. Python is the final layer in complexity when reading from a plain binary file; the native Python functions `f.read()` and `f.seek()` were used.

To affirm that the C program was performing correctly the `dd` [1] program, a simple Linux utility written in C, was used to read the plain binary files. The `dd` read testing showed very similar results to the C program reading from a plain binary file (the results for reads using `dd` are not shown), showing the C program behaved correctly.

NetCDF4 is built on HDF5 which allows the file to be compressed and chunked. For the work in this chapter the most basic form of a NetCDF4 file was used – a one-dimensional unchunked and uncompressed field. The netCDF4-python library is built on the C NetCDF4 library so the performance of the latter needed to be evaluated, in order to evaluate the performance of the python library.

NetCDF4-python is not the only Python library that can read NetCDF4 files. To provide more information about whether any effect on the performance is caused by the combination of Python and NetCDF4, or whether it was caused by the netCDF4-python library itself, the h5netcdf (Hoyer, 2015) library was also used.

Different platforms were used to provide a larger testing base so that the results are more widely relevant, and any potential system specific bottlenecks could be determined. Three different parallel file systems on three different platforms available to the atmospheric science community were tested on: JASMIN (Lawrence et al., 2012) which uses a Panasas file system[2]; ARCHER (Henty et al., 2015) which uses a Lustre file system[3]; and the RDF [4] which uses a GPFS file system[5]. For a more detailed discussion on the details of these platforms and file

[1] https://linux.die.net/man/1/dd
[2] http://www.panasas.com/
[3] http://lustre.org/
[4] http://www.archer.ac.uk/documentation/user-guide/
[5] https://www.ibm.com/support/knowledgecenter/en/SSFKCN/gpfs_welcome.html

systems, refer to Appendix A.

3.1.2 Read patterns

Non-sequential reads are well known to have a detrimental effect on the read rate (Childs et al., 2005; Blower et al., 2013). It is important, however, to establish, quantitatively, how the read rate is affected on the platforms that tests were run on.

Three read methods were evaluated: sequential reads, striding reads, and random reads. Examples of the sequential and striding reads are shown in Figure 3.3. The sequential reads access the whole file contiguously (that is in the order stored in the file), reading sections of a given read buffer size until the whole file is read. The buffer sizes used started at 512B and doubled up to the largest block size of 1GiB – a complete file read for each buffer size comprised a single experiment. The striding read has the same process except it reads one section of the given buffer size then skips three, repeating this pattern until the end of the file. For the random reads, a set of one hundred uniformly distributed random numbers were generated for the read offsets, the read is then performed for the required buffer size, e.g. 1 GiB. This number of reads was chosen to keep the chance of re-reading a section of the file from cache low (reading from cache would artificially increase the read rate, because it would be read from cache or memory). The random reads differ from the other reads in that the direction of the seek is not always forward and the size of the strides through the file vary.

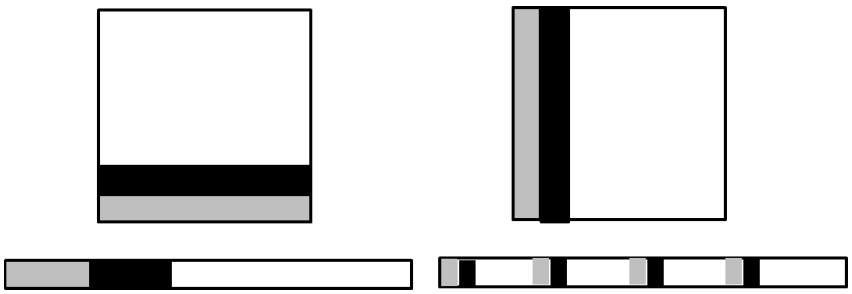


Figure 3.3: Examples of a sequential read (left) and a striding or striding read (right). The top images show what the read would look like in two-dimensional array space, and the bottom images show how the read corresponds to the read pattern through the one-dimensional file space. The grey shows the first read and the black shows the second read.

The sequential reads were designed to simulate a best case scenario where the read from the file is contiguous – a read where the bytes in the file are stored next to each other. The striding reads are designed to simulate a read which is not contiguous in the file with a regular stride pattern through the file. This is representative of a slice through a dimension which is

not stored contiguously in the file (where there are two or more dimensions), see Figure 3.3 and Section 2.6. Striding reads are a common read pattern in atmospheric science analysis. The random reads simulate a worst case scenario, where the direction of the seeks through the file are not consistent, and neither is the distance of the seek.

3.1.3 Testing algorithm

Pseudo code examples of each read from plain binary files is included below. See Appendix C for the full code.

```
# Sequential read
f = open(filename)
for number of buffers until end of file:
    data = f.read(buffersize)

# Hopping read
f = open(filename)
readoffset = 0
for number of buffers until end of file:
    f.seek(readoffset)
    data = f.read(buffersize)
    readoffset = readoffset + 4*buffersize

# Random read
f = open(filename)
readoffsets = genrandoffsets(length=100)
for offset in readoffsets:
    f.seek(offset)
    data = f.read(buffersize)
```

3.1.4 Test files

For the binary tests using C and Python, a plain binary file consisting of random numbers was created on each system using the Linux `dd` command. To avoid disk buffering as much

as possible, the file size was set to be over twice the size of the RAM of the system. The file size on all platforms was 256GiB (275GB). The plain binary file was read using `fseek()` and `fread()` in C and `f.seek()` and `f.read()` in Python.

The netCDF4-python library was used to create the NetCDF4 files for testing. The file, as with the plain binary files, was created to be over twice the size of the RAM on the compute nodes. The file size on each platform was 257GiB, and the file contained a single 1D contiguous (not chunked) variable consisting of 8 byte floating point random numbers. Each read size,in bytes, was converted into a number of 8 byte floating point numbers to stream from the file on each read with files indexed by element – the plain binary file was indexed by bit, but the NetCDF4 file was indexed by array element.

3.1.5 Repeats

To gain a better understanding of the variability involved, tests were repeated between three and five times. Ideally more repeats would have been run, but this was not possible due to the time intensive nature of the tests. However, useful conclusions can be drawn with the number of repeats using the mean and the standard deviation to obtain an understanding of the variability.

3.1.6 Deeper analysis

A few different methods were used to gain a deeper understanding of what was going on in the results.

The balance of CPU and wall time is important to understand what is happening when the libraries read the NetCDF4 file. The CPU time is the terminology that will be used throughout for any time which work is being done on the processor. The clock functions from C and Python report the current processor time. Using this, the time elapsed for the CPU processing instructions can be timed. When the processor is idling waiting for data, for example, the processor time will not increase, but the wall time will. This work assumes that the majority of the time difference between the wall and CPU time accounts for I/O latency – the time for the data to be read from disk. For the C testing program the POSIX function `clock`[6] was used, and in the Python testing program the function `clock` was used from the `time` library[7]. This

[6] http://pubs.opengroup.org/onlinepubs/009695399/functions/clock.html
[7] https://docs.python.org/2/library/time.html

can be used to determine whether the program is waiting on I/O from the file system.

The Linux utility `strace` was used to gain more information about the system calls, including the size of each I/O call and the time taken for the system calls.

3.2 Baseline read performance

Baseline performance was investigated using C and Python reading from plain binary files for each system in the following sections.

JASMIN

Figure 3.4 shows the plain binary read performance using C and Python on the Panasas platform. The Python rate is lower at higher read buffer size (above 16MiB) than the C results - 30% at 1GiB buffer size. The profiles of the striding read with C and Python are very similar throughout the whole range of read buffer sizes starting off with very low read rates, increasing to around 400MB/s at 512MiB . The variability of the C reads is much higher than the python reads from 128MiB and above. The variability for the remaining read buffer sizes for each remains low. The random read profile increases in a similar way to the hopping reads, although with much higher variability when using C. It is not clear what has caused this variability.

The Python striding and random test were not run on the other platforms because of the general similarity between the C read rate and the python read rate from plain binary files on JASMIN, and for the sequential reads on the other platforms, only the C program was used to measure striding and random reads – because of the very long run times at such low buffer sizes.

ARCHER

The results for the plain binary reads on the Lustre platform are shown in Figure 3.5. The results are similar to the Panasas results, differing at low buffer size for the random reads with a more gradual slope up to peak performance, and an increase in performance above 64MiB for the sequential reads. The striding reads also have a higher read rate than would be expected from the JASMIN results above 256KiB.

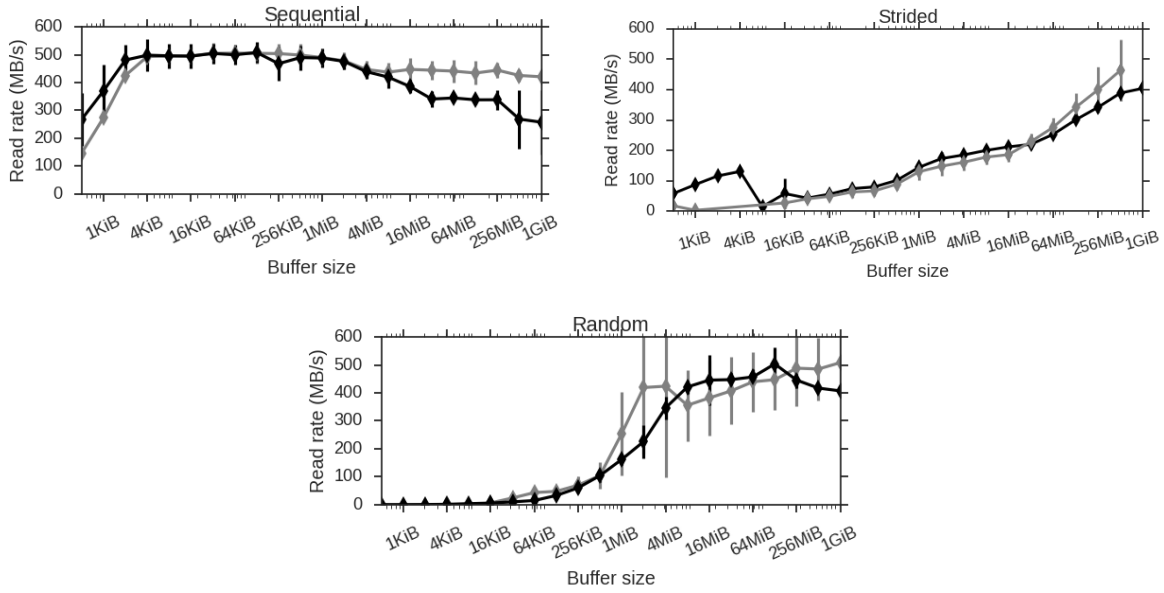
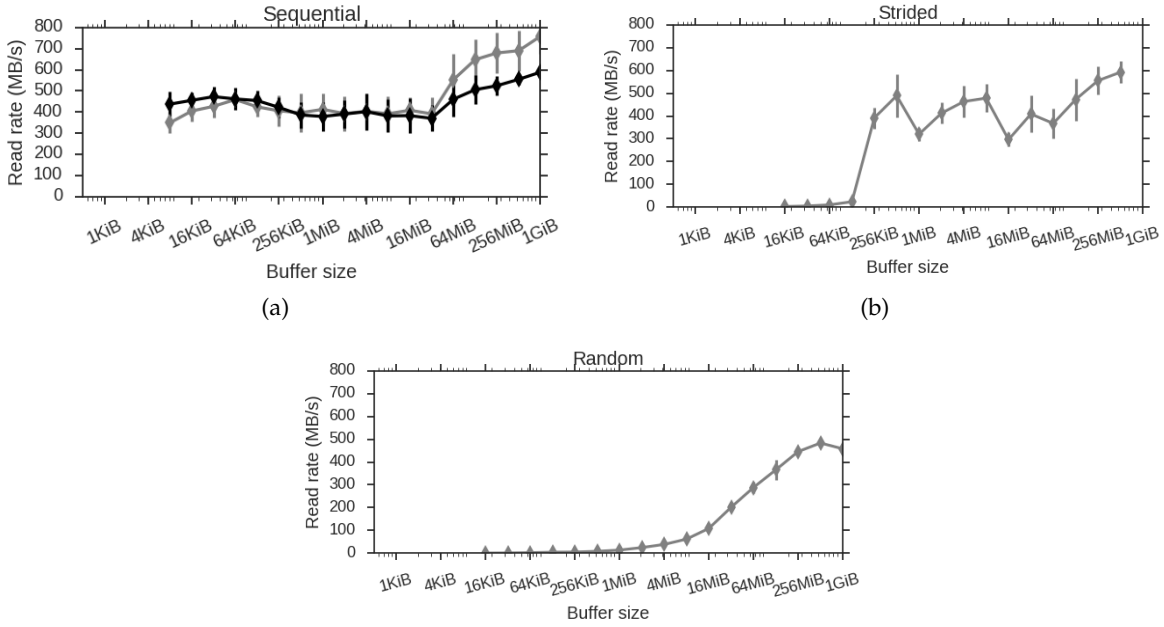


Figure 3.4: Average read rate (diamonds) and one standard deviation (error bars) for the tests reading from a binary file on the Panasas platform. Grey lines show the results using C and black lines show the results using Python. The first graph shows the sequential reads, second shows hopping reads, and bottom shows random reads.



(a) (b)

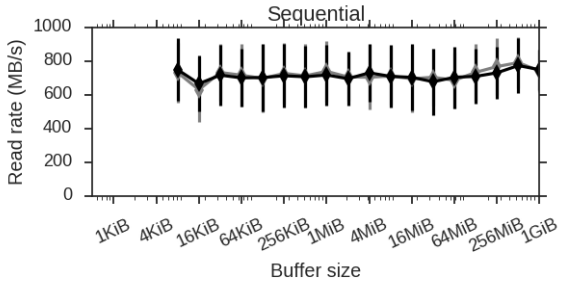
(c)

Figure 3.5: Results from ARCHER showing the read rate from the C scripts reading from plain binary files

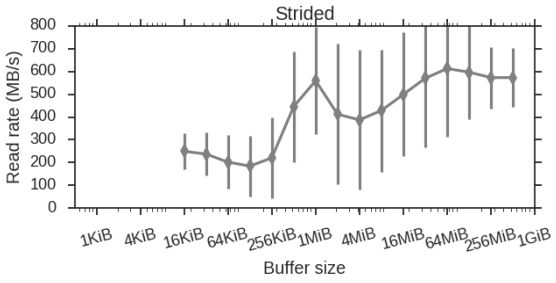
RDF

The results for the C plain binary reads on GPFS are shown in Figure 3.6. The profile for the sequential reads is similar to the Panasas results, albeit at higher bandwidth because of the

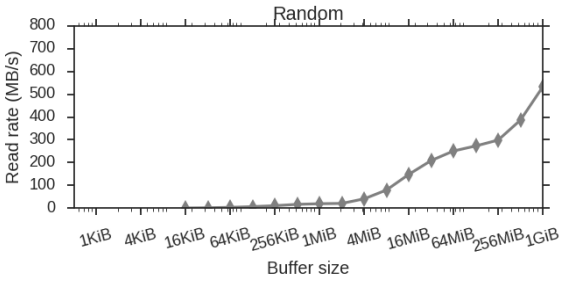
Infiniband used on the GPFS platform compared to 10Gb/s Ethernet in Panasas. The variability for the hopping reads is high which makes the profile more difficult to interpret. It is interesting however that the average striding reads are consistently higher than the random reads.



(a)



(b)



(c)

Figure 3.6: Results from the RDF using a C program to read from plain binary files.

Discussion

All of the results between different platforms have similar characteristics. At low buffer size for the sequential results on JASMIN, below 4-8KiB, there is a significant hit in performance; likely due to the reads being smaller than disk sector size which reduces read rate, although this has not been verified. For all platforms the sequential reads are the fastest, which is expected because it is a contiguous read on disk, and being the easiest for the file system to anticipate the next read request from the program.

All the random reads and striding reads have similar profiles, showing the same gradual increase in read rate and have a lower read rate, interpreted as the effect of having to seek through the file. Because of the faster reads at larger buffer sizes, the effect of this is reduced due to larger reads. On JASMIN the small difference between the random and striding reads show that the size and direction of the seek does not have much of an effect on the read rate. The variation of the random reads is much higher than the other reads; likely due to the possibility of hitting caches because of the random nature of the read. The small difference between the striding and random reads and the shape of the profile implies that the size of the read has more of an impact in the read rate than the direction or size of the stride.

The results here compare well with that of others: the profile shape for the sequential reads for all platforms agrees with the results from Bartz et al. (2015). The expected bandwidth on the Lustre platform is around 500MB/s (Henty et al., 2015) which agrees with our results, and the peak read rate on JASMIN is similar to performance results (personal communication, Bryan Lawrence). The variation in the non-sequential reads compared to the sequential reads is also seen in Schmid and Kunkel (2016).

For all of the results there are two stand out patterns with increasing read size. The first is for the sequential reads which very quickly rise to a level maximum and stay roughly constant for the remainder of the read size, this shape will be referred to as a flat profile. The second pattern seen in the striding and random reads more gradually increases, peaking at larger read sizes, this will be referred to as a steadily increasing profile. Using these two terms, the results are summarised in Table 3.1.

The main conclusions from this section are as follows:

- The different read patterns have a very large effect on the read rate, with a lesser effect at larger buffer sizes. The cause of this is likely to be because of seeking through the file, which does not happen to the same extent in the sequential reads, therefore not decreasing the read rate. Alternatively, the file system could be successfully reading ahead for the sequential reads, but not for the striding and random.
- The read rate drops significantly at very small buffer size which agrees with results from Bartz et al. (2015).
- There is very little difference between C and Python when reading plain binary files.

Table 3.1: Summary of results when reading from 'plain' binary files.

Platform	Language	Read Pattern	Resulting Profile	Peak Rate (MB/s)	Notes
JASMIN	C	Sequential	flat	500	
JASMIN	C	Striding	steady increase	500	
JASMIN	C	Random	steady increase	500	
JASMIN	Python	Sequential	flat	500	
JASMIN	Python	Striding	steady increase	400	
JASMIN	Python	Random	steady increase	500	
ARCHER	C	Sequential	flat	750	increase above 32MiB
ARCHER	C	Striding	different	600	sudden increase around 256KiB
ARCHER	C	Random	steady increase	500	
ARCHER	Python	Sequential	flat	600	
RDF	C	Sequential	flat	800	
RDF	C	Striding	steady increase	600	much lower peak than sequential
RDF	C	Random	steady increase	600	much lower peak than sequential
RDF	Python	Sequential	flat	800	

- The read rate profiles between the different platforms were similar, but different in magnitude due to bandwidth differences.

3.3 NetCDF4 read performance

The following sections show the results for the read performance when using NetCDF4 files on each system using the same conditions as the previous section, but reading from NetCDF4 files.

JASMIN

To investigate the effect of NetCDF, the experiment from the previous section was repeated using the C NetCDF library and the netCDF4-python library. Figure 3.7 shows the results. The C results look similar to the results when reading from a plain binary file – same peak performance and profile. A reduction in performance is seen in the Python profile for read block sizes less than 64K. Testing using the Linux utility strace[8] showed that the reduced rate is caused by the netCDF4-python library reading a minimum of 64KiB per read, even when less was requested. The peak performance for the Python library was almost 40% worse than

[8] https://linux.die.net/man/1/strace

the C library at the 1MiB buffer size, then reduced further at larger read sizes to 150 MB/s, around 75% worse than the C read rate.

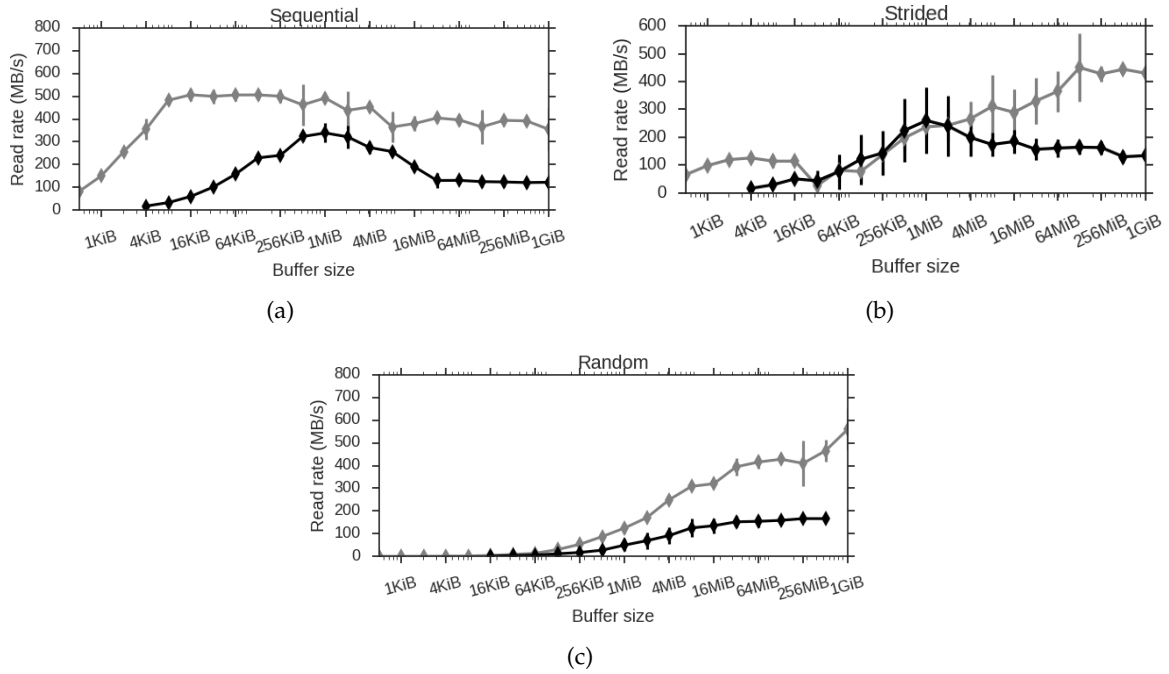


Figure 3.7: Results from JASMIN using netCF4-python (black) and the C NetCDF4 library (grey).

h5netcdf

To determine whether the drop in performance when using the netCDF4-python library was due to the combination of the NetCDF4 file format and Python, or whether it was due to the library, another library was used. h5netcdf is built on the Python library h5py, which reads HDF5 files, and extends the h5py library to also be able to read NetCDF4 files using the same syntax as the netCDF4-python library. This meant it was an ideal library substitute for netCDF4-python.

The results are shown in Figure 3.8. The peak performance is greater than when using netCDF4-python and much closer to the Python plain binary results and the tests using the C programs. The same slow initial ramp up in the read rate is seen on the h5netcdf results and this is due to the same reason as the netCDF4-python results – the libraries always read at least 64KiB of data from the file, likely due to a buffer in the libraries, or interaction with the 64KiB Panasas stripe width. The performance at larger read sizes is much closer to Python reading from a plain binary file, not showing the same significant reduction in performance as

netCDF4-python.

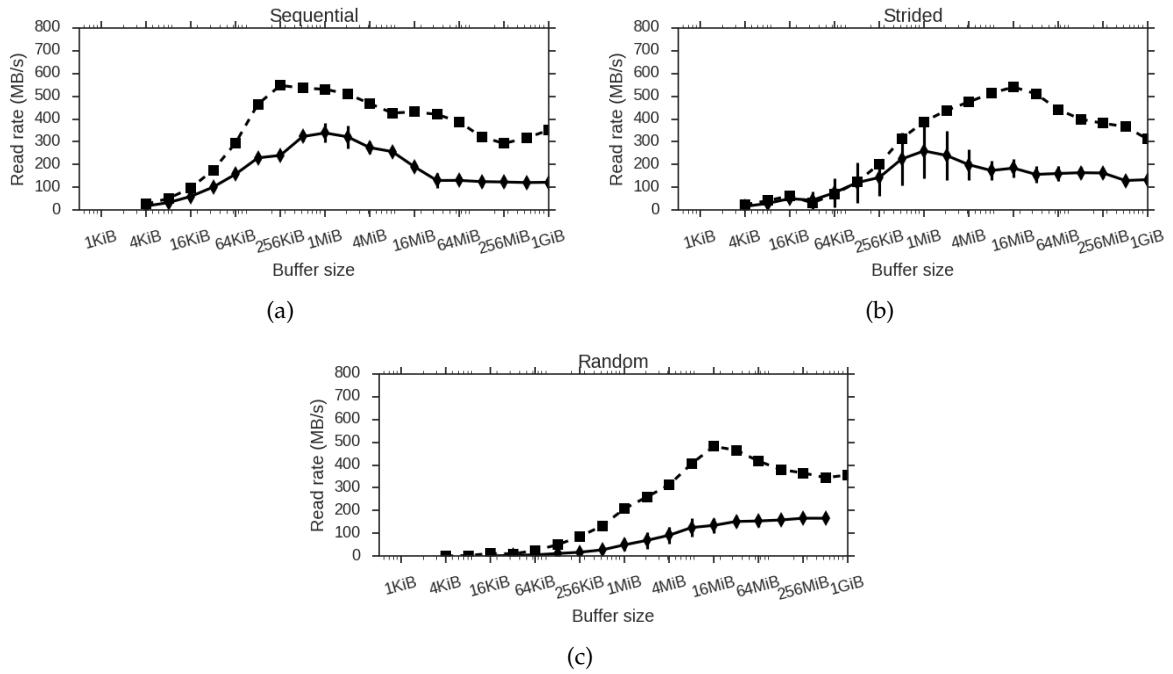


Figure 3.8: h5netcdf read rate results when using JASMIN. The diamonds with solid lines show the netCDF4-python results, and the squares with dashed lines show the h5netcdf results.

ARCHER

The majority of the results from ARCHER when using NetCDF4 are similar to the JASMIN results (Figure 3.9), showing the same drop in netCDF4-python performance at larger buffer size. The rise in performance to around 700 MB/s at 1 GiB read size for the NetCDF4 C library is also seen when reading from the plain binary files.

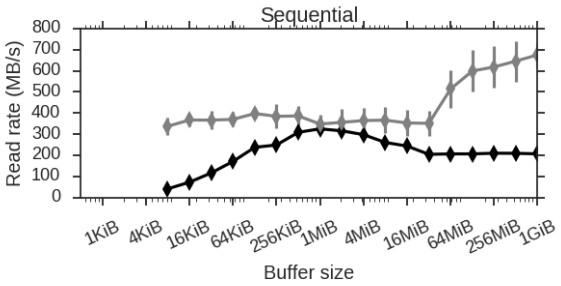


Figure 3.9: Results from ARCHER using the C NetCDF4 library (grey) and the netCDF4-python (black)

RDF

The results from the RDF are similar to the JASMIN results in that the performance profile for the netCDF4-python library is very similar (shown in Figure 3.10). The C library results are very variable, as with the C results from plain binary files, which is likely due to other users working on the RDF. This means, taking into account the variability the results, the performance when reading from plain binary files and NetCDF4 files is likely to be similar as when using C. When reading from NetCDF4 files using netCDF4-python the same humped profile is seen with a significantly reduced peak performance, with the peak at around 1MiB, as other platforms.

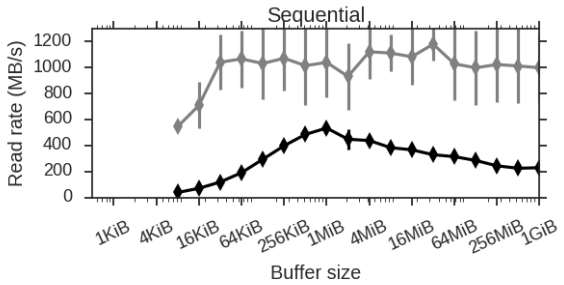


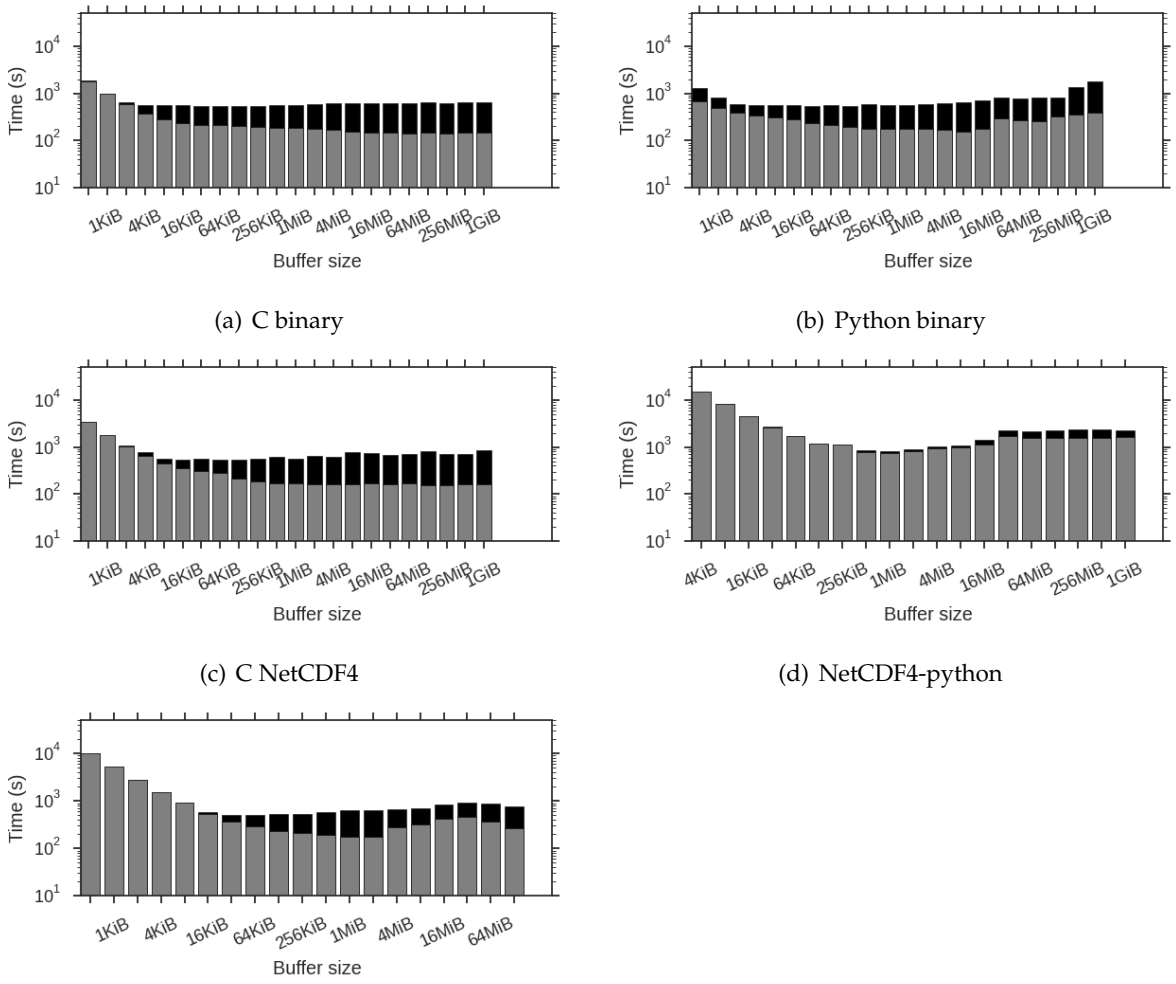
Figure 3.10: Results from the RDF using the C NetCDF4 library (grey) and the netCDF4-python (black)

CPU time vs. wall time

To further investigate the reasons for the reduced performance, the time the CPU spent processing instructions was measured using the POSIX and Python `clock` functions. The results from this are shown in Figure 3.11. For the all reads except when using NetCDF4-python, the general pattern is that above 4KiB to 16KiB the total time for the program (wall time) is much greater than the CPU time. This is indicative of a read which is waiting for the I/O system rather than CPU work. The assumption here is that the time spent during system calls (i.e. non CPU, or user time) is mostly spent on I/O. From using strace to analyse the system calls it can be said with confidence that this is true – the read calls take over 99% of the time reported time from strace. For the NetCDF4-python reads, the CPU time dominates, increasing the total time above what is seen from the I/O bound time of the other reads. This means that the reads using netCDF4-python could be CPU bound. This is also due to the library, as is shown by the difference in pattern between the netCDF4-python results and the h5netcdf results - if the increased CPU time was attributed purely to Python and NetCDF4, the h5netcdf reads would be

CPU bound as well. Therefore, the netCDF4-python library is CPU bound and this is caused by how the library is written.

To summarise, the CPU limited behaviour explains the significantly reduced read rate for netCDF4-python compared to the other tests – those being I/O limited.



(a) C binary

(b) Python binary

(c) C NetCDF4

(d) NetCDF4-python

(e) h5py

Figure 3.11: Comparison between average wall time and CPU time for the different sequential read tests on JASMIN. Black bars show wall time, and grey bars show CPU time, note that the bars overlap.

Discussion

For all Python read rates, there is a reduction at larger read buffer sizes (much more severe for netCDF4-python). This reduction is caused by more time being spent processing instructions (CPU time) compared to the C library, as shown by the CPU time results in Figure 3.11.

Another element in the software stack for netCDF4-python is Numpy (Figure 2.5). To elim-

inate Numpy as a factor in the performance drop, a test was run using Numpy to read the plain binary files. This gave similar results to the Python profile in Figure 3.4, indicating that the significantly lower read rate seen in the netCDF4-python results was not caused by Numpy. A more in depth look at the library is required to determine the reason for the drop in performance.

The striding reads with netCDF4-python follow a similar profile to the sequential reads, albeit with much higher variability and are similar to the C striding read until 16MiB where the-larger reads have a more significant effect. The random C profile looks similar to the Python, and the other Panasas results, with the netCDF4-python results performing worse, particularly at large buffer size.

Similar to the baseline results reading from plain binary files, there were stand out profiles. In addition to the flat and steadily increasing profiles, a humped profile was introduced for the results when using netCDF4-python; consisting of peak around 1MiB, with the read rate reducing for larger reads. The NetCDF4 results are summarised in Table 3.2

Table 3.2: Summary of results when reading from NetCDF4 files.

Platform	Language	Read Pattern	Resulting Profile	Peak Rate (MB/s)	Notes
JASMIN	C	Sequential	flat	500	
JASMIN	C	Striding	steady increase	450	
JASMIN	C	Random	steady increase	500	
JASMIN	Python	Sequential	humped	300	
JASMIN	Python	Striding	humped	250	
JASMIN	Python	Random	steady increase	200	
JASMIN	h5netcdf	Sequential	near humped	500	
JASMIN	h5netcdf	Striding	steadily increasing	500	
JASMIN	h5netcdf	Random	steadily increasing	450	
ARCHER	C	Sequential	flat	700	increase above 32MiB
ARCHER	Python	Sequential	humped	300	
RDF	C	Sequential	flat	1000	high variability
RDF	Python	Sequential	flat	500	

The main conclusions from this section are as follows:

- The C NetCDF4 read performance is very similar to read performance when reading plain binary files using C and Python.
- Using netCDF4-python the performance is lower generally, but is significantly reduced at buffer size of 8 MiB and higher.
- Peak read rate for sequential reads using netCDF4-python is at around 1MiB buffer size.

3.4 Conclusions

The effect on performance in the NetCDF4 software stack was identified by running tests using C and Python when reading from plain binary files and NetCDF4 files. The drop in read rate performance was found to be due to the netCDF4-python library, and not due to either Python or the NetCDF4 file format.

Returning to the aims of the chapter the first aim was:

- Evaluate the performance effect of using NetCDF4-python, and assess the reasons for any reduction in performance.

The netCDF4-python library performance was compared to that of the C NetCDF4 library and another Python library (h5netcdf). It was found that:

- The netCDF4-python library performs less efficiently than the other tests, giving a lower read rate, which was especially prevalent at small (less than 64KiB) and large (greater than 8MiB) buffer sizes, with peak performance at a buffer size of about 1MiB. This drop in read rate could have a significant performance impact on analysis scripts which use netCDF4-python.
- There is little difference between the performance of C reading from plain binary files, Python reading from plain binary files, and C reading from NetCDF4 files.

The second aim of this chapter was:

- Evaluate the effect on the read rate for different read patterns and read sizes.

As part of the comparison between the different libraries and file formats, different read patterns and read sizes were tested. The results from this showed:

- The read pattern has a large effect on the performance of a read, meaning that any seeking done in an analysis script is very expensive. Therefore, keeping as many reads as possible contiguous on disk is very important. The striding pattern performed better for larger read sizes, so keeping the read requests large helped with the throughput – the fewer IOPS with non-sequential reads were compensated for by the larger buffer size (see Section 2.3 for details).
- The read size had most impact at small buffer sizes (less than 4-8 KiB for C when using NetCDF4 and not using it, and when reading from plain binary files with Python, and

less than 64 KiB for Python reading from NetCDF4 files), and for netCDF4-python at larger buffer sizes over 8 MiB.

These results could have implications in the design of analysis scripts, and choices made when deciding what order to store the dimensions, and what chunking specification to use in NetCDF4. The buffer size of around 1MiB may be the most efficient size for reads and chunk sizes. Also, avoiding any reads of less than 64KiB would benefit performance. The results also show that sequential reads are significantly more efficient, meaning that for analysis scripts to be most efficient, as much of the reading done from a file should be contiguous. Another important implication is that NetCDF4 is read at the same rate as plain binary files when using C. This means that there is no apparent disadvantage to reading from NetCDF4 files from the point of view of the read rate.

The results from the netCDF4-python portion of this chapter are summarised in Figure 3.12. With the maximum rate from the C tests shown at the top of about 500 MB/s. The reads are then split into sequential and non-sequential. The striding and random reads are combined into the non-sequential reads; random reads are not generally a directly applicable read pattern for atmospheric science, and the profiles were similar. The two reads combined can give a range that the non-sequential reads could be.

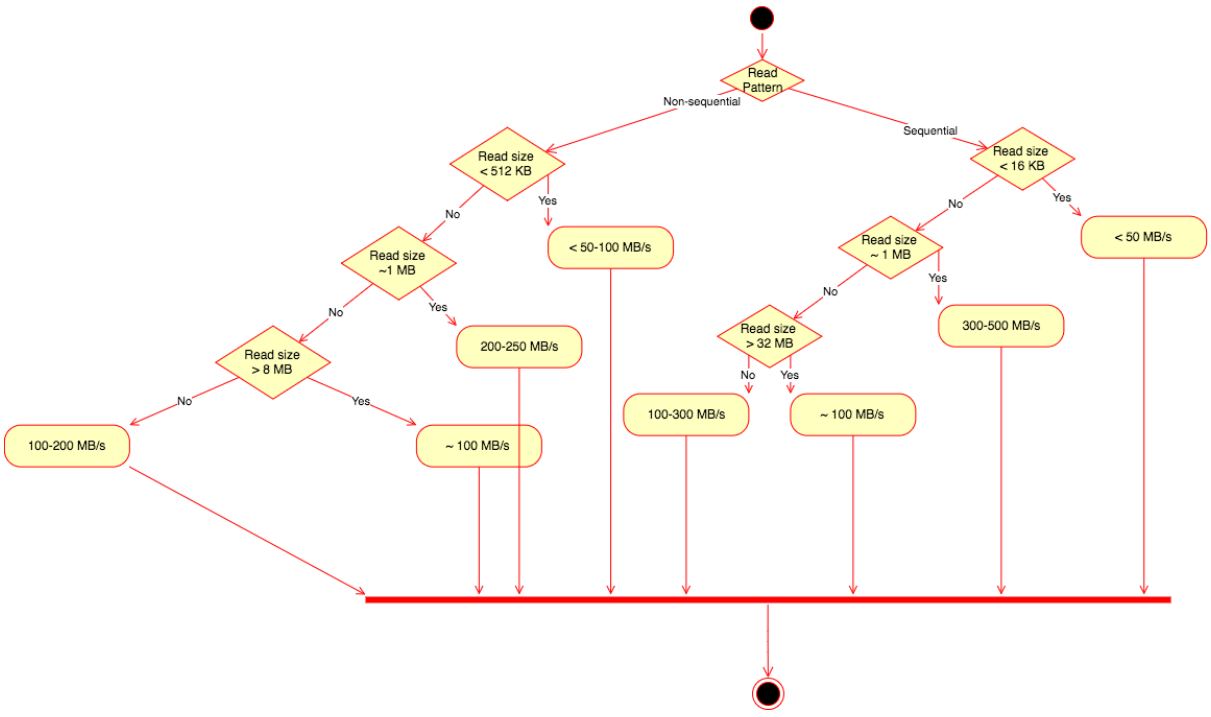


Figure 3.12: Summary of the performance results from the chapter. The rates were estimated from the results in this chapter, taking a range of values from the relevent graphs.

This chapter assumes a very simplistic view of a NetCDF4 file. In general atmospheric science data will be multi-dimensional, and may be chunked (for faster alternative access patterns), or compressed (to reduce the impact of storing large data sets). The next chapter deals with these complications, looking at the read performance for four dimensional chunked and compressed NetCDF4 files.

Chapter 4

NetCDF4 Chunking and Compression Read Performance

The previous chapter investigated and quantified the read performance from NetCDF4 files with different read patterns, particularly when using the netCDF4-python library. The data in atmospheric science typically contains more than a single dimension (particularly when stored in NetCDF4 files). One method of improving access for reads along the dimension which is not the fastest varying, is to change the order of data in the file by shuffling sections of data in to a different order, while keeping the same logical order in the dimensions of the file. This can be achieved via a process called chunking with NetCDF4 (and HDF5) files (see Section 2.4.2). This chapter looks to quantify the effect chunking and compression has on the read rate on HPC clusters with parallel file systems (using JASMIN as a typical example).

In general, matching the chunk shape with the read will provide the best possible performance from chunking (Lee et al., 2008). This, however, would give poor performance for the other read patterns, for example, if rechunking data to provide fast reads for an x-t read slice, the x-y read would be significantly worse compared to an unchunked or x-y chunked file. It is possible that a compromise can be made for access along different dimensions, as has been previously shown on a HDD (Rew, 2013). The results from Rew (2013) showed a performance increase of around 100 times for a time series read from contiguous (unchunked) data which favours spatial reads. However, the spatial read was then around 100 times worse. While the speedup for the time series is good, with large data sets the reduction in read rate for the spatial read could have a very large impact in the workflow for spatial data analysis. Part of the investigation in this chapter is to assess whether this behaviour is the same on a parallel file

system.

There are two aims for the chunking sections of this chapter. The first is to quantify, for the tested reads and chunk specifications, how the read rate is affected. The second aim is to assess whether there is a compromise with the chunk specification on a parallel file system to benefit a striding read, while also giving a passable read rate for the compromised sequential read. The importance of the second aim is that if the file can be chunked in a way that gives acceptable read rate for two different reads, then only one version of the file needs to be stored rather than two – this assumes that the aim of the data being stored is to give relatively good read rates for multiple users reading from data in an archive with different workflows.

An alternative to chunking is to store multiple version of the file and compress them. Compression can be highly beneficial for a HPC environment, by reducing the time and storage involved with data storage and access (Chasapis et al., 2014). The two files could take the same or less space than a single uncompressed file, giving multiple different workflows a good read rate. This leads to the second investigation in this chapter – how does compression affect the read rate?

Reducing the size of the file on disk would mean that the time to get that section of data to memory could be reduced (less bytes to read), but this compressed data then needs to be unpacked increasing the work for the processors. To compress NetCDF4 files they need to be chunked. This leads to an interaction between chunking and compression. This is complicated because each compressed chunk needs to be read into memory in its entirety before being uncompressed, which could lead to unwanted data being read and uncompressed.

Compression reduces the size of data by using an algorithm to discover patterns in the data, so that the file can be represented as repetitions of patterns in the file which reduces the amount of bytes it takes to represent a file. NetCDF4 uses deflate data compression, which uses the zlib compression algorithm (Lee et al., 2008), allowing sections of the file to be compressed and accessed individually (in this case these sections are chunks). It provides a good balance between compression ratio and compression speed (Liu et al., 2015). Because the data size is reduced, reading less data from disk could mean that the read rate of the file could increase (Miller, 2015), but obviously this data will then need to be uncompressed on the processing node, meaning there is a balance between increased read rate due to fewer bytes being read, and the extra CPU required to uncompress the data. This chapter investigates the compression difference and read rate from the default NetCDF4 zlib compression, and the Bit Grooming

method (from Zender 2016, discussed in Section 2.6.4). The first part of the compression investigation in this chapter is to implement the compression on real climate data and measure the compression ratios. The second part then investigates the read rate from these files.

It could be faster to take data in one chunk shape with a poor read rate, and reformat into another before analysis. This is an important section of the investigation in this chapter, because it could be faster for the work flow to reformat the data before executing an analysis, rather than perform an inefficient read – but this needs quantifying, since it is not discussed in the relevant literature.

An alternative to changing the data to suit multiple use cases would be to stage the data so that only one version of the data is archived, but the data (which could be a subsection of the data) that the user reads is stored in a temporary space. This could be done using burst buffering (to give a faster read rate), or read by the file system while the user is queuing. Another solution could be to simply read the data into a machine with very large memory, though this might be impractical due to memory costs. Neither of these alternatives are being tested in this thesis because they are architecture dependant.

The aim of this chapter is to:

- Quantitatively evaluate the effect that chunking and compression have on the read rate, and therefore the overall workflow.

The results in this chapter are an important stepping stone in the greater picture of investigating factors which can affect performance of atmospheric data analysis applications; in which chunking and compression can have a huge effect.

4.1 Method

There are three parts to the investigations in this chapter, which are depicted in Figure 4.1. The following sections discuss the methods for each. For each test, four-dimensional files were used to be more representative of atmospheric data (time, height, latitude, and longitude dimensions) than the one-dimensional files in Chapter 3. A side benefit of this is to affirm that the results from Chapter 3 in terms of idealised read patterns are relevant in higher dimensional data.

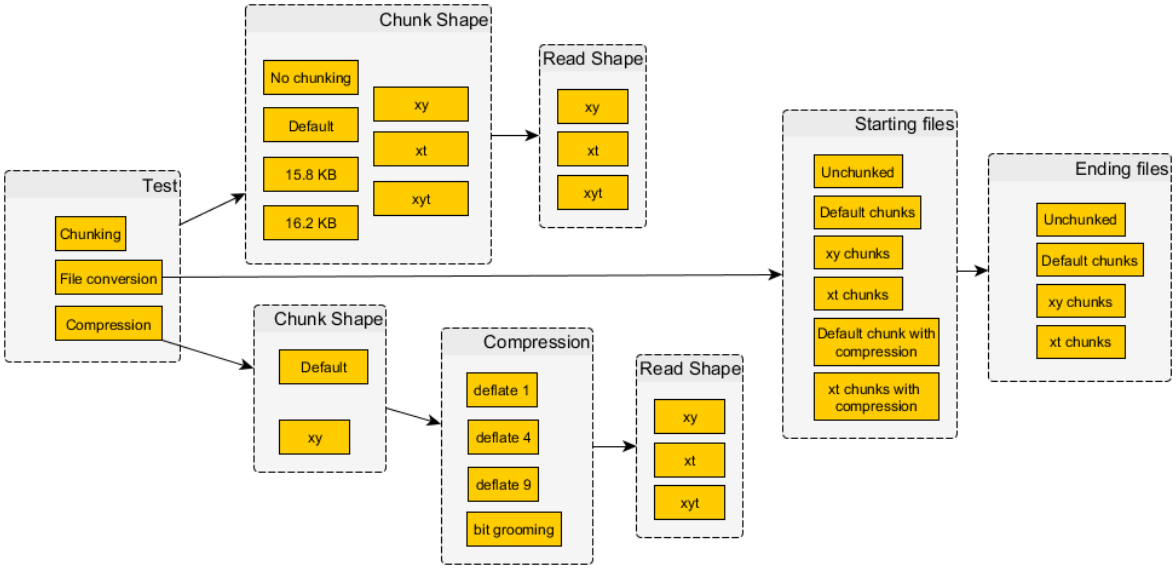


Figure 4.1: Scope of the testing in this chapter.

4.1.1 Chunking performance

To test the effect of chunking on the read rate, files with different chunking specifications needed to be created. The following files were created using the nccopy utility[1] from an unchunked file with dimensions [430,430,430,430] – giving a 274 GB file, which was significantly larger than the size of RAM on the processing nodes. Equal dimensions were used to try and make analysis of the performance easier, by keeping the reads along different dimensions the same size. Both chunk and read shapes are described in this work by the dimensions which they contain, for example x-y describes where the shape would be [1,1,430,430] from the files, containing all the data from the latitude and longitude points for a single time and height.

- Chunk shape A – unchunked. This provided a baseline to compare the read rate to the NetCDF4 sequential read rate from Chapter 3, and should give fast contiguous reads (e.g. x-y maps).

- Chunk shape B – with equal dimensions ([43,43,43,43]), similar to the NetCDF4 default chunking scheme, but as a multiple of the array dimensions. This specification will probably not perform very well on large data sets, but multi dimensional hyper cube shaped chunks smaller than the dimension lengths could give a compromise between different read patterns (Rew, 2013).

[1] http://www.unidata.ucar.edu/software/netcdf/docs/netcdf_utilities_guide.html

- Chunk shape C – 15.8KB chunk size chunked as [1,1,43,43]. The reason for this chunk specification is that small chunks can represent a good compromise in read rate for different read patterns (Rew et al., 2010). Four of these chunks fit into the 64 KB file system stripe width (see Appendix A), which should provide better read rate than D.

- Chunk shape D – 16.2KB chunk size chunked as [1,1,45,45]. Four of these chunks are larger than file system stripe width (64KB), and three are significantly smaller. This would likely decrease the read rate when compared with chunk shape C, because of the incompatibility between the chunk size and stripe width.

- Chunk shape E – xy chunks [1,1,430,430]. This would be fast for xy read, but how does this read compare with the unchunked read rate? Also, how are the other read shapes quantitatively affected?

- Chunk shape F – xt chunks [430,1,1,430]. This will be faster than default chunking for xt reads, but how much faster? How does this affect xy and xyt reads (xt contiguous, y next fastest varying dimension)?

- Chunk shape G – xyt chunks [430,1,430,430]. X-y-t reads are contiguous in this chunking scheme, as are x-y reads, with x-t reads having smaller strides through the file compared with the contiguous data. Therefore, how will it affect xy and xt reads?

Along with the chunk shapes, three different read shapes were used. These are outlined below:

- x-y reads – a sequential read along contiguous data in an unchunked file. This is a common read pattern when analysing data from atmospheric models. It also provides the baseline read rate (from unchunked files) for comparison.

- x-t – a strided read in unchunked files. The expected performance will be poor from the unchunked files but better from the x-t chunked files. This is the read pattern from the space-time spectral analysis case study in Chapter 6.

- x-y-t – this is a larger read than the x-y read, so from unchunked files the read rate should be lower than the x-y read (as shown in Chapter 3 for serial netCF4-python reads). Analysis based on this read pattern could be to do with time analysis on maps, or latitudinal averages on x-t reads.

It is useful to visualise the comparison between the chunk shape and the read shape in order to estimate how the read will perform. This is shown in Figure 4.2, depicting the four-dimensional data in two dimensions. For each grid, the horizontal axis shows x and y varying, and the vertical axis shows z and t varying.

All the tests in this chapter were run on JASMIN, using Python and the netCDF4-python library. The file tested on was [430,430,430,430] (t,z,y,x) giving a file size of 255GiB for all chunk shapes except chunk D which had a file size of 277GiB. The increase in file size is due to the chunk shape in D not being a multiple of the file dimensions, which causes the file to take up extra space on disk. Along with the read rate, the balance between wall time and CPU time (measured by the Python `clock` function) was measured to gain more information in a similar way to in Chapter 3 (see Section 3.1.6 for details on the CPU time and `clock`).

4.1.2 Compression performance

For the compression performance testing, real data was used from a high resolution climate simulation. The reason for this was that randomly created numbers would likely not compress as well as real data – real data has patterns in it. An uncompressed version of the file was used to provide a baseline comparison with the compressed files, for compressed size and read rate. Three files were created using the NetCDF4 deflate compression algorithm.

An argument can be passed to the algorithm to define the level of compression from 1, being the least compressed and the quickest, to 9 being the most compressed and slowest. The three files were created using deflate levels 1, 4 and 9 to cover the whole range. The different deflate levels were not expected to make any difference in the size of the compressed file (Zender, 2016), but were created to repeat the results. A file was also created using the Bit Grooming method described in the Section 2.6.4, which should give a smaller file than when using purely the deflate algorithm. The most extreme level of significant digit reduction was used to test what would be expected for the best compression possible.

A similar test set up was used as for the chunking testing with multiple read patterns and chunk shapes, but with a reduced number of chunking specifications. Only default chunks and x-y chunks were tested to compare between the performance when no chunking decision had been made (giving default chunking) and when the chunking had been designed for a specific kind of access, maps in this case. Three different reads were tested: x-y reads, x-y-t reads, and x-t reads. The x-y would give good performance from the x-y chunked files, used

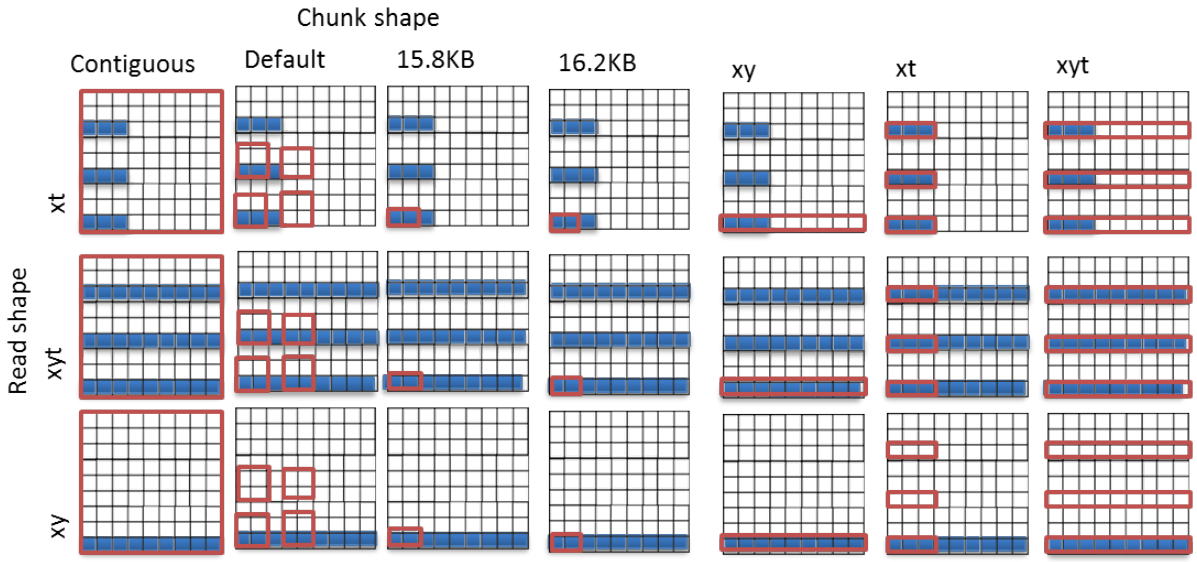


Figure 4.2: Visualisation of the reads compared to the chunk specifications. The read shapes are down the left hand side and shown by the blue shading, and the chunk shapes are across the top shown by the red boxes. Only the first read and the first chunk is shown. For each image, the axes contain two varying dimensions, the horizontal axis shows x varying then y, and the vertical axis shows z and t varying. As an example, consider xt reads from the default chunk shape. As depicted in the diagram the data has dimensions [3,3,3,3], the first xt slice can be described by the slice [0:3,0,0,0:3], where colons denote a range of values, where 0 is the first index. The xt read can then be pictured as shading all the data for all x values and t values, for the first y and z. The default chunking shape is a fraction of each dimension, as an example here the chunk shape is [2,2,2,2] – not a realistic representation of default chunk size, only used for the example depictions here. The first read is shown by the blue shaded region and the first chunk is shown by the red boxes. The effectiveness of the read can then be predicted by looking at the overlap, in this case the read rate would be poor.

to compare the compressed performance to the uncompressed performance for reads with a good read rate. The x-y-t read and x-t read were used to assess whether compression could provide increased performance for those reads.

4.1.3 File conversions

The final section of investigation in this chapter is to measure the time to convert from one chunk shape to another, and from compressed to uncompressed files. The reason for this was that in an overall workflow it may be faster to convert from one chunkshape to another before reading the file, rather than read from an ineffectively chunked file. To evaluate this the time to convert is needed. The copying was done using the nccopy utility[2].

Only a subselection of the testing files in this chapter was used to get a general idea of the time to convert, which were: unchunked files, default chunked files, x-y chunked files, and x-t chunked files. Converting between every different combination of files would have been too time and computing resource intensive.

4.2 Chunking performance

This section measures the read rate for different read patterns from files with different chunking schemes.

4.2.1 Results

Figure 4.3 shows the resulting read rate from the combinations of read rate and chunk shapes shown in Figure 4.2. As expected, when the chunk specification and read pattern match, the read rate is higher i.e. xt-F, xy-E, and xyt-G. The rate for xy-A is particularly high because the xy read pattern is optimal through the contiguous file. The rate for xyt-G is not as high because of the larger read – the serial netCDF4-python testing showed that the performance drops at buffer sizes above 1MiB and the buffer size here is 636MB.

For the x-t read, all the reads were poor except the matching chunk shape, confirming that this is a particularly poor read pattern, and hard to compensate for. Chunk shape F gives poor performance for the xy reads, because of the read shape badly matching the chunk shape, as shown in Figure 4.2.

[2]http://www.unidata.ucar.edu/software/netcdf/workshops/2011/utilities/Nccopy.html

Chunk shape C, the 15.8 KB chunk, improves the xt reads compared to the unchunked file, but not by much – approximately 3 MB/s. For chunk C and D the xy read pattern has better performance than the xyt and xt reads because the xy read is more contiguous on disk than the other reads for these chunk shapes. Chunk shaped D performs worse than C in general, with less difference seen for the x-t read.

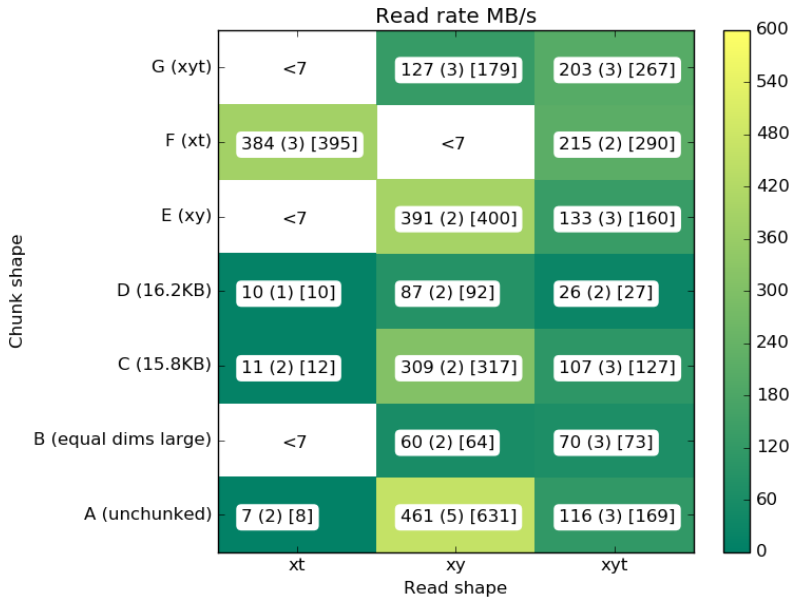
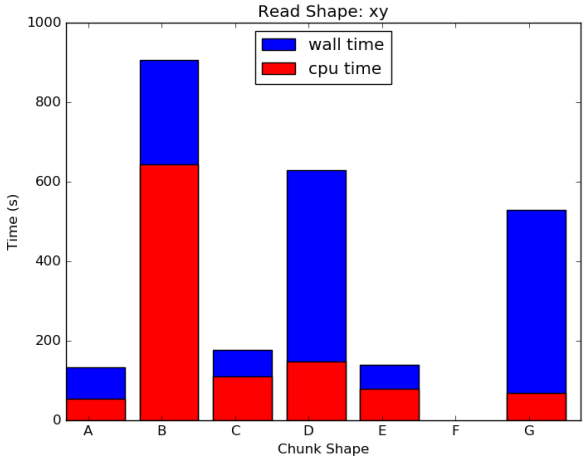


Figure 4.3: Read rate for different combinations of chunk shape and read pattern with no compression. The colour and the first number in each box shows the mean for the tests, the number in () shows the number of repeats which finished, and the number in [] shows the maximum read rate from the test. White boxes with < 7 show where the test did not finish due to a very slow rate - the rate at this time limit would have been 7 MB/s.

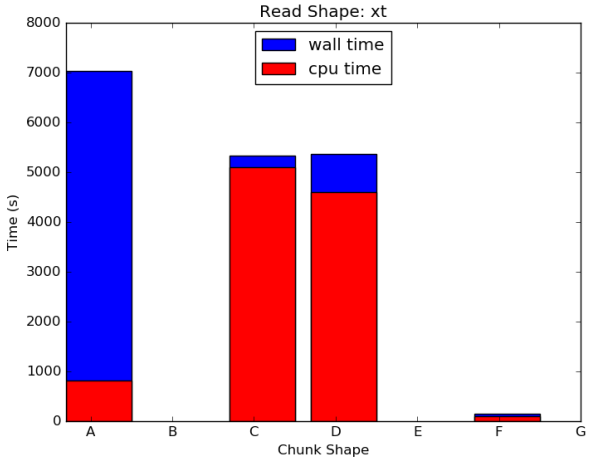
CPU-wall time balance is shown in Figure 4.4. For x-y reads the fastest reads are from chunk shapes A, C, and E with approximately one third to half of the time spent in CPU time. For chunk shape B, the time is significantly higher because of the mismatch between read and chunk shape, and the time is spent mainly in CPU time. For chunks D and G, the increased time for the read is shown as non-CPU time. In the case of D this is due to the chunk being mismatched with the stripe width, therefore increasing the I/O latency.

For the x-t reads, chunk shape A shows only a small proportion of CPU time because of the heavily strided read through the file. C and D show a much higher proportion of CPU time than the x-y reads because the read spans many more chunks.

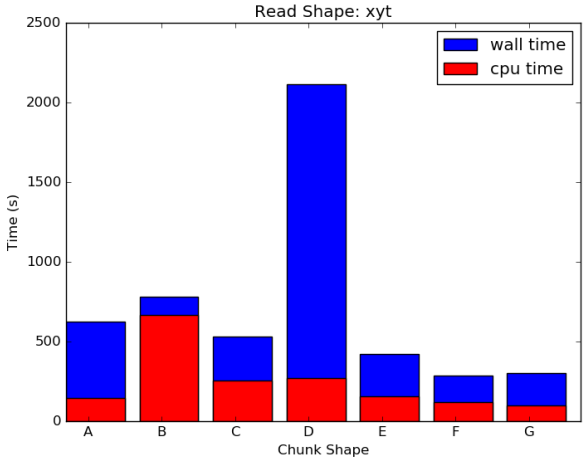
For x-y-t reads with unchunked, xy chunked, xyt chunked, and 15.8KB chunks the CPU time is less than half the wall time, and these all have relatively low times compared to the other chunk specifications. For the default style chunk shapes, the CPU time is a much higher



(a)



(b)



(c)

Figure 4.4: Wall time - CPU time balance for xy (a), xt (b), and xyt (c) reads for all chunk shapes. Y axis shows time (lower is better).

proportion of the wall time – this indicates that when the chunk shapes are badly designed it can have a significant impact on the read rate, which is CPU side rather than caused by the file system – as expected. The comparison between the 15.8KB chunk and 16.2KB chunk shows the impact of having chunks which fit well, or not, into the Panasas stripe width. The overhead of this is very high; the wall time is significantly higher than the CPU time for this result, meaning that the program was having to wait for the I/O showing that the limiting factor is in the filesystem – in this the 16.2 KB chunks create a mismatch between the read sizes and the filesystem striping.

4.2.2 Discussion

In general, the results from this chapter are qualitatively consistent with respect to a priori expectations. The results from this chapter show that when increasing the number of dimensions the results from Chapter 3 still apply, when taking into consideration whether the reads are sequential or not.

Table 4.1 summarises the results from this investigation.

Read shape	Best chunk shape	Effect on rate	Worst chunk shape	Effect on rate
xy	xy	20% reduction	xt	$> 50\times$ worse
xt	xt	$48\times$ better	default/xy/xyt	>10% decrease
xyt	xy	$2\times$ better	16.2KB	$4\times$ worse

Table 4.1: Summary of the chunking read rate results. The effect on the read rate for best and worst chunk is compared to the unchunked read rate.

The results from this section compare well to the advice from the NetCDF4 blog (Rew, 2013) – using small chunk sizes does indeed improve the read rate for multiple access patterns while reducing the rate of previously effective read patterns, the results from Rew (2013) show a much larger speed up than what has been measured here. This could be partly because the slow reads are not as bad on the JASMIN parallel file system as on a single HDD. For large data sets, reducing the speed of previously efficient reads may not be a good solution to speed up reads from non-optimal read patterns.

The results investigating the wall time and CPU time balance (Figure 4.4), show that depending on the combination of read and chunk shape in the file, the time increase is either spent as CPU time, or waiting for the I/O system (majority wall time). With the unchunked files, for slow reads, the dominant factor is the wall time. The reason for this being that the program is waiting for I/O, caused by the striding pattern of the read through the file. The

wall time is the dominant factor for other reads when the read is from a single chunk or the entirety of multiple chunks for the same reason, i.e. xy reads from C, D, E, and G (more prominent in the slower reads); xyt reads from C, D, E, F, and G. The CPU time is the major factor where the read is from partial chunks which also spans multiple chunks, i.e. xy reads from B; xt reads from C, and D; xyt reads from B. This therefore needs to be taken into account when deciding on the chunk shapes for a file.

For the xyt reads and xy reads from files with the chunk specification C and D (15.8 KB, and 16.2 KB chunks respectively), the decrease in read rate was shown in the wall time rather than the CPU time. This indicates that the reduction in read rate is indeed due to the mismatch between the chunk size and the stripe width. The increase in read time for C and D with the x-t reads is due to only part of the data from each chunk being needed for each read over multiple chunks, as discussed above.

4.3 Compression performance

This section measures the compression ratios and read performance from NetCDF4 files when using the deflate compression algorithm, and when using the bit grooming method to improve the compression ratio.

4.3.1 Results

Table 4.2 shows the achieved compression ratios with the NetCDF4 deflate algorithm, and when the data has been compressed in conjunction with the bit grooming algorithm. The highest level of bit grooming was implemented to show the best compression which can be achieved – in reality the level of bit grooming would need to be decided upon for each individual workflow depending on the analysis. The compression ratio for all the of the NetCDF4 files compressed without bit grooming achieved around 23% compression with the resulting size being about 60GB. Bit grooming enable almost double the amount of compression, reducing the file size down to 39 GB.

Figure 4.5 shows the read rate for different combinations of read shape and compression level for an x-y chunked file. As expected from the chunking results, the uncompressed file reads have the same pattern as the chunking results with the xy read being fastest and xt being slowest. When the file is compressed the read rate drops. The different levels of zlib

Table 4.2: Observed compression ratio for the different compression methods used. Chunking was left at default. Compression ratio displayed as the percentage of the original file size (compressed/uncompressed x 100).

Compression Type	File size (GB)	Compression Ratio (%)	Resulting Size (GB)
No compression	275	100	275
Deflate lvl 1	275	23	64
Deflate lvl 4	275	22	61
Deflate lvl 9	275	22	60
Bit Grooming nsd = 1	275	14	39

compression do not affect the performance, because the uncompression algorithm is the same in each case and the size of the compression file is similar, therefore reading the same amount of data from disk. The read rate for the bit groomed files was similar if not slightly lower than the standard zlib compressed files. A similar pattern is seen with the compressed default chunked data (Figure 4.6), with reduced read rates when the file is compressed, however the reduced read rate meant that more tests had a read rate lower than 7 MB/s and did not finish.

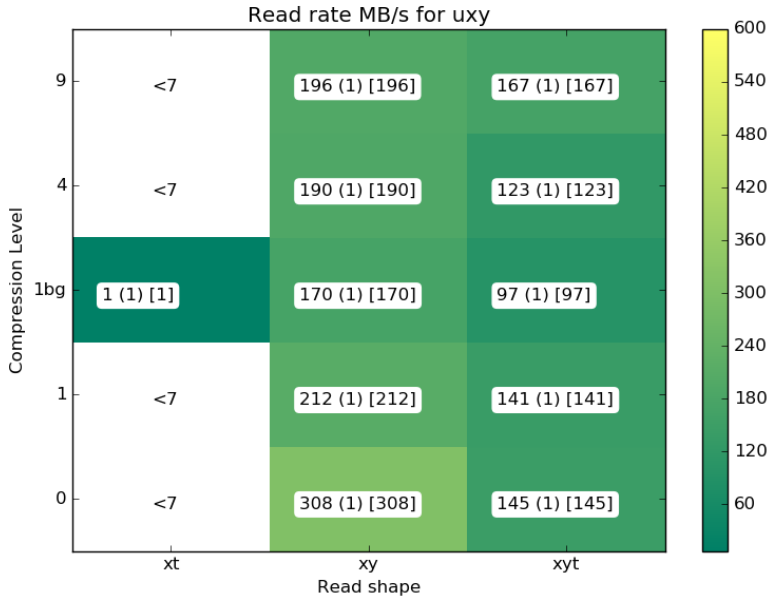


Figure 4.5: Read rate for x-y chunked file with different read shapes for different levels of compression. The colour and the first number in each box shows the mean for the tests, the number in () shows the number of repeats which finished, and the number in [] shows the maximum read rate from the test. White boxes with < 7 show where the test did not finish due to a very slow rate - the rate at this time limit would have been 7 MB/s or less.

The comparison between the wall and CPU time for the tests which finished are shown in Figure 4.7. The compression level does not make a significant difference to the read time so is not shown. For the x-y reads from x-y chunk shaped files, the increase in time is shown to be in the CPU time, so not due to the file system. For the xyt reads from xy chunk shaped files

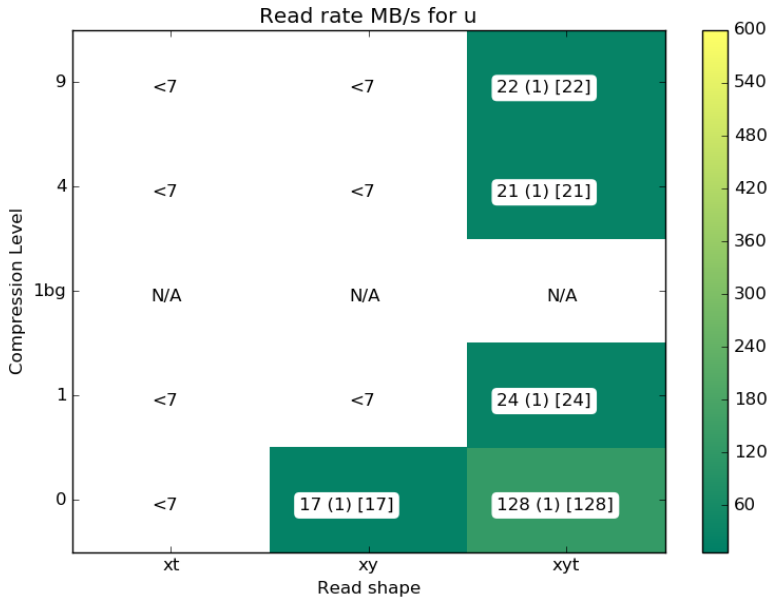


Figure 4.6: Read rate for default chunked file with different read shapes for different levels of compression. The colour and the first number in each box shows the mean for the tests, the number in () shows the number of repeats which finished, and the number in [] shows the maximum read rate from the test. White boxes with < 7 show where the test did not finish due to a very slow rate - the rate at this time limit would have been 7 MB/s. N/A shows that the tests were not run for the bit groomed files with default chunking.

the CPU time is more similar. With the xyt reads from the default chunked files, the wall time is about 4-5 times higher for the compressed file than the uncompressed file, mainly shown in CPU time.

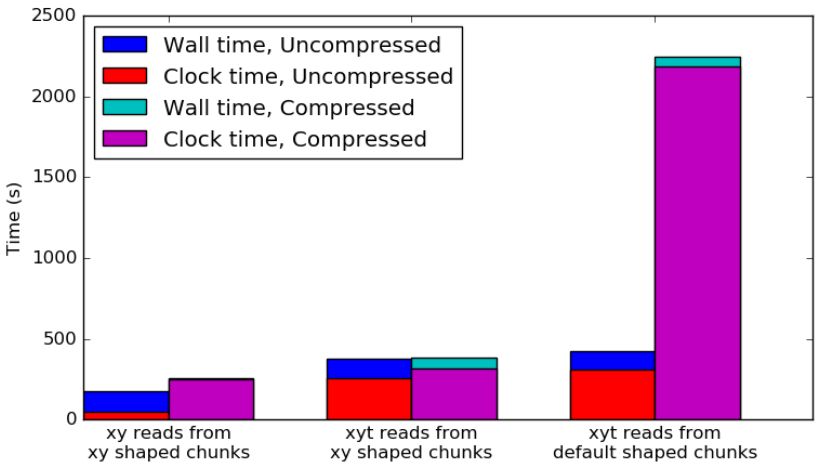


Figure 4.7: Wall time - CPU time balance for reads from compressed and uncompressed files. X-axis shows which read and chunk shape was used. Note, bars overlap

4.3.2 Discussion

The effect of compression on the read rate is summarised in Table 4.3.

Table 4.3: Summary of the compression results from this chapter.

Chunk shape	Read shape	Compression effect on the read rate
xy	xy	30% reduction
xy	xt	DNF
xy	xyt	no observed effect
default	xy	>50% reduction
default	xt	DNF
default	xyt	85% reduction

The compression ratios from the results in this chapter were similar to results from Zender (2016). In all cases, reading from the compressed files slowed down the read, and the smaller file size of the bit groomed files did not significantly affect the read rate. When reading x-y and x-y-t slices from the file, the additional time due to compression was small compared to when reading from the default chunked file, where the the time is approximately 4-5 times longer from the compressed file. The reason for this is the entire chunk need to be read to be uncompressed. With the default chunking, the required data from the chunk is only a portion of the file, compared to the xy chunks where the entire chunk is required; meaning that for the default chunk, time is wasted reading and uncompressing non-required data.

The read rate for x-y reads from x-y chunked files is lower than the uncompressed files by around 100 MB/s. The argument above for the reduced read rate for default files does not apply here, because the whole file is used; the extra CPU time for the compressed read must be purely from uncompressing the file. The x-y-t read from the x-y chunked file is slower because the read spans multiple chunks – therefore striding through the file

Comparing the different NetCDF4 deflate (zlib) compression levels, it is unsurprising that the read rate is the same from files with each level of compression considering the files were the same size. Also, the uncompression algorithm is the same regardless of the level of compression, so the CPU overhead to uncompress would be the same for each case.

4.4 Layout conversion

A possibly important part of a workflow could be to convert from one chunking format to another - as shown in the previous sections, both compression and chunking can provide very

poor read rates. This section measures the time taken to convert from one chunking scheme to another.

4.4.1 Results

Table 4.4 shows the time to convert between different chunk shapes. The rate of conversion between default chunking and an unchunked file is high compared to the other results. The conversion rate from the compressed default files is lower, but still the second highest rate. As expected from the low read rate of x-t reads from x-y chunk shapes, the conversion between x-y chunks and x-t chunks is very low. It is significantly faster to convert from x-y to unchunked, then unchunked to x-t.

Table 4.4: Effect rate of conversion between different chunking formats.

Start	End	Method	Mean rate (MB/s)
Unchunked	Default	nccopy	22
Default	x-y	nccopy	12
Unchunked	x-t	nccopy	25
x-y	Unchunked	nccopy	24
Default	Unchunked	nccopy	178
x-y	x-t	nccopy	< 1 (Job killed after 64 hours)
x-t	Unchunked	nccopy	27
Default compressed	Unchunked	nccopy	28
x-t compressed	Unchunked	nccopy	11

4.4.2 Discussion

The results from this section enable an estimate for conversion between chunking schemes. Another interesting result from this section is that it is faster to convert an x-y chunked file to an unchunked file, then to an x-t chunked file, than it is to convert the x-y chunked file to an x-t chunked file – by at least 12 times (x-y to unchunked 24 MB/s, but double the amount of work so twice the time, giving 12 times faster). This indicates that the nccopy utility is not doing the conversion in the most efficient way. It is also faster to convert a default chunked file to another chunk specification via an unchunked file.

Similarly to the previous section, conversion from compressed data takes significantly longer - a compressed default chunked file to unchunked uncompressed file, takes 6-7 times longer than when starting with an uncompressed version of the file. This is due to having to read and uncompress the whole whole chunk to access a subset of it.

4.5 Conclusions

The effect of chunking and compression on the read rate is summarised in Figure 4.8. This extends the figure from the previous chapter showing the NetCDF4-python read rate results. The main points from this figure are that if the file is chunked and the reads match the chunk shape the performance is comparable to sequential reads from an unchunked file – this is not surprising as reading the entire chunk is effectively a sequential read from the file. Also, if the file is compressed and the whole chunk is required, the performance is similar to the unchunked, uncompressed sequential read.

The alternative cases are when the file is chunked and the data required for the algorithm is only a partial chunk. In the uncompressed case there is an overhead in the library because the entire chunk is not required. The performance is similar to the non-sequential read performance from an unchunked file, determined by the read size. When the file is compressed and the entire chunk is not required, the library still has to read and uncompress the entire chunk, which has a large impact on performance. Despite not quantifying the exact performance reduction of these two cases, the important point for this work is that it detrimentally affects the read performance – in the compressed case, significantly.

Figure 4.8 also includes the time taken to reformat data if the read is not optimal for the data's chunk specification. The exact additional time depends heavily on the work flow, so whether it would be worth doing would need to be determined for each workflow individually. In general the more times the data is read in an inefficient format, the more useful reformatting the data would be.

Returning to the aim of this chapter:

- Quantitatively evaluate the effect that chunking and compression have on the read rate, and therefore the overall workflow.

The results show that, at least for the experiments in this chapter, an acceptable compromise with the chunk size is not found. The example read patterns were chosen to represent two different analysis types in atmospheric science, one looking at analysis of maps (xy read), and one looking at time series analysis (xt read). The third read xyt represents where the secondary direction of analysis in each example scenario is along the t and y dimensions respectively. These results show that reading a larger section of data from the file where the secondary dimension of analysis is pre-emptively read from disk, could be a good solution to achieving

a good read rate within analysis code.

An interesting result is that when the chunk shape and read shape are not matched, and the read is spread over multiple chunks the additional time is shown as CPU time rather than in the file system. Contrarily, when the read is either from non-chunked data or from a single chunk the additional time is shown as non-CPU time (wall time much higher than CPU time), indicating that an improvement in the file system speed would increase the performance.

The investigation into conversions between chunk shapes in files, show that going from compressed default to unchunked, then unchunked to a different format could be good strategy compared to direct conversion between formats. Converting to another chunk shape would give a good read rate for an application. Of course this means there would be some overhead involved, but if the read is executed more than once, for instance when developing an analysis script, then the total overhead of this one read would be small. This approach would also be beneficial when intermediate storage was used such as with burst buffers, or if the workflow included adaptive reformatting of data. This method would be even faster if the initial data was uncompressed on disk, but because of the size of data this may not be possible.

In terms of analysing very big data, the fastest read rate is desirable. Because of this, using chunking to compensate for different read patterns is not a good solution because it reduces the read rate, while not significantly improving the read rate for multiple different read patterns. On the current JASMIN platform, the best way to provide a high read rate for multiple read patterns in serial, would be to have multiple versions of a file. Obviously, this significantly increases the cost of storing data, so is not a good solution. Two options to improve the read rate could be used: adaptively creating new version of files which provide improved access speed, whilst only storing one version of the file; or to use a burst buffering system.

Some of the results here show that the performance is significantly reduced depending on the read pattern and whether the file is chunked or compressed. However, the reduced performance could be compensated for by implementing the workflow in parallel. The next chapter quantifies the parallel scaling which would be seen by a typical user on the JASMIN super-data cluster, so that experiments can be undertaken in the following chapter combining all the factors that have been investigated in this thesis. This is to determine whether the reduced performance of chunking, compression, and inefficient read patterns can be compensated for by executing the workflow in parallel, and what level of parallelisation is required to achieve this.

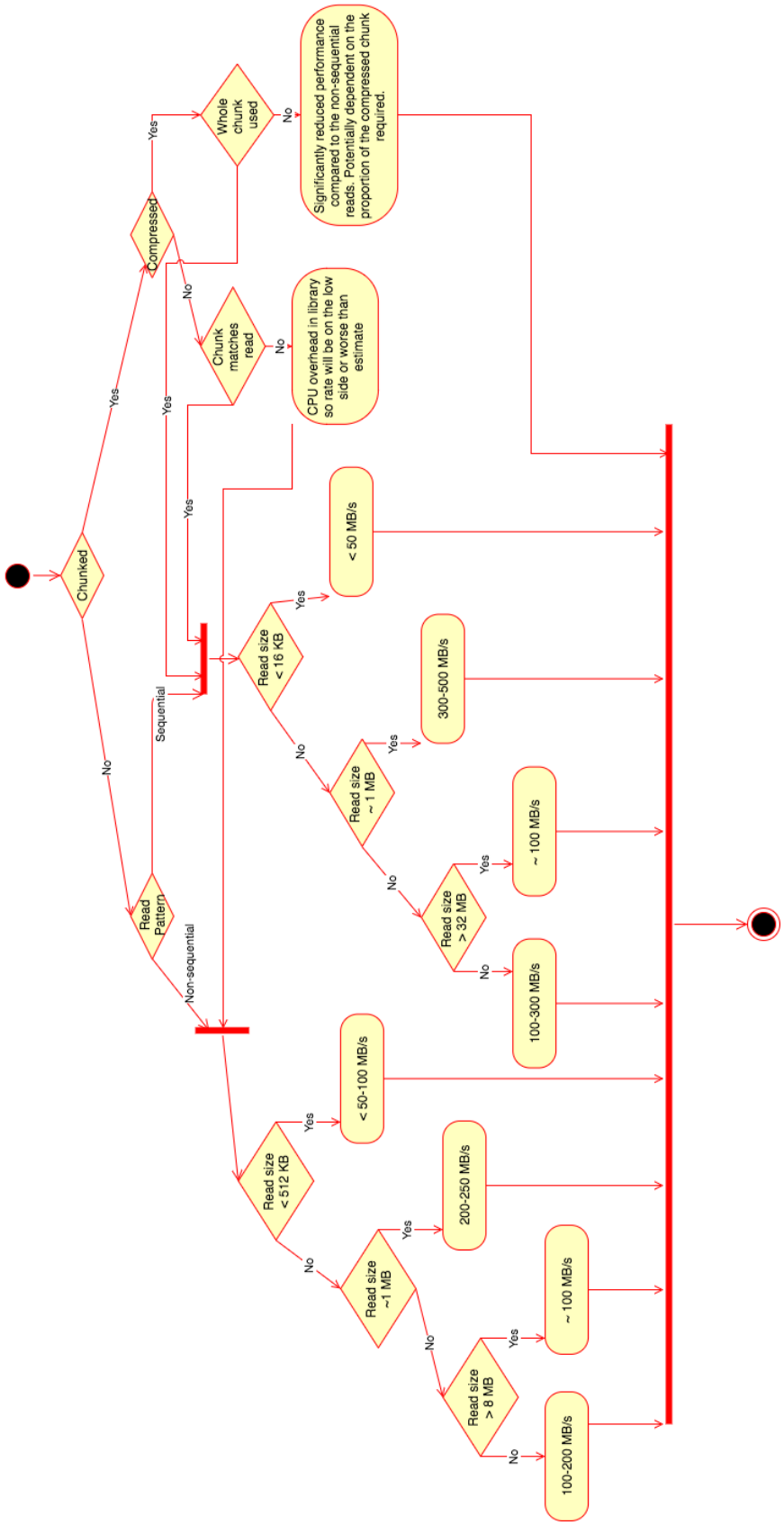


Figure 4.8: Summary of the performance results from the chapter, extending the results shown in Figure 3.12. The results from this chapter showed an similarity with results from Chapter 3, in that: where the whole chunk is read the read rate was similar to the sequential reads from an unchunked file, and were a partial chunk is read the read rate was similar to the striding read rate although towards the low side of the estimate.

Chapter 5

Parallel Reads from a NetCDF4 File to Improve Read Performance

By using parallel techniques, it is possible to increase the read rate by utilising multiple network interface cards (NICs) on processing nodes, therefore reducing the overall read time for an application or workflow. This could compensate for reduced performance due to chunking, compression, and mismatched read patterns. In order to make sensible parallelisation decisions, it is important to understand how the parallel read rate scales for real workflows. The JASMIN super-data cluster will be used as an exemplar HPC analysis platform, to quantifiably investigate the parallel scaling of workflows using NetCDF4 files.

Figure 2.1 showed the scaling of the JASMIN platform using IOR to measure the total bandwidth of the storage. This is obviously useful from the standpoint of assessing total system performance and comparing file systems. However, this is not useful from the standpoint of a user designing an application – the data they are using might only be stored on a single shelf as is the case in this chapter (see Appendix A for details on the JASMIN platform). In addition to this, there are other factors which could affect the performance of the application so this peak system performance may never be reached. In order to assess what factors affect the performance of a parallel application (as was done with a serial application in Chapters 3 and 4), the typical parallel scaling of the platform needs to be assessed – not the peak system wide performance – therefore, the aims for this work are:

The aims of this chapter are to:

- Assess the parallel scaling of the JASMIN super-data cluster for application realistic reads.
- Investigate whether MPI collective reads can improve non-sequential read performance on JASMIN.

The results from the NetCDF4 performance serial read chapter, showed that the peak performance with a single task was around 500 MB/s for binary files when using C and Python, and when using C to read NetCDF4 files, and around 300 MB/s for NetCDF4-python. However, the theoretical bandwidth to a single node is 1250 MB/s (Lawrence et al., 2012). Therefore, to determine whether the read rate to a single node can be improved, multiple cores were utilised for the first part of the testing. The second part of the testing measures the scaling for parallel reads across multiple nodes for typical analysis scripts. In both cases, testing was conducted using C and netCDF4-python because of the difference in performance observed in Chapter 3.

Figure 5.1 shows three different approaches to reading when using an HPC architecture. The first shows a serial read from the parallel file system. The second shows a parallel read either using multiple, independent tasks, or using MPI-IO in independent read configuration. The third approach depicts how a collective read using MPI-IO works. MPI-IO can use collective, or cooperative, I/O which means they can take advantage of a two phase read strategy (Li et al., 2011), where the data is read from disk in an efficient way, then redistributed among the processing nodes for the application (del Rosario et al., 1993).

As an example of the different read strategies in Figure 5.1, imagine three files each with red, green, and yellow blocks. To process the data, each of three nodes requires a single colour of block. In the independent read, each node does a partial read of the file. In the collective, two phase read, each node reads the whole section of multiple colours and then rearranges the blocks between the nodes so that each has the required data. The collective read can often be faster in some cases – the example here is too simple to benefit from a collective read.

The tests in the first part of this chapter use an MPI harness to submit multiple jobs concurrently for independent reads, without using a parallel I/O library. Parallel I/O can be implemented using MPI-IO, which can be used through the NetCDF4 C library, but is not implemented in the netCDF4-python library. The second part of this chapter use MPI-IO to test

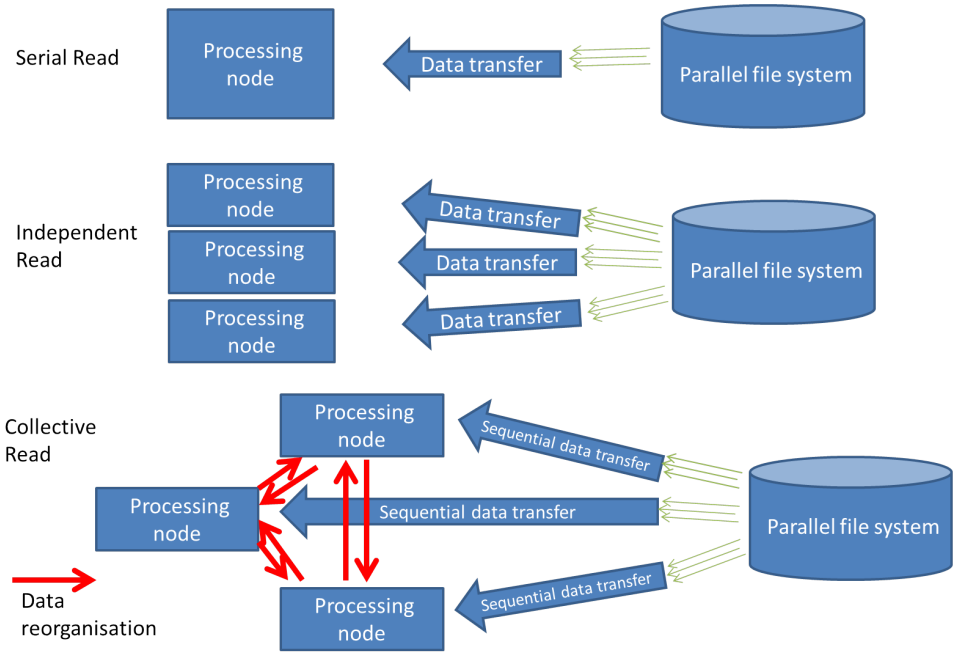


Figure 5.1: Depiction of a serial read, a simple parallel read treating each processing node as independent, and a collective two-phase read using MPI-IO. The Collective read has two stages, the first being a fast sequential read from disk to each node, then reorganisation among the nodes for the required data. The green arrows on this figure represent the fact that each read from the parallel filesystem is not a simple serial read.

independent and collective reads to improve the performance of striding reads.

5.1 Method

Figure 5.2 depicts the testing parameters for the two different investigations in this chapter. The reasons behind the parameters are described in the following sections, along with more detail on the testing algorithms.

5.1.1 Testing data

There are three different sets of data used for the testing in this chapter. These are depicted in Figure 5.3. The first is a number of 256 GB files equal to the number of parallel tests, and large enough so that a single file cannot fit into the RAM of the nodes used in testing. The second, used to test reads from a single file, was just a single 256 GB file. Finally, a set of sixteen, 16 GB files was used to contrast the read rate from reading from a single file, and splitting this file into smaller chunks. All of the files were one-dimensional, unchunked, and uncompressed NetCDF4 files containing random numbers. All the data was stored on the same shelf (see

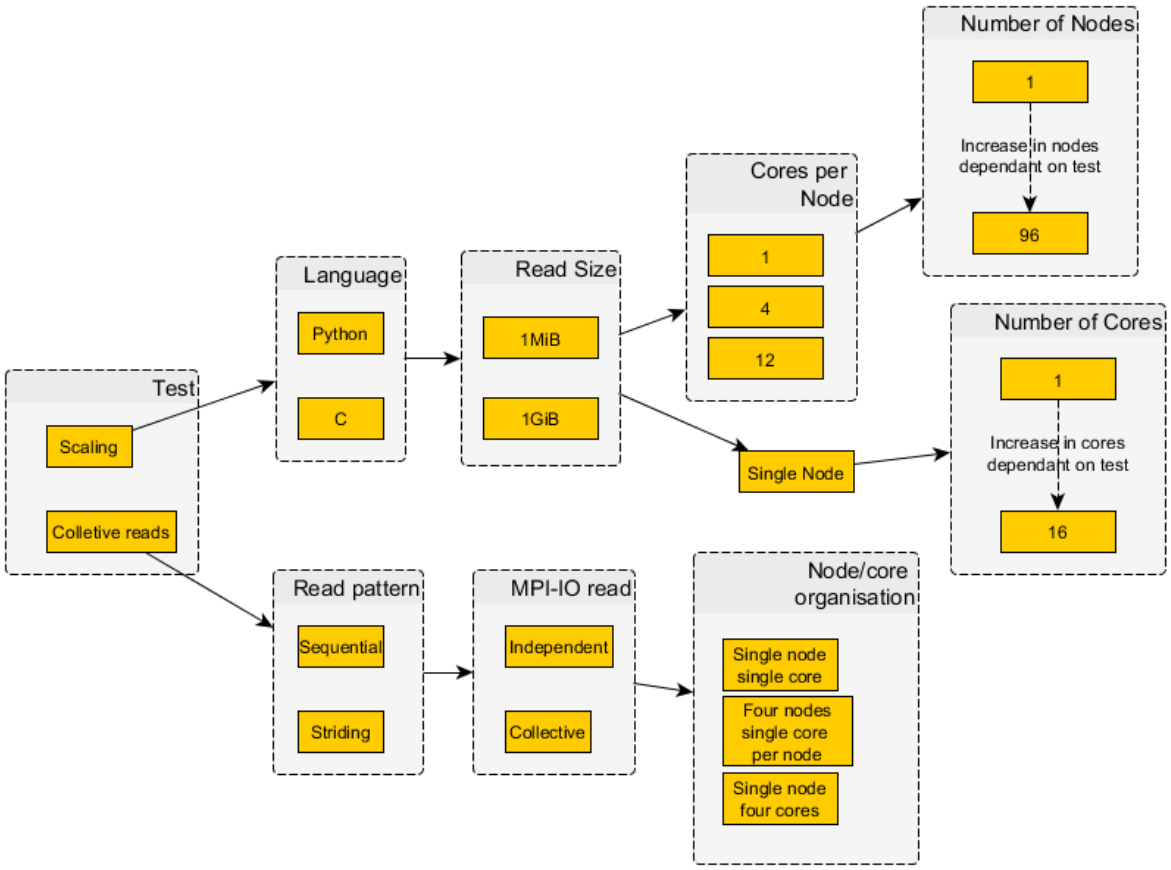


Figure 5.2: Testing domain for this chapter. From left to right is the slowest to fastest varying test variables.

Section A.1), and extrapolation from Figure 2.1 shows the maximum read rate for parallel tests on mulitple nodes is expected to be around 5-10 GB/s.

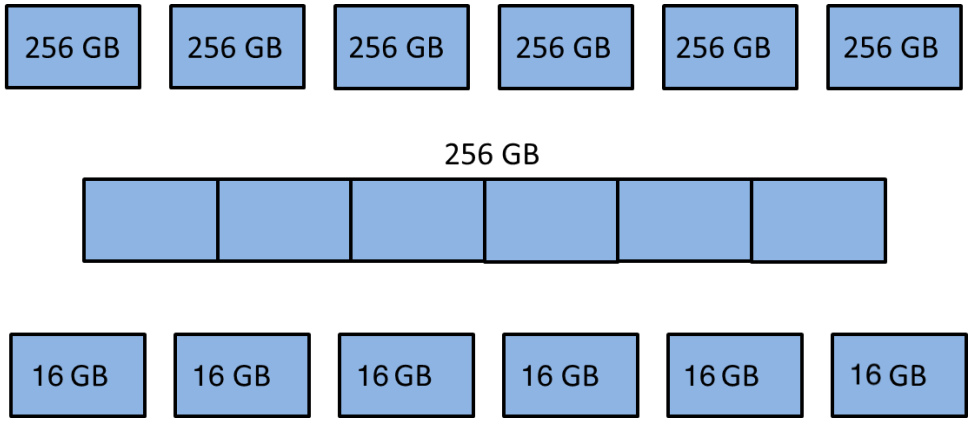


Figure 5.3: Sets of test files for each style of test in this chapter. The examples here show how the data is arranged for 6 tasks. The first row shows six, 256 GB files, the second shows a single 256 GB file split into six pieces, and the final row shows six, 16 GB files.

5.1.2 Parallel scaling

The results from Chapter 3 showed that the peak performance of netCDF4-python was about 200MBs less than the C NetCDF4 library performance. This difference in performance means that investigating the parallel scaling of both libraries was important. The main investigation was whether the same peak performance can be obtained for C and Python, and at what level of parallelisation does it occur.

Another result from Chapter 3 affecting the design of the method in this chapter was the drop in performance for netCDF4-python at larger read sizes above 8MiB. To test any difference in scaling between larger reads with netCDF4-python, 1MiB and 1GiB reads were tested. 1MiB was chosen because it was the peak performance for netCDF4-python, and 1GiB because of the reduction in read rate at this size. The two different read sizes were also tested with C where there should be no difference in performance.

The results from Chapter 3 showed that the theoretical bandwidth to a single node of 1.25 GBs on JASMIN was not reached with any of the tests, with a maximum rate of 500 MB/s seen. To investigate what percentage of this bandwidth can be achieved, single node tests were run. The nodes used for testing had a maximum of 16 cores, therefore giving a maximum for the concurrent tasks on a single node.

To test how competition for node bandwidth affected the performance, different levels of multiple core per node parallelism were used. These were: single core per node, four cores per node, and 12 cores per node. This competition could be from one user, as is the case here, multiple users working from the same storage shelf, or other users working on the same node.

The tests were performed using the same basic read code as for the NetCDF4 serial performance tests. For the parallelisation, a simple MPI harness was used to launch multiple independent instances of the same script, each reading from its own 256 GB file.

In contrast to the NetCDF4 serial tests in Chapter 3, these tests were not run exclusively on the nodes[1]. This was partly due to queue times, but also to give a more accurate representation of the performance of a typical analysis workflow. During testing the queue times were significantly longer when multiple exclusive nodes were requested, which would be an inefficient use of time for a real analysis workflow given that, during initial testing (not shown), the exclusive and non exclusive read rates were comparable.

[1]Exclusive node use means that no other users can run jobs on the node. On JASMIN this is an option within the batch queuing system. This is not always an option on HPC platforms.

5.1.2.1 Measurements

For all results in this chapter the tests were repeated between three and five times.

Performance was quantified by measuring a mean aggregate read rate, which was calculated by taking the average time across all tasks. The rate was then calculated by dividing total aggregate data by average time to give an average rate. Therefore the rate is defined as:

$$\bar{r} = \frac{x_{tot}}{\bar{t}},$$

where $\bar{r}$ is the mean aggregate read rate, x_{tot} is the total data read across all tasks, and $\bar{t}$ is the mean time across all tasks. Strictly speaking, the aggregate read rate would be calculated from the maximum time for each experiment; the total read would not be finished until the longest running task finishes. The mean task was used however because there is inherent variability in the read rate due to many factors (some of which were discussed in Section 2.6).

The aggregate rate is displayed against the number of cores used and the number of nodes used, to give slightly different perspectives on the performance. The individual times for each task were used to calculated the read rate assuming that all the tasks were reading at this rate, which was then used to estimate the range for the results.

The variation in the results were analysed using box plots where the time to completion on each core was used to calculate the aggregate rate which would be observed if all cores were performing at the same read rate. The box plots are calculated using the matplotlib (Python library) `boxplot()` function, showing the median, upper and lower quartile, the range, and outliers in the data.

5.1.3 MPI collective Method

The second section of testing in this chapter was to investigate whether collective reads with MPI-IO would be of beneficial use on JASMIN. To test collective reads, a non-sequential read was needed. The striding read pattern from Chapter 3 was chosen. Sequential reads were also run, so as to get an idea of the overheads involved with running collective reads – collective reads should have no benefit for sequential reads.

Serial tests were run to draw comparisons with the serial read rate results. The striding code reads one section then skips three, so the parallel tests for the collective reads were parallelised using four concurrent tasks to read the whole file. Two different parallel tests were

used, one on a single node using four cores, and another on four nodes with a single core used on each. MPI-IO independent and collective reads can then be compared with the non-MPI-IO reads from the parallel scaling results.

The NetCDF4 C library was used to implement tests using parallel reads (Python was not used because the netCDF4-python library does not implement the parallel capability of the C library). All these tests read from a single file and are tested for sequential and striding reads – it is expected that collective reads increase performance of striding reads, but not for sequential reads.

The sequential read test read a quarter of the file on each core, i.e. the first read the first quarter, the second read the second quarter and so on. The striding tests used the same striding code as for NetCDF4 performance, so read one section, and skipped three. When 4 tasks were running, the first read from the start of the file, the second started from one buffer in and read in the same "skip three" pattern. In this way the whole file was read.

5.2 Parallel read scaling on JASMIN

This section tests the performance and scaling of parallel reads on the JASMIN super-data cluster. The aims of this section are to: test single node parallel scaling, and test multinode parallel scaling.

5.2.1 Results

Figure 5.4 shows the results when running multiple concurrent tasks on a single node.

- For C using the NetCDF4 library reading from multiple 256 GB files, the performance increases up to a maximum of 1200 MB/s at 12 concurrent tasks, then the performance reduces. This is close to the theoretical bandwidth of 1250 MB/s.
- This read rate then reduced at 16 tasks by almost 200 MB/s.
- Testing showed that there was no difference in performance between reads of 1 MiB and 1 GiB when using the C program, as expected, so the results for the 1 GiB buffer size are not shown.
- The 1 MiB read using netCDF4-python have peak performance of approximately 850 MB/s at 6 concurrent tasks. The rate then reduces by about 50 MB/s at 8 tasks, and

continued to reduce.

- Up until 6 tasks all tests from Figure 5.4 performed similarly to each other for the 1 MiB buffer when reading from multiple files, for increased numbers of tasks only the C program reading from multiple 256 GB files showed a higher read rate.
- When reading from a single file, there was no performance benefit obtained through increasing the number of concurrent tasks, the read rate staying at approximately 200 MB/s.

Small files (16 GB) were used to test the scaling for a smaller scale workflow, and compare with the lack of improvement in the read rate for a single 256GB file on a single node. For one to eight tasks, the rate scales similarly to when the 256 GB files were used. Interestingly the mean read rate then drops by approximately 300 MB/s for 16 tasks, however the maximum read rate for 12 and 16 tasks is more similar to when reading from 256 GB files, indicating that this reduction in the mean could be due to slow reads due to competition on the nodes with other users at the time of testing.

For the 16 GB file reads the large range, with high maximum rate for 4 and 8 nodes was due to a single particularly fast and uncharacteristic read compared to the rest of the results for that test instance; this could have been caused by hitting a buffer or cache – harder to avoid with smaller files.

Results using multiple nodes are shown in Figure 5.5. The bars on these graphs show the range (max and min) of the read rate, giving an estimation of the variability of the read, and what range of rates could be expected in real workflows. The results for 1 MiB reads for C and netCDF4-python have similar profiles, with the 1 GiB netCDF4-python reads showing a similar but more slowly increasing rate with increased parallelism e.g. for one task per node with 16 tasks, C and Python for 1 MiB buffers the read rate is around 3700 MB/s, whereas the 1 GiB Python buffer read rate is approximately 2000 MB/s. In all cases, increasing the number of concurrent tasks increases the rate with diminishing returns, i.e. sub linear scaling.

For the C single task per node test, the number of nodes was increased as far as the queuing system would allow, and the rate reached a maximum of 6000 MB/s at 52 tasks. The performance in a number of places for all the results is lower than what would be expected for consistently increasing performance, caused by low rate outliers (shown in Figure 5.7) reducing the average, e.g. C 1 MiB on task per node at 24 tasks. For the 1 GiB netCDF4-python

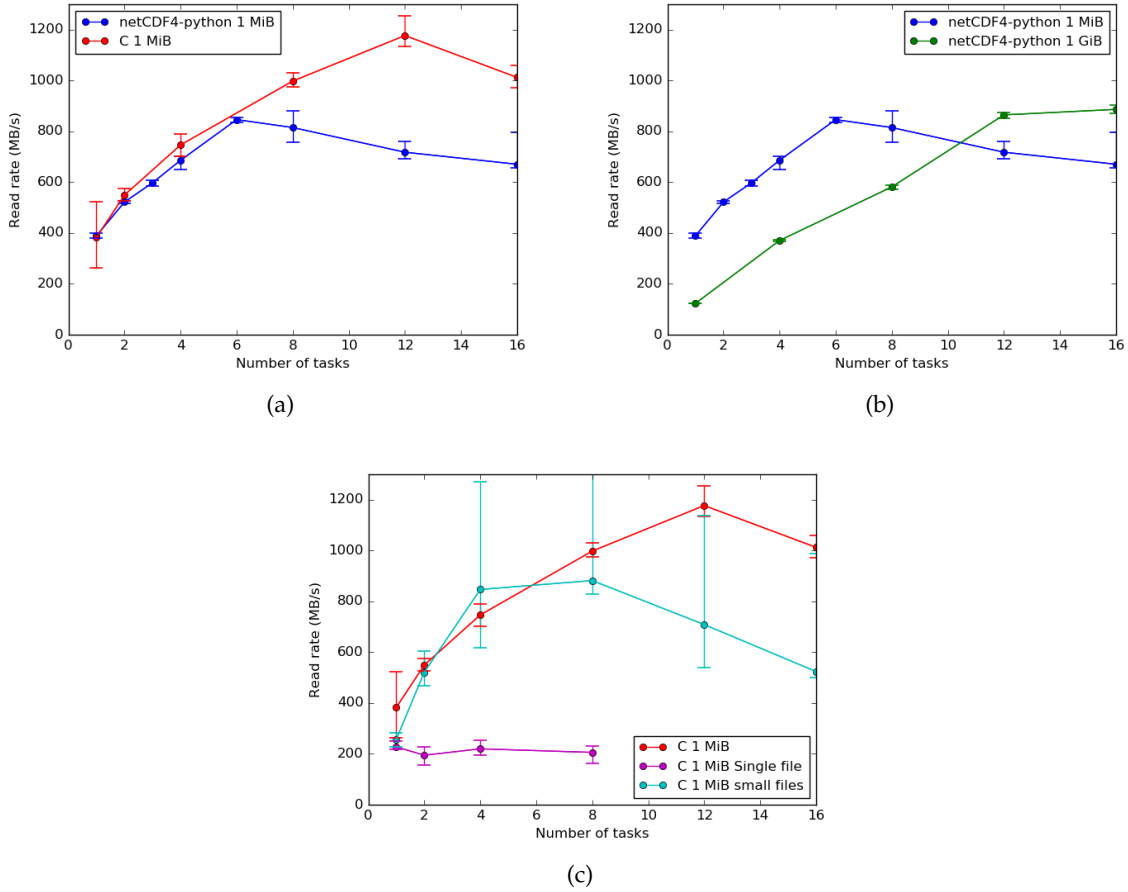
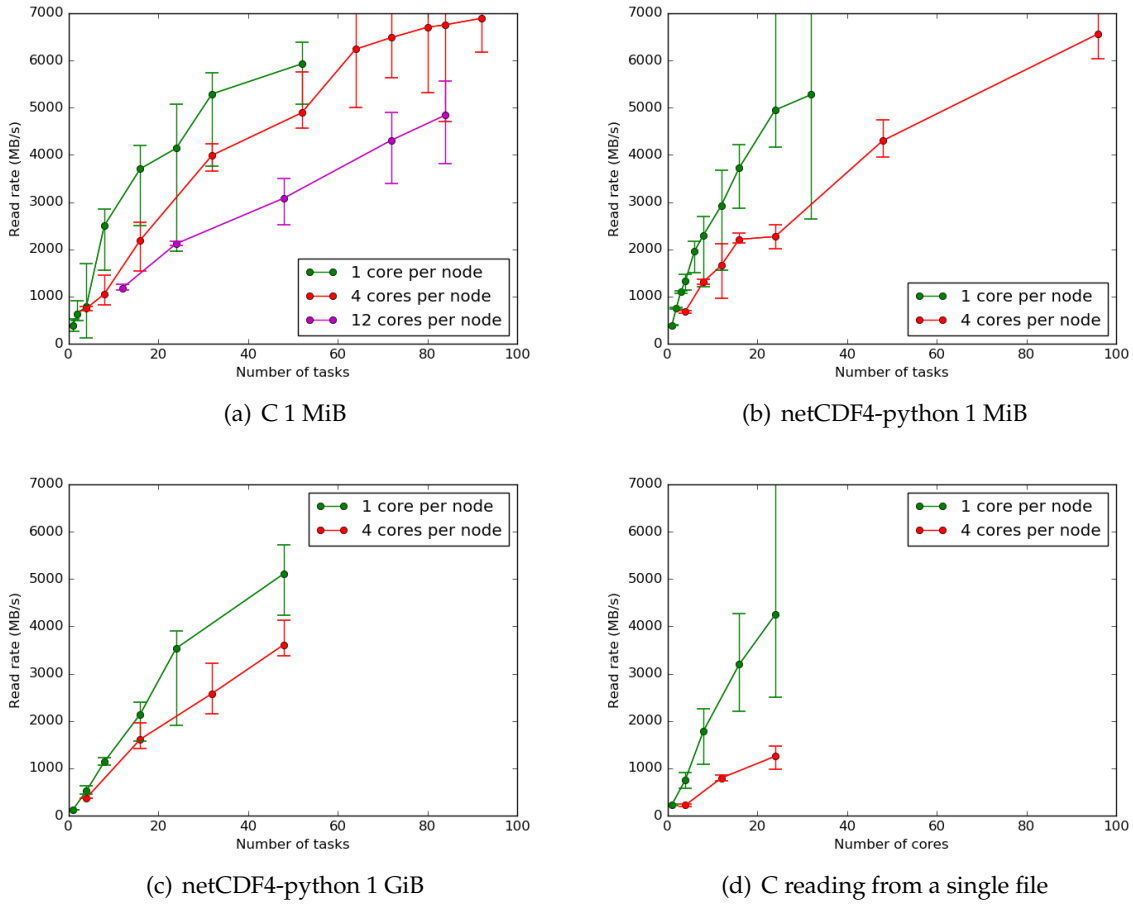


Figure 5.4: Results from the tests conducted on a single node with increasing numbers of concurrent tasks. (a) shows the comparison between C and Python reading from NetCDF4 files with a 1 MiB buffer size. (b) shows the comparison when using Python reading from NetCDF4 files for 1 MiB and 1 GiB reads. (c) shows the comparison using C reading from NetCDF4 files when reading from multiple 256 GB files (red), a single 256 GB file (magenta), and multiple 16 GB files (cyan). The upper and lower whiskers on the bars show the maximum and minimum rates.

buffer tests were not extended past 48 concurrent tasks due to very large queuing times with the batch queuing system, and the performance showing the same pattern as the 1 MiB buffer tests but with a smaller magnitude.

As shown by the results in Figure 5.5, when using four cores per node the read rate scales in a similar way as when using a single task per node, but at lower read rates. Having four cores per node allowed the number of concurrent tasks to be increased up to 92 concurrent tasks achieving a rate of approximately 7000 MB/s (the batch queue did not run jobs with more than around 50 nodes requested). This rate is around what was expected from extrapolation of Figure 2.1. The 12 cores per node speedup also scaled similarly with increasing tasks but with further reduced read rate, therefore, the test was only run for the C 1 MiB reads because of the



(a) C 1 MiB

(b) netCDF4-python 1 MiB

(c) netCDF4-python 1 GiB

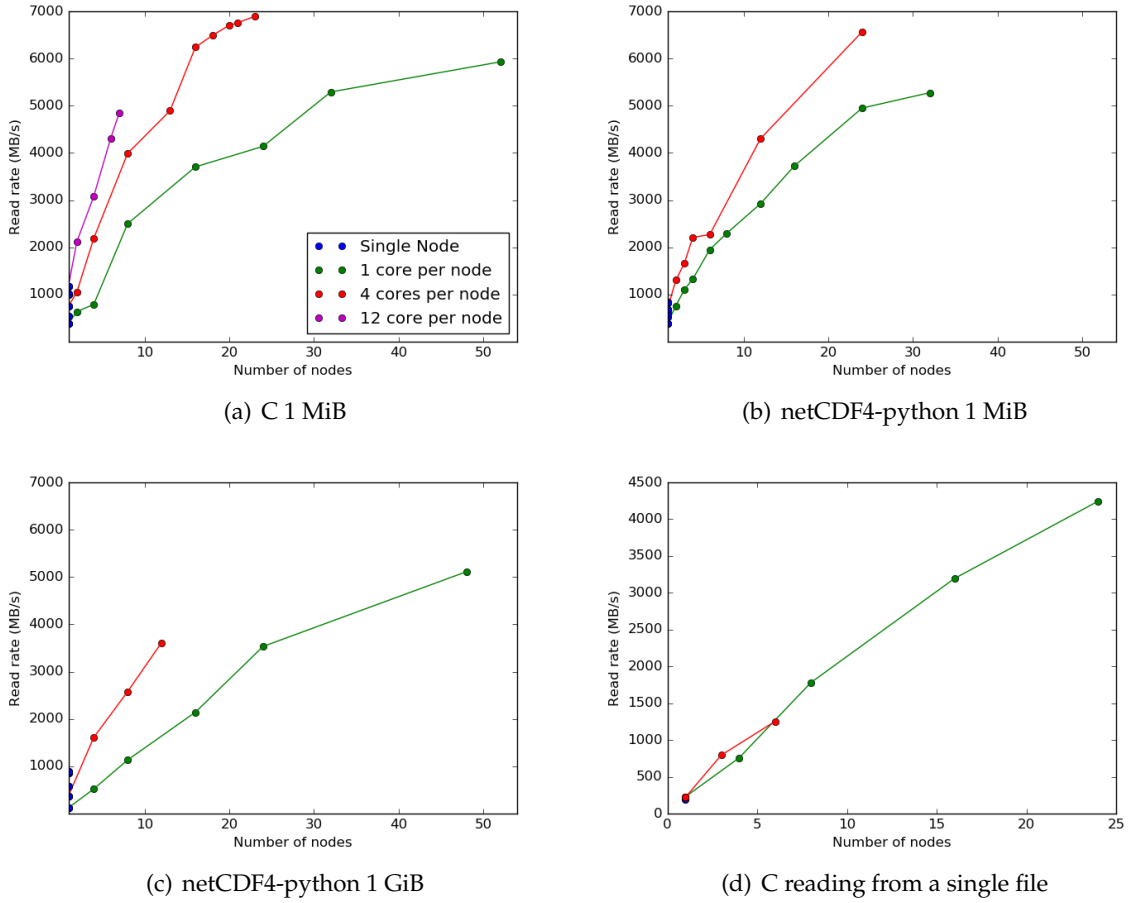
(d) C reading from a single file

Figure 5.5: Read rate results scaling with the number of parallel tasks. Green shows where there was one task per node, red show 4 tasks per node, and magenta shows 12 tasks per node. (a) shows the results when reading from multiple NetCDF4 files using C with a 1 MiB buffer size, (b) shows results when reading using netCF4-python from multiple files with 1 MiB buffer and (c) with 1 GiB buffer, and (d) shows results using C when reading from a single NetCDF4 file. The upper and lower whiskers on the bars show the maximum and minimum rates.

time intensive nature of the tests (long queue times for requesting 12 out of 16 cores per node).

The read rate can also be considered with respect to the number of nodes used rather than the number of concurrent tasks – shown in Figure 5.6. This shows that the rate per node is increased when more cores are being utilised as the parallelisation is increased.

Box plots showing the variation for each set of results are shown in Figure 5.7. For the single node tests, the variation is relatively constant with increasing tasks, compared to the multiple node tests. For the multiple node tests the variation increases with more nodes.



(a) C 1 MiB

(b) netCDF4-python 1 MiB

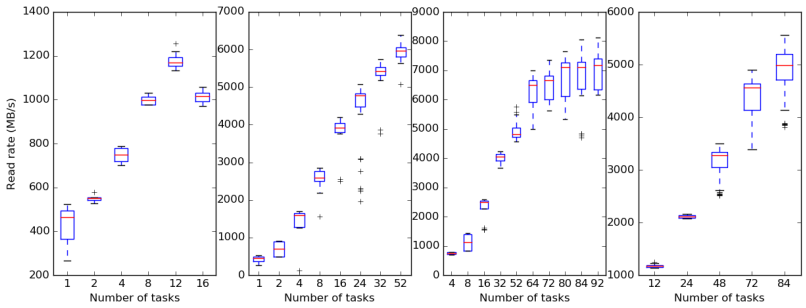
(c) netCDF4-python 1 GiB

(d) C reading from a single file

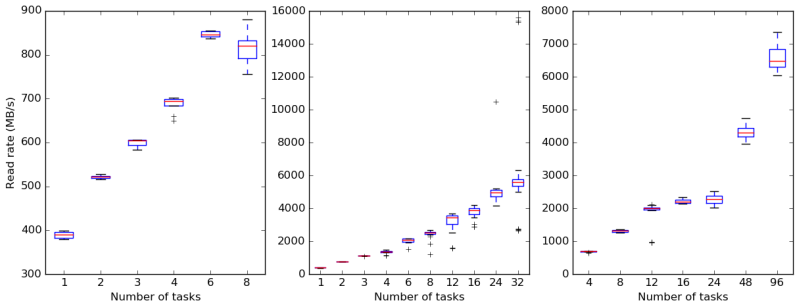
Figure 5.6: Parallel scaling with respect to the number of nodes used. Green shows where there was one task per node, red show 4 tasks per node, and magenta shows 12 tasks per node. (a) shows the results when reading from multiple NetCDF4 files using C with a 1 MiB buffer size, (b) shows results when reading using netCF4-python from multiple files with 1 MiB buffer and (c) with 1 GiB buffer, and (d) shows results using C when reading from a single NetCDF4 file.

5.2.2 Discussion

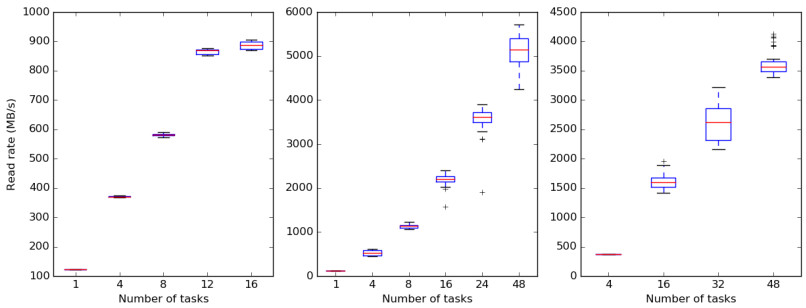
The results show that the netCDF4-python library cannot attain the same total bandwidth as the C library on a single node basis. The netCDF4-python results are limited by the time spent processing CPU instructions (CPU time), whereas the C performance is limited by the I/O speed (shown in Chapter 3). When utilising multiple nodes, the lower maximum rate would only affect the total aggregate read rate as long as the number of tasks per node does not exceed six. For both cases the maximum read rate is achieved before the maximum number of cores per node. This shows that at a certain point the competition on the node out weighs the benefit for further parallelism. Its is also worth noting that the read rate when using 12 cores with the C program utilises 95% of the node bandwidth, showing that almost the whole bandwidth of the node can be used.



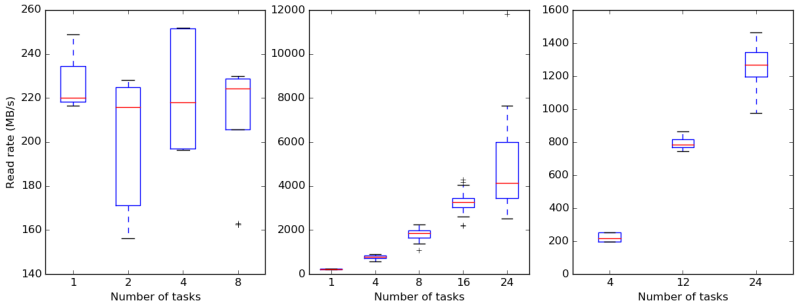
(a) C 1 MiB. Left to right: single node, 1 core per node, 4 cores per node, and 12 cores per node.



(b) netCDF4-python 1 MiB. Left to right: single node, 1 core per node, and 4 cores per node.



(c) netCDF4-python 1 GiB. Left to right: single node, 1 core per node, and 4 cores per node.



(d) Single file reads with C. Left to right: single node, 1 core per node, and 4 cores per node.

Figure 5.7: Box plots showing range of the results used for the averages in the previous result plots. (a) shows the results for C reading from NetCDF4 files, (b) shows the results using 1 MiB reads using netCDF4-python, and (c) uses shows 1 GiB reads using netCDF4-python. The red line shows the median, the box shows the upper and lower quartiles, the whiskers show the range of the data, and the plus symbols show calculated outliers.

When using netCDF4-python and a single node, the maximum read rate was the same for both read sizes, but achieved at different numbers of concurrent tasks. This shows that when working in parallel, the lower read rate for the larger buffer does not affect the maximum achievable read rate.

When considering the utilisation of the node, an important result from the single node testing is that the peak performance is not maintained once the peak has been reached when adding further parallel tasks – caused by competition for resources. This could be important if charged by time for the use of a platform.

Despite multiple node parallelism showing a significantly higher read rate, parallelism on a single node could still be useful. Parallel programs on a single node can be simpler to write because they share memory (no communication), and the read rate can be doubled in the case of the C NetCDF4 library, and almost tripled in the case of 1 MiB netCDF4-python reads by utilising more concurrent read tasks.

When using Python, the differences between the 1 GiB and 1 MiB buffer size performance was around 60%. This is due to the reduced performance of larger reads from NetCDF4 files with netCDF4-python. However, the difference when more nodes are utilised is reduced because of the file system becoming the limiting factor for the read rate rather than the read rate on each node.

In Figure 5.5, the different number of concurrent tasks per node indicate how the scaling could work when multiple users are reading from the same shelf in the filesystem. For example, consider users utilising one core per node each across multiple nodes. With four users reading from the same shelf, the read rate for each user would be shown by the four cores per node curve in Figure 5.5.

Variation in read rate could have a large impact in the performance of an application. This is shown by the variation in the results shown in Figure 5.7. Many factors could play into the performance variation, the largest of which is probably down to running on nodes with other users – particularly with NetCDF4-python which is possibly limited by CPU work (as opposed to waiting for I/O). In a workflow where all read tasks must complete before the application moves on, the low rate outliers will have a significant effect on performance of the application. However, if a tool was used where each job can run more than one task in succession (for example the jug Python library, see Appendix B), the low performance outliers would not have as much effect on the overall application performance. Assuming that there

are less workers than tasks to run (each worker runs a task, then moves onto the next task), the variation has less impact on the application performance; if one task is running slowly the other workers can compensate.

5.3 Collective I/O with MPI-IO and NetCDF4

5.3.1 Results

The results for reads using MPI-IO in independent and collective read configuration are shown in Figure 5.8. When using the independent MPI-IO read, the rate increases almost four times when using one task per node on four nodes. However, when using four tasks on a single node the read rate drops to approximately a third of the rate.

For the one task on one node and one task per node on four nodes, the collective reads perform worse than the independent reads. When working on a single node with four tasks, the collective read is faster than the independent read.

The striding reads perform worse than the sequential reads when working with one task on one node (comparing red and blue for the first set of bars), to a similar magnitude as in Chapter 3. Also for the striding reads, when using a single task per node on four nodes the performance is double compared to a single task on a single node – interesting because the sequential reads have around 4 times the performance increase. Comparing one task on one node and four tasks on one node for the striding reads, the independent read rate drops by about 200 MB/s, whereas the collective read rate is more similar in each case.

The only performance increase seen when using collective I/O is when working on a single node with four tasks. For the sequential and striding reads there is about a 60% improvement in the read rate for the sequential read and more for the striding reads compared to independent I/O.

5.3.2 Discussion

The performance of collective reads improved the performance compared to the independent reads when working on one node, but not when working on four nodes. This could be due to overhead with inter-node communication when using multiple nodes, which does not exist when working on a single node.

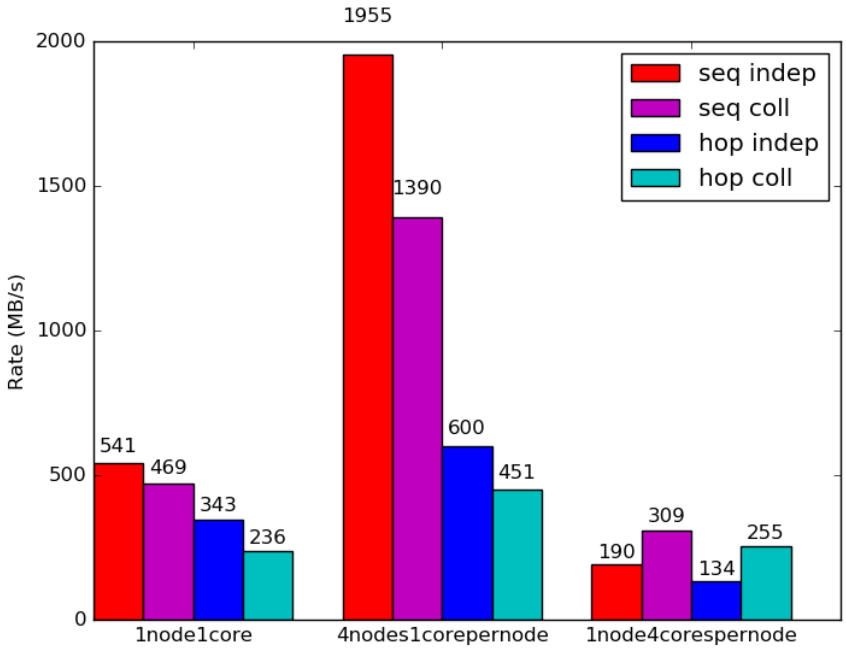


Figure 5.8: Results for the read rate comparing MPI-IO independent and collective reads for different parallel configurations from a single file. The red bars show sequential when using the independent MPI-IO read, the magenta bars show the sequential reads when using collective MPI-IO, the blue bars show the striding read when using independent MPI-IO, and the cyan bars show the striding reads when using collective MPI-IO. The x-axis shows which combination of nodes and tasks were used.

The performance with four tasks on a single node is less than was expected for the independent reads when compared with the reads not using MPI-IO – 540 for one node with one task, dropping to 190 for four tasks on one node. For the single file reads without MPI-IO in Figure 5.4 this drop is not expected. Note that the values are within the variation seen for one task, so the reduced performance may be due to variation. Alternatively, this drop in performance could be due to contention between the tasks on the same node. This could however indicate that there might be some overhead associated with the MPI-IO independent reads when using multiple tasks reading from the same file and working in parallel on a single node.

The benefit for collective reads was only seen when working on a single node (comparing red and purple bars for the third set of bars in Figure 5.8), however, this was a lower rate than when working in serial. When using multiple nodes, there is little reason to use MPI-IO to increase the read rate from a single file. Therefore, to improve the read rate in the most efficient way, multiple nodes should be used without MPI-IO on JASMIN. Even for striding reads where the performance increase was expected, there was no benefit in using collective I/O across multiple nodes. This is likely due to the communication between nodes on JASMIN not being designed primarily for MPI applications as, for example, with Infiniband based platforms (see Appendix A). This comes with the caveats that this is true for these tests on this platform, but may not necessarily transfer to other systems, and does not consider the potential need to write in a workflows, where MPI-IO can be of benefit.

5.4 Conclusions

The aims for this work were:

- Assess the parallel scaling of the JASMIN super-data cluster for application realistic reads.

The number of concurrent tasks and the number of nodes were used in different combinations to assess the scaling. The main conclusions from this were:

- From multiple files, the read rate increases with the number of tasks.
  - For multiple nodes, the read rate increases to a maximum of around 7 GB/s, restricted by the shelf bandwidth. The results agree with extrapolation of results from IOR testing on JASMIN (Lawrence, 2014).

  - On a single node, the bandwidth maximum is reached before the maximum number of nodes was used when considering the read size that gave peak performance with each library. This could reduce the performance of an application if too many tasks per node are used.

- There was no benefit to parallelising the application for reading from a single file on a single node, but there was an increase when using multiple nodes and reading from a single file.

- When reading from multiple nodes, the best scaling for number of concurrent tasks is seen when one task per node is used. The results for scaling when multiple tasks are used per node also mean that if multiple users are running on the same nodes the performance hit would be negligible compared to other factors affecting read performance (e.g. read size, read pattern).

- The reduction in performance when using NetCDF4-python can be easily overcome through the use of multiple tasks on multiple nodes (scaling was similar for this and C).

  - Investigate whether MPI collective reads can improve non-sequential read performance on JASMIN.

MPI-IO was used in independent and collective modes on one and multiple nodes to assess whether there was a benefit to using it for the kinds of workflows simulated here. The conclusion was:

- On JASMIN there is no real reason to use MPI-IO for workflows similar to those that were simulated here. It should be noted, however, that for other workflows it may be a valid solution.

Some advice for reads can be drawn from the results in this chapter. Firstly, when reading from a single file, to see any increase in the read rate, multiple nodes are required. Additionally, only one user using a node compared to multiple users reading from the same shelf does not seem to have a very large impact to the read rate compared to other factors (20% drop from one task per node to four tasks per node at 32 concurrent tasks Figure 5.5(a)), even when reading from the same shelf the greatest impact to the read rate would be about a factor of 2. This

means that exclusively requesting the use of a node would not be worth the extra queue time. Finally, the benefit of faster reads needs to be compared with the amount of potential overhead for communication. For high communication jobs, it may be better to work on a single node, and in this case a collective read could benefit the workflow if the read is striding in nature.

The results from this chapter are summarised in Figure 5.9. The main point from this figure is that the scaling from multiple nodes is similar whether reading from one file or many. If reading from a single node the read rate is obviously limited by the bandwidth, however when reading from multiple files the rate is limited because of the CPU overhead when using netCDF4-python. When reading from a single file, there is no perceivable benefit to executing the read in parallel.

These results are likely to look very different if a burst buffer system was in effect – data layout could be changed when read into the buffer, and the read speed to the buffer would be significantly faster compared to the read rate to the storage.

The next chapter implements the results from this chapter, using the results to design the parallelisation of a particular workflow, and understand the resulting performance.

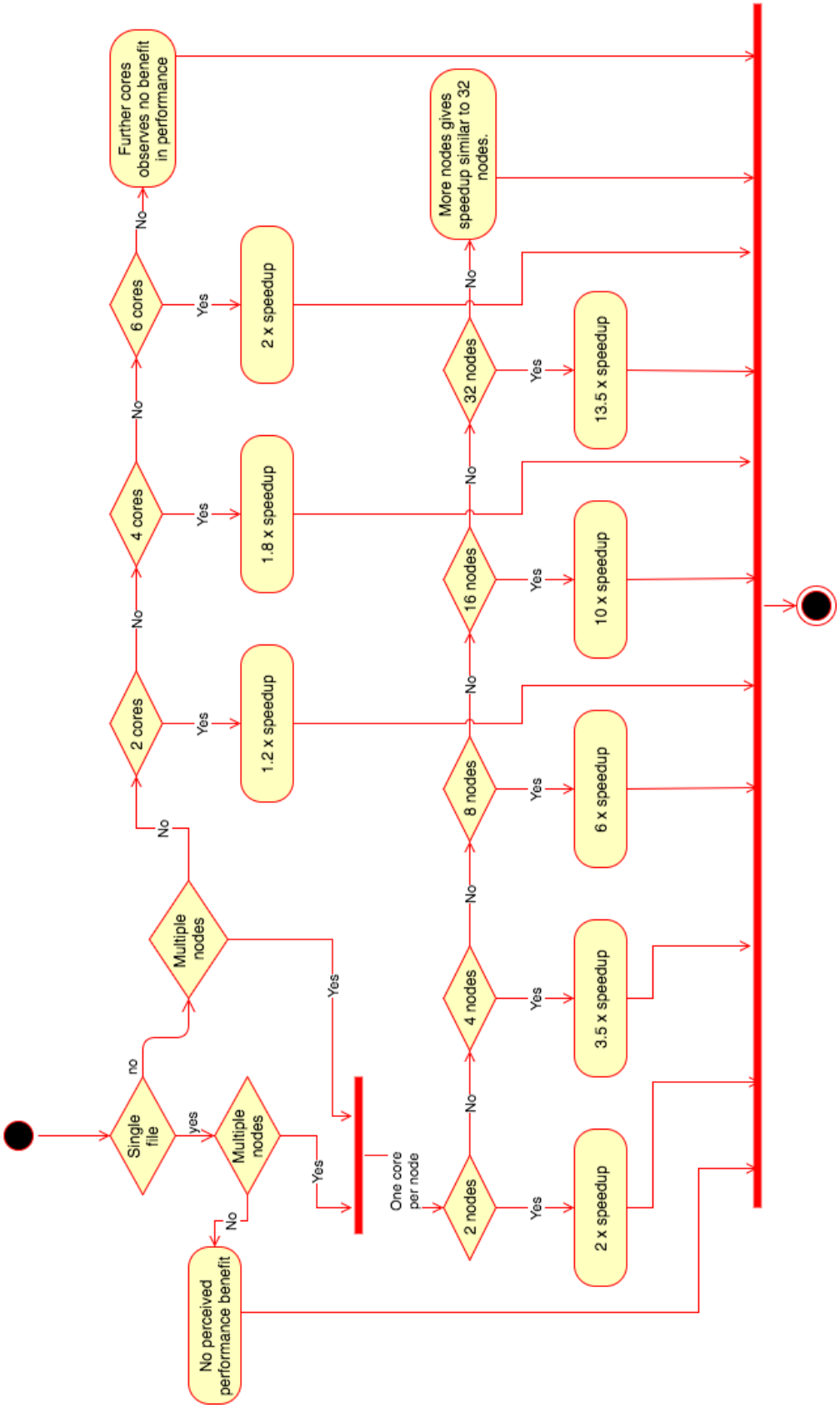


Figure 5.9: Summary of the performance results from the chapter. Speedup values are estimate from the graphs in this chapter.

Chapter 6

Application to Atmospheric Science Workflow - Space-Time Spectral Analysis

The aim of this chapter is to apply the work from the previous chapters into an atmospheric science workflow: testing the case study algorithm with different levels of parallelisation, to confirm that the parallel scaling results from Chapter 5 are relevant when working with a real work flow. The testing presented in this chapter used the netCDF4-python library, so the results of Chapter 3 were used to interpret the read rates in this chapter. The chunking and compression results of Chapter 4 were used to make decisions about which kinds of files to use in the testing of the case study. The computational overhead when using compressed files was also tested in the more realistic circumstances of this chapter.

The aims of this chapter are:

- Assess whether parallelisation can counteract the inefficient layout of data on disc. This will be tested by reading the data from multiple different file layouts.
- Investigate whether running an algorithm in parallel and asynchronously, reduces the overall time for an application to run compare to running it synchronously, when computational work is added.
- Chapter 4 showed that compression adds additional CPU work. Investigate whether this additional CPU overhead significantly affect the completion time for the algorithm.

The work flow used for this work was a space-time spectral analysis (STSA). The next sections: describe the STSA algorithm (Section 6.1), describe the relevance of it in atmospheric science (Section 6.1.1) with an example of its use (Section 6.1.2), and discuss its suitability as a case study for this chapter (Section 6.1.3). The algorithm is investigated in more detail in Section 6.2. The method is outlined in Section 6.3, with results in Section 6.4, discussion in Section 6.5, and conclusions in Section 6.6.

6.1 Space-Time Spectral Analysis

Space-time spectral analysis (STSA) is a method for analysing waves in the atmosphere (Hayashi, 1982). It uses wavenumber-frequency relations to analyse the atmospheric waves and can determine the propagation of the different frequency waves – either eastward, westward, or stationary. The STSA algorithm is outlined in Figure 6.1.

STSA consists of taking a longitude-time slice of the data – normally a longitude circle with a long enough time series to capture the required phenomena. After slicing, the next stage is to compute a fast Fourier transform (FFT) along the longitudinal dimension to get the spatial Fourier coefficients. Then a time analysis technique is computed along the time dimension on each of the Fourier coefficients. The method is determined by the specific analysis technique. Three methods which can be used are a time lag auto correlation method, a discrete Fourier transform method, and the maximum entropy method (Hayashi, 1982). For the tests here the Hayashi method (Hayashi, 1971) is used, consisting of a Fourier transform along the time

dimension. The sine and cosine coefficients from the FFTs are used to compute the stationary, eastward, and westward travelling waves. These two FFTs constitute the most CPU intensive part of the algorithm. These calculations are typically executed for slices for varying latitude and height which are then averaged. The average when the analysis is discretised along the latitude and height contains inter-node communication, which would complicate the analysis, so is not included.

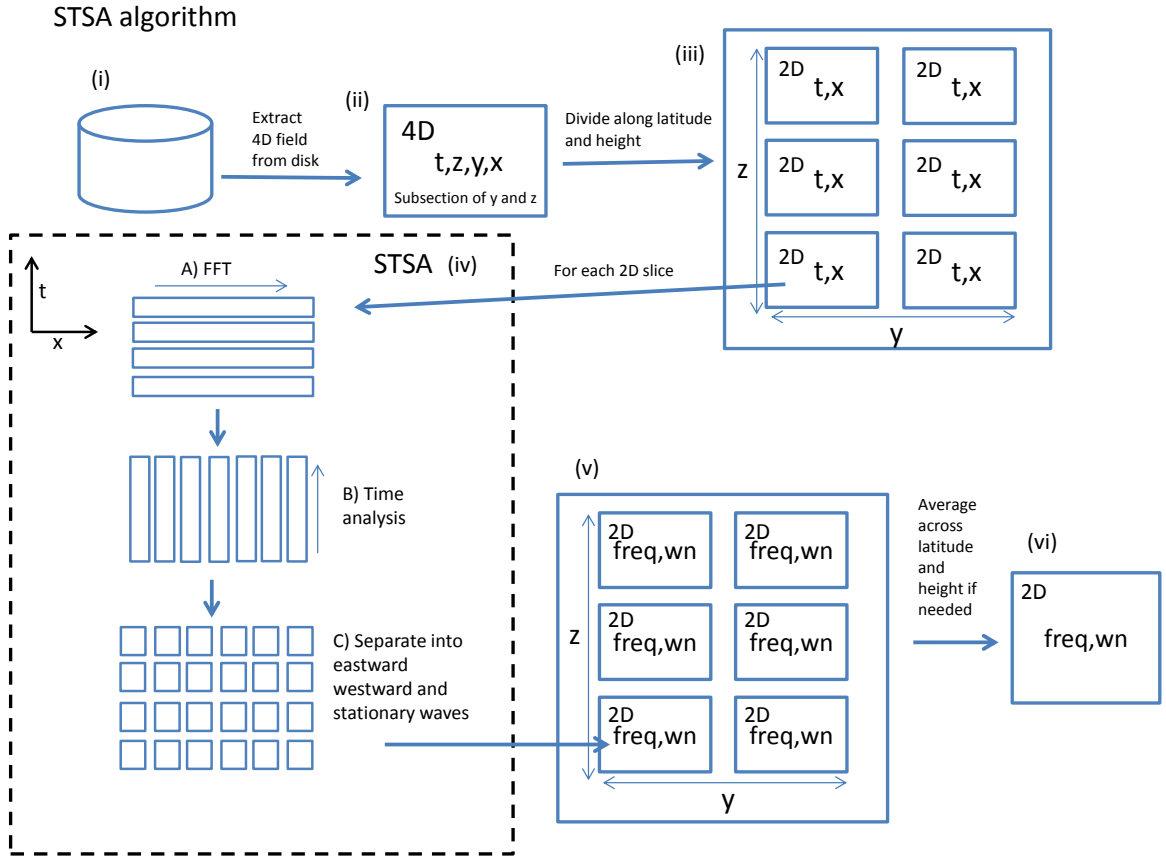


Figure 6.1: Visual description of STSA algorithm.

6.1.1 Relevance of STSA

STSA is a widely used technique because it can be used to analyse waves in the atmosphere. One use of STSA is to identify waves in observations and models. On the former, one example of the use of STSA is to link near-equatorial convection with specific large scale atmospheric waves (Yang et al., 2003), or being used to identify waves from the outgoing longwave radiation (Jiang-Yu and Zai-Zhi, 2008). On the latter, STSA can be used to identify the types of waves which appear in a model (Hayashi and Golder, 1977). STSA can also be used to ex-

amine whether atmospheric phenomena appear in models. For instance, the quasi-biennial oscillation (QBO) is thought to be important in the correct modelling of atmospheric global circulation in models (Lawrence, 2001); STSA can be used to examine the reasons for the QBO being accurately represented (for example Yang et al. 2011).

6.1.2 Example of STSA use

Jiang-Yu and Zai-Zhi (2008) used STSA on out-going longwave radiation (OLR) data from observations to identify waves associated with the tropical intraseasonal oscillation (ISO). ISO has been recognised as a significant factor in local weather and global climate, and can be related to the Indian monsoon. The variability in the synoptic[1] variability in the tropics can be related to large scale convection, for which OLR can be used as a proxy. Jiang-Yu and Zai-Zhi (2008) aimed to show that spectral analysis of OLR can be a useful diagnostic tool for identifying and investigating tropical waves.

Figure 6.2 shows some results from Jiang-Yu and Zai-Zhi (2008). The main results shown in this figure are:

- the large ratio for the Kelvin waves show that they were dominant over the time period investigated,
- and there were significant peaks in the ISO period, indicating that it was predominant for the time period – as observed in synoptic conditions, therefore observable in the OLR observations.

The spectra can then be filtered and the FFTs reversed to isolate waves of interest. The results from Jiang-Yu and Zai-Zhi (2008) show that the OLR observations can be used in conjunction with STSA to identify the predominant waves over a period of time. More generally, that STSA is a useful tool which can be used to better understand phenomena – in this case the observed ISO and convectively coupled tropical waves from the spectral analysis can be used to identify the signature dominant waves for ISO.

6.1.3 Suitability as a case study

STSA is a good case study for a number of reasons:

[1] Synoptic scale being of order 1000km – in the extra-tropics this would relate to extra-tropical cyclones.

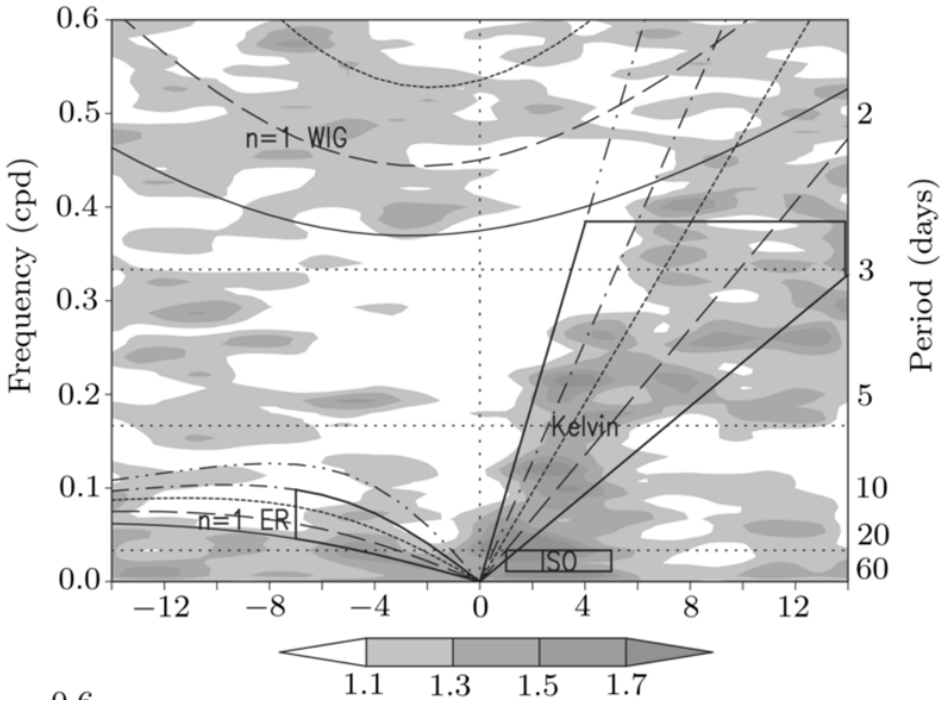


Figure 6.2: Ratio of power spectra in the wavenumber-frequency domain for a set of results from Jiang-Yu and Zai-Zhi (2008) (Figure 2). The darker the shading, the more significant the wave is, and the boxes show waves which were filtered in the results.

- The x-t slice is an inefficient striding read in typical [t,z,y,x] file format (see Chapter 4 for more details).
- FFTs are very strongly communicative so typically do not perform well when distributed, meaning that the best way to perform the FFT is when the array is in a single memory space. This means the easiest way to discretise the algorithm is along the latitude and height dimensions, where only averages are computed.
- For other examples of STSA that do not use an FFT along the time dimension, they will generally use some sort of time lagged auto-correlation which also depends heavily on communication along the time axis, therefore would similarly benefit from being in shared memory.

6.2 Quantitative workflow analysis

The previous section discussed the STSA algorithm in detail. Using the main points in the algorithm, the potential performance can be analysed. The data used in the testing for this investigation was from a N512L180 (see Section 2.4) simulation – the spatial dimensions were

[180,768,1024] ([height, latitude, longitude]).

A longitude-time slice read from standard CF organised files is a very I/O intensive read. For the high resolution N512L180 hourly data, an x-t slice for 400 days is around 80MB, which consists of about 8KB striding reads. From Chapter 3, the estimated read rate for this is around 40 MB/s. With splitting the read between 16 nodes along the latitude direction (assuming a final average along latitude) the read would be around 3.8GB for each tasks with 48 latitude values. At 40MB/s this would take around 1.5 minutes to read - not in itself a problematic time, but long enough to demonstrate speed up. At the peak sequential read from JASMIN (500MB/s) this read would be reduced to 7.5s - showing there could be some significant improvement from reorganising the data.

FFTs scale $n \log n$ (Cooley and Tukey, 1965). Increasing the number of FFTs (not the length of the FFT) will linearly increase the time for the algorithm to complete when the algorithm works in serial. Therefore, to significantly increase the compute time for the algorithm, a long time series is used so the compute time has more chance to be a limiting factor in the algorithm compared to the I/O. Note that when implementing STSA algorithms, the time series can be split into multiple smaller time series which would scale as $m(n \log n)$, where m is the number of FFTs. Therefore, the longer time series will contain a greater number of calculations.

When running the parallelisation asynchronously (e.g. using jug – see Appendix B) there could be some overlap when one node is running the I/O and another is computing the FFTs. This could reduce the overall time for the algorithm to run.

6.3 Method

The testing algorithm was discretised into 16 parts, split along the latitudinal (y) dimension. The Python library `jug` (see Appendix B) was used to enable the parallelisation. Jug works by creating a pool of tasks, from which tasks are removed by workers. The levels of parallelistion used in this testing was:

- a serial test,
- a completely parallel test where all 16 tasks are run simultaneously,
- and an in between test using 4 tasks.

When 16 workers were created, the algorithm ran all 16 tasks simultaneously. When 1 task

was used the algorithm was run in serial. Finally, when 4 workers were used, the first four tasks were run simultaneously, then when a worker finishes a task it started the next task in the queue. This may enable the asynchronous overlap discussed in the previous section.

The testing algorithm had the following steps (see Appendix C for the full code):

- Read x-t slice from file. Which consisted of:
  - The parallel split was be along the y dimension [768], with 16 workers.
  - Each worker looped through the 48 latitude dimensions, and also through the height dimension to execute the read.
  - Each file was 10 days long, so for simplicity in the workflow (only using a single file) some of the height data was used to artificially increase the length of the time dimension of the resulting file which effectively turned a 10 day time series into a 400 day time series, and so increased the amount of CPU work for the FFTs significantly.
- The 'IO' version of the tests stopped there.
- For 'CPU' versions of the test, the STSA algorithm was then computed as described in Section 6.1. `rfft` was used as the FFT algorithm.
- The parallel jobs were on different nodes because the tests in Chapter 5 showed that reading from the same file on the same node had no increase in read rate, whereas reading from multiple nodes did.

For most instances of each test, the test was repeated five times so that the confidence of the mean could be calculated using a 95% confidence interval, along with the mean and standard deviation. In cases where the confidence interval was not calculated, five iterations of the test could not be completed due to the time intensive nature of the tests.

6.4 Results

The results from the case study implementation are summarized in Table 6.1, along with Figures 6.3-6.6.

The total wall times for each different implementation of the case study are shown in Figure 6.3. As expected, the time for each reduced with more concurrent workers (showing the

Table 6.1: Results from applying different techniques to case study. Each average and STD calculated from 16 results. Equivalent read rate would be the read rate if the wall time was just reading the file. All results shown to 2 s.f.

Test type	File type	Concurrent workers	Wall time (s)	Avg wall per task (s) (STD)	Avg cpu per task (s) (STD)	Equivalent read rate (MB/s)
IO	unchunked	1	5600	350 (170)	35 (4.5)	11
IO	unchunked	4	1600	340 (130)	37 (9.0)	39
IO	unchunked	16	380	290 (71)	36 (7.8)	160
CPU	unchunked	1	5900	370 (25)	170 (14)	10
CPU	unchunked	4	1800	400 (76)	160 (34)	33
CPU	unchunked	16	680	390 (46)	160 (41)	89
IO	default	1	11000	680 (82)	660 (81)	5.6
IO	default	4	2700	620 (60)	600 (59)	22
IO	default	16	900	640 (100)	620 (100)	67
CPU	default	1	12000	810 (130)	790 (130)	5.0
CPU	default	4	3300	750 (160)	730 (160)	18
CPU	default	16	1200	860 (210)	830 (200)	48
IO	xt	1	230	14 (4.6)	8.5 (3.5)	270
IO	xt	4	93	21 (9.4)	7.0 (2.6)	650
IO	xt	16	21	21 (6.7)	6.9 (0.80)	2700
CPU	xt	1	1000	63 (23)	56 (22)	59
CPU	xt	4	430	95 (48)	87 (46)	140
CPU	xt	16	210	110 (51)	96 (48)	290

parallelism – 16 workers means totally parallel, 1 means totally serial). For example, the run time for the unchunked I/O only test reduced from about 5500s to 500s, around an 11 times speedup. The default chunking I/O only test also had about 10-11 times speed up. The results for the compressed files showed similar behaviour when using the x-t chunk scheme without compression – the read was fast and the STSA calculations added extra time onto the job – in both cases being much faster per task than any other test combination (Figure 6.4). The default chunking files with compression add a significant amount of time to the read, being about 5-6 times the uncompressed read (Figure 6.4). Note that some of the tests were excluded due to excessive run times.

Figure 6.4 shows the breakdown of average times for each individual task consisting the tests. The results show wall time and CPU time (from Python `clock` function, see Section 3.1.6 for more details on CPU time and `clock`). This work assumed that when the wall time was significantly higher than the CPU time the program was idling waiting for I/O. This meant that the unchunked data reads were limited by the I/O bandwidth, whereas the other reads were limited by time spent processing CPU instructions (CPU time). Unsurprisingly, the fastest reads were from the x-t chunk shaped files (chunking matches read shape).

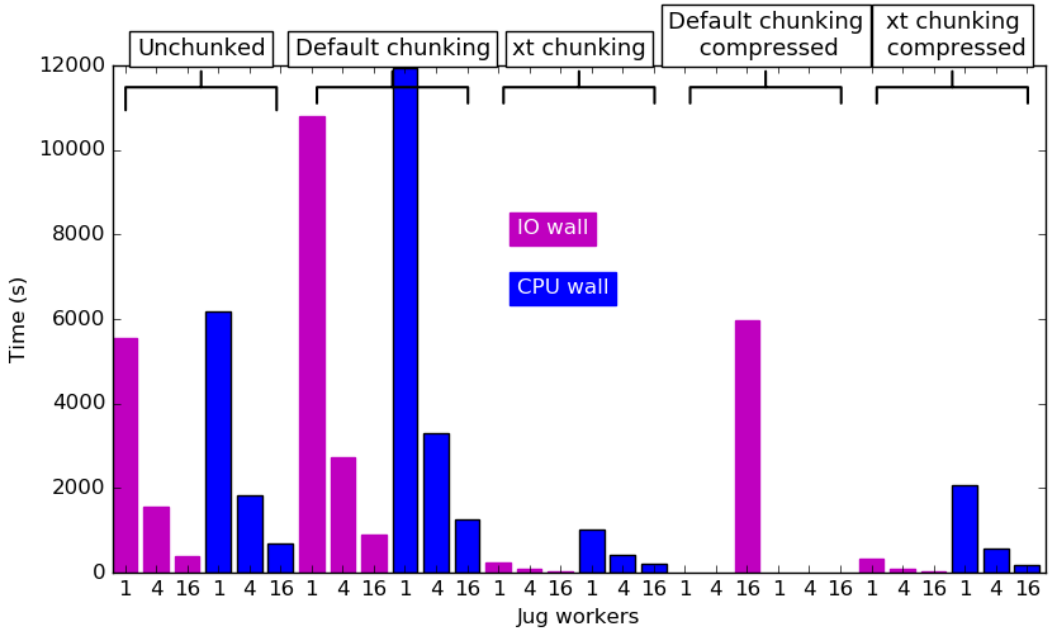


Figure 6.3: Average total wall time for each test. The purple bars show the read only tests, the blue bars show the tests including STSA, the top labels show which file type was being used, and the x-axis shows the number of parallel workers. Note, no results for default chunking compressed files for IO tests at 1, or 4 workers, and for the CPU tests due to excessive run times.

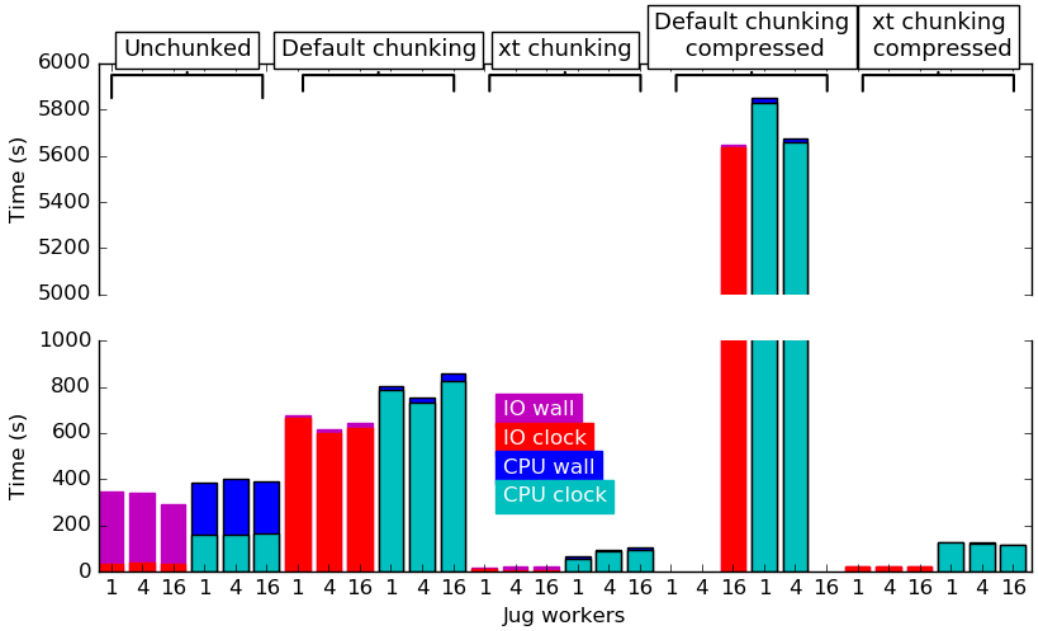


Figure 6.4: Average time per task with CPU and wall time for each test. The purple bars show the wall time for the read only tests, the red bars show the CPU time for the read only tests, the blue bars show the tests including STSA, the cyan bars show the CPU time for the tests including the STSA, the top labels show which file type was being used, and the x-axis shows the number of parallel workers. Note that the bars overlap. Also note that, no results for default chunking compressed files for IO tests at 1, or 4 workers, and for the CPU tests at 16 workers.

The CPU test results consistently added CPU time to the results of around 100-150s per task (Figures 6.3 and 6.4). This added CPU time for the default and xt chunking increased the total time for each task. For the default chunking reads this was because the read was bound by the work on the CPU in the library, and for the xt chunking the wall time increased because the read was so fast and was dominated by the extra CPU work. The wall time for the unchunked files was not significantly increased because the read was bound by the filesystem.

Figure 6.5 shows the same information as Figures 6.3 and 6.4, but contains 95% confidence intervals for the job wall time and task wall time. The main point from these graphs is that the confidence intervals for the task wall time for each test were either overlapping or very close, for example the horizontal lines on the purple squares overlap showing that the mean for the task wall time on the unchunked I/O only test could have been the same for each level of parallelisation.

6.5 Discussion

The approximate 11 times speed up between 1 and 16 workers (Figure 6.3) was what was expected from the speed up results when using NetCDF4 on multiple nodes in Chapter 5. The results in Chapter 5 were based around weak scaling, but the results in this chapter were for strong scaling, so it gives confidence that the results are similar for both. (Strong and weak scaling are explained in Section 2.5.1)

For the tests with STSA included, the results are similar for the unchunked 16 worker test, and the x-t chunked 1 worker test (approximately 700-1000s – Figure 6.3). This leads to two possible workflow strategies. Firstly, that by reorganising the data in the file (chunking in this case but could be dimension reorganisation) similar speedup can be achieved compared to when parallelising the file into 16 tasks, meaning there is more benefit to reorganising the data – parallelising the algorithm is more difficult than changing the data layout. The other strategy is the opposite – by having one copy of the data in unchunked format, a similar level of speed up can be achieved as when reorganising the data, but without having multiple versions of the file. As to which approach is correct depends on the situation – if storage is the limiting factor, then obviously having a single version of the file is much better, as long as parallelising the algorithm is relatively straightforward. Of course, one could argue that by changing the data layout and parallelising the algorithm you can even further decrease the run time. This

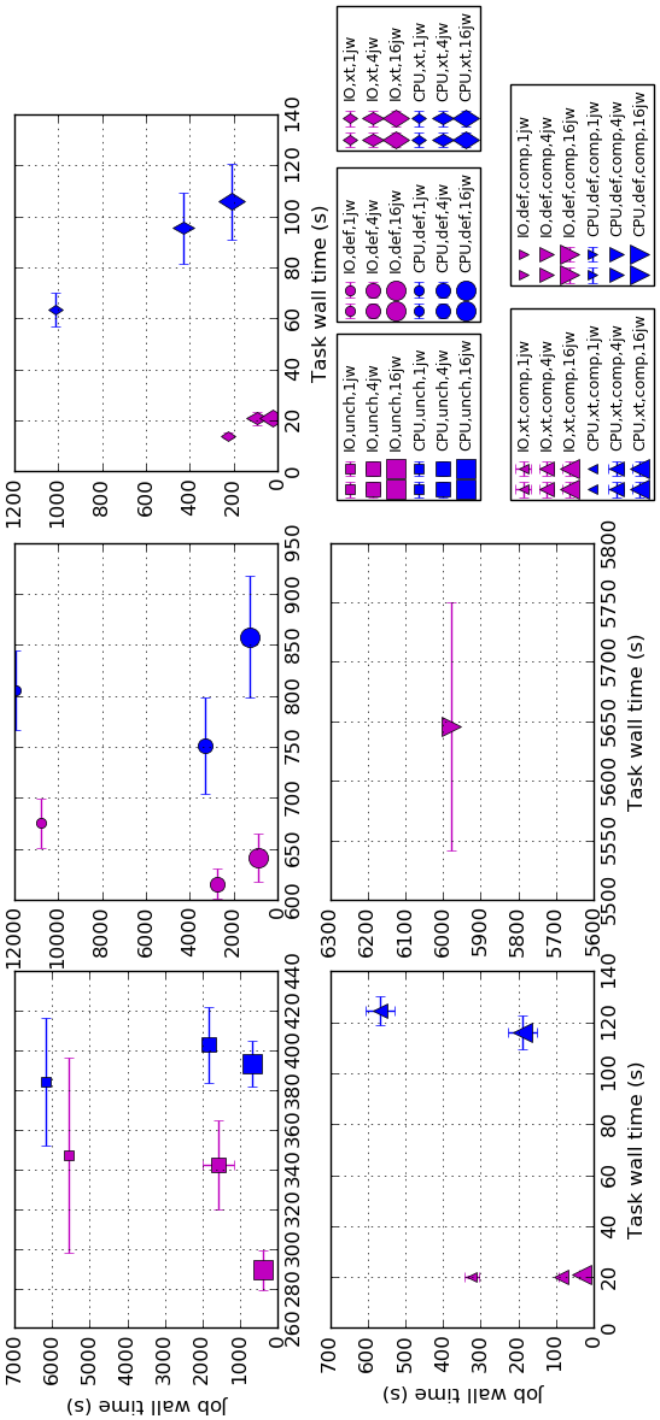


Figure 6.5: Total wall time against task time for each test, with the bars showing the 95% confidence interval. The legend shows which symbol belongs to each test, for example, the smallest purple squares are the read only tests from unchunked files with 1 worker. Each file is shown in its own graph.

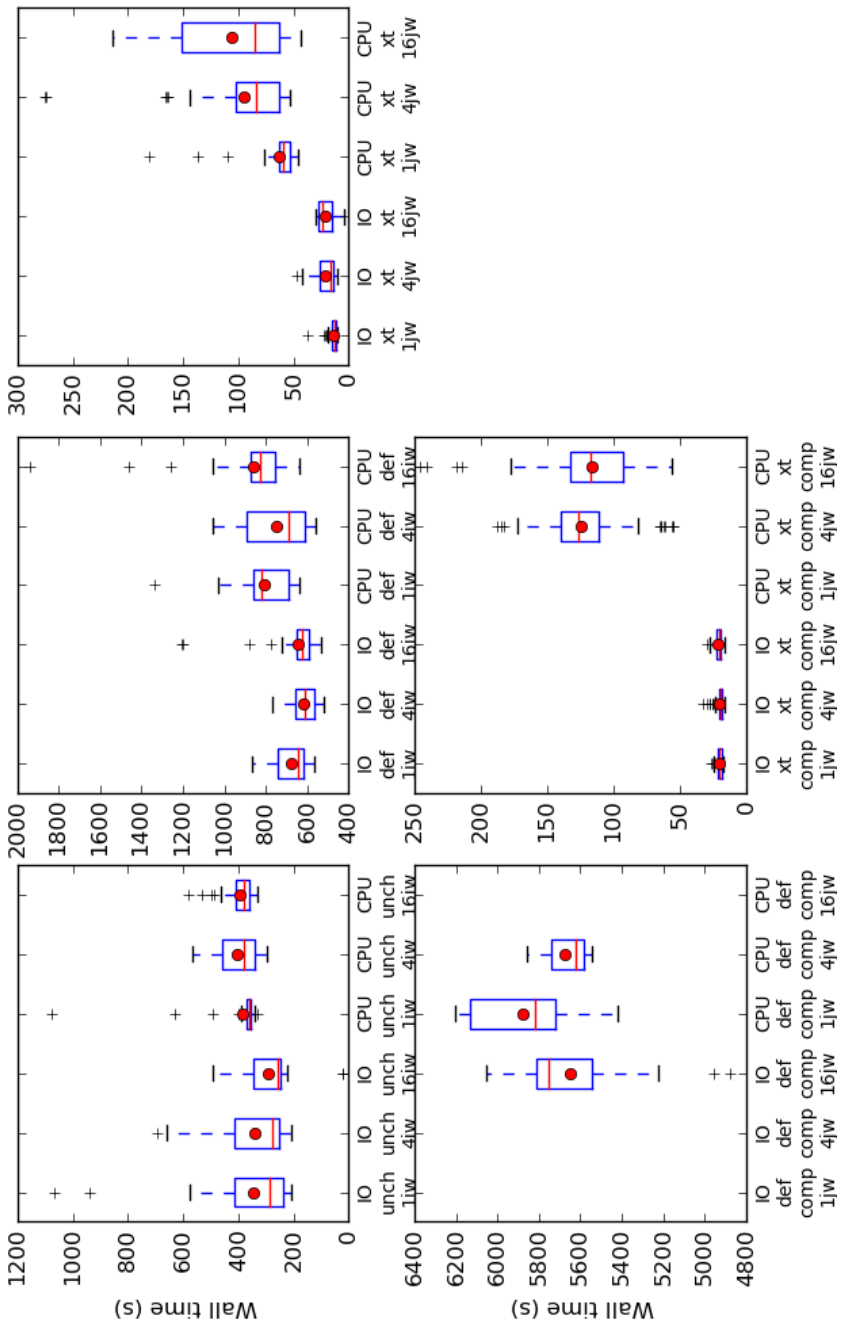


Figure 6.6: The distribution of results for each test. The red dots show the mean, the red line the median, the outside of the blue box shows the upper and lower quartile, the lower whiskers show the lower quartile minus 1.5 times the interquartile range (IQR), the upper whiskers show the upper quartile plus 1.5 time IQR, and the crosses show potential outliers. This is all calculated automatically using matplotlib's `boxplot` function. The labels on the x-axis show what the test composition was for each box and whisker.

final approach is the most beneficial if low run times, or low computing resource costs[2] are the main priority.

The significantly lower run time when using xt chunked files with xt reads compared to the other tests is expected, and is comparable with the results from Chapter 5 (using matching read and chunk shapes significantly benefits the read). Likewise the poor performance with the default chunking is also expected. However, these results do reinforce that the results from Chapter 5 are applicable to the more realistic workflow in this chapter.

The read times from compressed default chunked files were around 5-6 times slower than the uncompressed files (Figure 6.3). This was due to having to read the entire chunk into memory to uncompress it – to uncompress the file the entire chunk is read and uncompressed irrelevant of how much of the data is required, thus drastically increasing the time to retrieve the required data when it is only a small proportion of the chunk. For the xt chunked file this was not a problem because all the data in the chunk was required – there is still an overhead to uncompress the data, but it is not significant compared to the other reads.

The speed up for parallelising the workflow should have been almost linear considering the average task times when accounting for the confidence of the mean (Figures 6.4 and 6.5). However, the average wall times for the whole job scaled sub linearly. This indicates that the variation in the timing for each task had an impact in the total run time for the job.

6.6 Conclusions

The space-time spectral analysis workflow was implemented in a variety of different ways to investigate how it was affected by: chunking, compression, and parallelisation; the interaction between these factors; and the relation to the idealised tests from Chapters 3-5 of the thesis. The main conclusions from this chapter are:

- Parallelisation can compensate for poor chunking decisions, but good chunking decisions can provide the same speed improvement as parallelisation.
- Compression can provide some benefits for reduced space in storage with only a small amount of overhead for uncompression when the data from the entire chunk is required, but if the chunks are inefficient for the read, the performance hit can be huge.

[2] What would also need to be considered here the balance between the cost of storage and the cost of processing time.

Relating the results back to the aims of the chapter which were:

- Assess whether parallelisation can counteract the inefficient layout of data on disc. This will be tested by reading the data from multiple different file layouts.

It was found that parallelisation is very effective in counteracting inefficient data layout.

- Investigate whether running an algorithm in parallel and asynchronously, reduces the overall time for an application to run compare to running it synchronously, when computational work is added.

This is not clear from these results because the extra work introduced from calculating spectra in this work flow may not have introduced enough extra work to tell. Contrarily, for this workflow, there is no benefit from running the parallelisation asynchronously.

- Chapter 4 showed that compression adds additional CPU work – does this additional CPU overhead significantly affect the completion time for the algorithm?

In the case where the required data from a chunk is the entire chunk, CPU time performing compression only adds a small overhead. However, when the data required is only a fraction of the chunk, the overhead can be huge.

The results from this chapter agreed well with results from the previous chapters. Firstly, the speedup with parallel execution of around 11 times with 16 parallel tasks agreed with the scaling curves in Chapter 5. This is encouraging for both sets of results, showing that the idealised tests in Chapter 5 showed a good approximation of the speedup expected in a realistic workflow.

The default chunking performed worse than the unchunked for this read pattern, which was unsurprising considering the mismatch between the read shape and the chunk shape. The difference was also consistent with the results in Chapter 4, but it was important to repeat to show that even in the more realistic case in this chapter that the default chunk specification provides poor read rates. The results in this chapter also showed the important huge increase in time when using compressed default chunking. The increase in read time when using xt shaped chunks was more consistent with the results in Chapter 4. This showed the importance of having knowledge about how the workflow relates to the organisation of the data.

Chapter 7

Conclusions and Further Work

Data volume is increasing in atmospheric science and new techniques are required to analyse it. There are some new libraries, middleware, and hardware solutions to handling big data in scientific domains. However, for atmospheric scientists developing or upgrading analysis scripts for a specific task, these solutions may not be viable. Therefore, knowing the quantitative effects of different layers in the software hardware stack on performance, is paramount for developing efficient applications. Throughout this thesis, different factors affecting the read rate for typical data analysis in atmospheric science were analysed. This chapter will summarise the results and draw overall conclusions of the work. The second part of this chapter discusses the work in the context of other projects, and the final part discusses further work.

7.1 Summary and conclusions

In Chapter 3, the effect of language, file type, read patterns, and read sizes, were investigated. The aims for Chapter 3 were:

- Evaluate the performance effect of using NetCDF4-python, and assess the reasons for any reduction in performance.
- Evaluate the effect on the read rate for different read patterns and read sizes.

The main conclusions from this chapter were, firstly, that the reduction in the read rate when using the netCDF4-python library is caused by the library (at least for that version of the library, see Appendix B for version details), therefore there was no reduction in performance compared to the C library performance due solely to Python or by using the NetCDF4 file type.

Secondly, non-sequential reads cause a significant reduction in read performance, probably due to the inability of the filesystem to correctly read ahead in the file. The reduction in read rate due to striding reads is greater than the difference between the netCDF4-python and the C NetCDF4 library. Therefore, despite the performance issues with netCDF4-python it is not the most important factor, meaning the use of netCDF4-python may not be as detrimental as it first seems. The results from the effect on the read rate of chunking and compression (Chapter 4), reinforce this.

In Chapter 4, the effect of NetCDF4 chunking and compression on the read rate was investigated. The aim from this chapter was:

- Quantitatively evaluate the effect that chunking and compression have on the read rate, and therefore the overall workflow.

A number of conclusions were drawn about how the interaction between chunking and compression can have a serious detrimental impact in the read rate from NetCDF4 files, if bad decisions are made about both. Firstly, there is no acceptable compromise whereby chunking can provide good read rates for multiple read patterns for multidimensional atmospheric data, at least with the current technology being used on JASMIN (and probably similar on other HPC platforms). The reduction in read rate for the compromised read is too great (50 to 100 times worse), whilst only providing a minor improvement for the targeted read patterns. Secondly, compression with the deflate algorithm can reduce the size of NetCDF4 files by approximately four times. Compression does, however, add an overhead to the read: for chunk shapes which are read in their entirety, the overhead is small, but when only a section of the chunk is required, the overhead is very large due to having to read and uncompress the entire chunk.

Chapter 5 investigated the parallel scaling of the JASMIN super-data cluster in terms of a typical user's analysis script from atmospheric science. The aims were:

- Assess the parallel scaling of the JASMIN super-data cluster for application realistic reads.
- Investigate whether MPI collective reads can improve non-sequential read performance on JASMIN.

The first major conclusion from this chapter was that it is possible to estimate the potential

achievable bandwidth for a single user, and multiple users, working off the same shelf in the file system, for multiple and single node parallelism. Secondly, if working from a single file on a single node, there is no performance increase for running multiple concurrent tasks on JASMIN. In addition, there is no benefit to using collective reads on the JASMIN platform. More testing would be required on other platforms to generally assess the benefit of MPI-IO collective reads for data analysis in atmospheric science.

The final chapter, Chapter 6, applies the investigation from Chapters 3, 4, and 5, into a realistic atmospheric science workflow.The aims were:

- Assess whether parallelisation can counteract the inefficient layout of data on disk. Tested by reading the data from multiple different file layouts.
- Investigate whether running an algorithm in parallel and asynchronously, reduces the overall time for an application to run compare to running it synchronously, when computational work is added.
- Chapter 4 showed that compression adds additional CPU work. Investigate whether this additional CPU overhead significantly affects the completion time for the algorithm.

The application of different chunk shapes, compression, and different levels of concurrency, show that parallelism can compensate for poor chunking decisions, and good chunking decisions can provide similar speedup to running multiple concurrent tasks. However, poor chunking decisions and compression can be disastrous for performance. The STSA workflow in this chapter did not contain as high a proportion of compute to I/O as originally thought, so it was not possible to make conclusions as to whether asynchronous reads would benefit atmospheric science analysis. Another interpretation of this statement is that the I/O is still the major bottleneck for atmospheric science workflows, agreeing with Balaji (2015). Another important implication of this chapter is that the flow charts created from the other chapters can be used to estimate the read rate for realistic workflows.

Figure 7.1 shows the aggregation of the flowcharts from Chapter 4, and Chapter 5. This flow chart can be used to estimate the read rate for an application on JASMIN, when given knowledge about the read, such as: read pattern, read size, and data layout, along with estimated speedup through parallelisation. Similar flowcharts could be made for different plat-

forms using a similar methodology. However, more investigation is needed to quantify the effect of the factors in the red boxes. As an example of the use of this flow chart, consider the STSA algorithm with data stored in an unchunked file. Following the flow chart in Figure 7.1, the data is unchunked with a non-sequential read pattern. Next, the read size is calculated from the sequential part of the slice, in this case, 1024 8 byte floats, meaning an 8 KB read, giving a read rate less than 50 MB/s. The total size of the data involved in the analysis is approximately 60 GB. Therefore, this read will take at least 1200 s – in reality, the x-t read is more like 10 MB/s giving a read time more like 6000 s, which is much closer to the observed time. This approximate figure is useful to compare the read rate which could be achieved by parallelising the workflow, or chunking the file into x-t shaped chunks. The conversion to x-t shaped chunks would improve the runtime from around 100 mins, to approximately 3 minutes, for which the benefit would increase the more times the application is run. This trade-off obviously needs to be carefully considered, but from a productivity point of view, and with some foresight, the 33 times speedup could be advantageous. This is also without considering further speedup through implementing the analysis in parallel – also, the file conversion could very easily be done in parallel reducing the time to convert between chunk shapes.

As a second example workflow accumulating the results from the NetCDF4 and chunking investigations to determine whether reformatting data is the correct solution for a workflow, consider the following:

- An STSA workflow working with one year of N512L180 data. The array order [t,z,y,x] gives fast (500MB/s) x-y reads and poor (8 MB/s) x-t reads (Figure 4.3).
- This resolution of [8640, 180, 768, 1024] gives a 9.8TB dataset.
- To read the entire dataset at around 500 MB/s (Figure 3.7) it would take around 5 hours.
- To give fast x-t reads the data is required to be in effectively [z,y,t,x] order. This can either be done via chunking, or rearranging the dimension order in the file.
- Assuming that either can be done in memory for free and saved at the same rate that it was read (not unreasonable from the results in Figure 2.1 showing the read and write rates scaled similarly on JASMIN), it would take about 10 hours to convert the data. (Comparing this to the results from Table 4.4, 10 hours gives an effective rate of 250 MB/s, and converting from unchunked files to x-t chunked files was at 25 MB/s. This

difference is explained by nccopy not performing the conversion in sequential reads and writes)

- Relating this time to the read time for a single x-t slice at 8 MB/s, gives 4.5s.
- However, the workflow for STSA is to average over y and z. So at 8 MB/s reading x-t slices from the [t,z,y,x] dataset would take 320 hours (13.3 days).
- The results from Chapter 4 showed that an x-y-t slice was much faster to read at 145 MB/s. Which means reading the 9.2 TB dataset in 18.8 hours.
- Now, returning to the [z,y,t,x] file, the x-t read could be executed at around 400 MB/s meaning the read would take approximately 6 hours.

Adding the conversion time, 10 hours, to the new read time, 6 hours, gives a total of 2 hours less than reading the original file with x-y-t slices, and 20 times less than reading with x-t slices. This estimate however assumes serial reads. Now considering the results from the parallel scaling investigation:

- The bandwidth from a single shelf on JASMIN was approximately 7 GB/s (Figure 5.5).
- This is 14 times faster than the serial rate, giving a total time for the combined conversion and subsequent read of approximately 70 minutes, at around 80 concurrent tasks (Figure 5.5).
- For the x-y-t reads from the original file, at linear scaling it would take 48 nodes to attain 7 GB/s – but Figure 5.5 shows that linear scaling is not achieved, so it would take many more nodes than the above case with conversion. However, the runtime could potentially be reduced to approximately 20 minutes.
- For the original x-t read, linear scaling implies that at least 300 tasks would be required to attain 7 GB/s.

This discussion shows that, despite achieving a similar serial time for the rearranged file and the x-y-t read pattern from the original file, the rearranged file is parallelised over fewer tasks to a reasonable run time (around an hour). The preferred approach depends on the level of parallelisation available. Parallelising the x-t read to an acceptable run time would not be practical however. A caveat of this thought experiment is that it is assumed that the conversion

of data order in the file, either through chunking or otherwise, is free – the conversion results from Chapter 4 show that when using the nccopy utility, there is, in practice, a significantly higher overhead.

The major contributions of this thesis which have not previously been quantified are as follows:

- The methodology in this thesis has provided a framework to develop an estimation of read rates for typical workflows in atmospheric science. In particular, this thesis has provided estimate effects on the read rate for:
  - non-sequential read patterns,
  - the netCDF4-python library,
  - NetCDF4 chunking and compression, and the combination of the two,
  - and the scaling of different parallel configurations on JASMIN.
- Demonstration that bad combinations of compression and chunking can severely detriment the read rate from NetCDF4 files.
- Demonstration that NetCDF4 chunking cannot provide an acceptable compromise for multiple read patterns from the same file.
- Demonstration that, on JASMIN, using multiple parallel tasks on a single node does not improve the read rate.
- Demonstration that parallelisation can compensate for poor read patterns, but good read patterns can provide the same speedup as parallelisation.

The limitations for this work are as follows:

- Only a limited number of repeats were run for all tests (between three and five generally). More results would reinforce the conclusions and gain a better idea of the mean and variability. Due to the nature of this kind of testing being very time intensive, and in most cases not being able to run multiples of the tests due to potential conflict for resources, it was only possible to run a limited number of tests. However, useful conclusions can still be drawn on the results from this thesis.

- While interesting results were found from the STSA application, the results from Chapters 3 through 5 were only applied to that workflow. Applying the same approach to other workflows would further test the usefulness of the methodology.
- No filesystem parameters were varied throughout any of the testing, since users generally may not, or are not able to, change these settings (as described in Chapter 2). However, the results could potentially have a large impact on the read performance for applications.

7.2 Related work

In Chapter 3, the importance of sequential read patterns was demonstrated. There are at least three techniques which can be used to improve the poor performance of non-sequential reads beyond the investigation in Chapter 4. They consist of:

- read prediction or prefetching (within the filesystem),
- preloading data (within layers above the file system, e.g. middleware or extra hardware),
- and data layout.

The lower IOPS with random or striding reads could be mitigated by accurate read prediction or prefetching in the filesystem. In a simple sense, this would entail using an algorithm to predict future reads from an application. He et al. (2013) demonstrate that using a pattern prediction algorithm (using a virtual filesystem) can reduce I/O latency by up to three times, and was much more effective than Linux readahead. With better prediction of reads, the difference between the sequential reads and striding reads in Chapter 3 would reduce. Jiang et al. (2013) also used a prefetching method and showed some speedup, and Byna et al. (2008) showed improved read rate when using a layer they developed for the MPI-IO library. All these cases show promise for implementing some form of prefetching on the HPC systems used in atmospheric science (e.g. JASMIN). Some of the levels of speedup demonstrated in these studies were not enough for smaller striding reads to match the performance of the sequential reads, needing 10 to 100 times speedup for the striding reads. He et al. (2013) demonstrated 3 times speedup, Jiang et al. (2013) showed 1.2-1.6 times speedup, and Byna et al. (2008) showed a maximum of 1.25 times speedup.

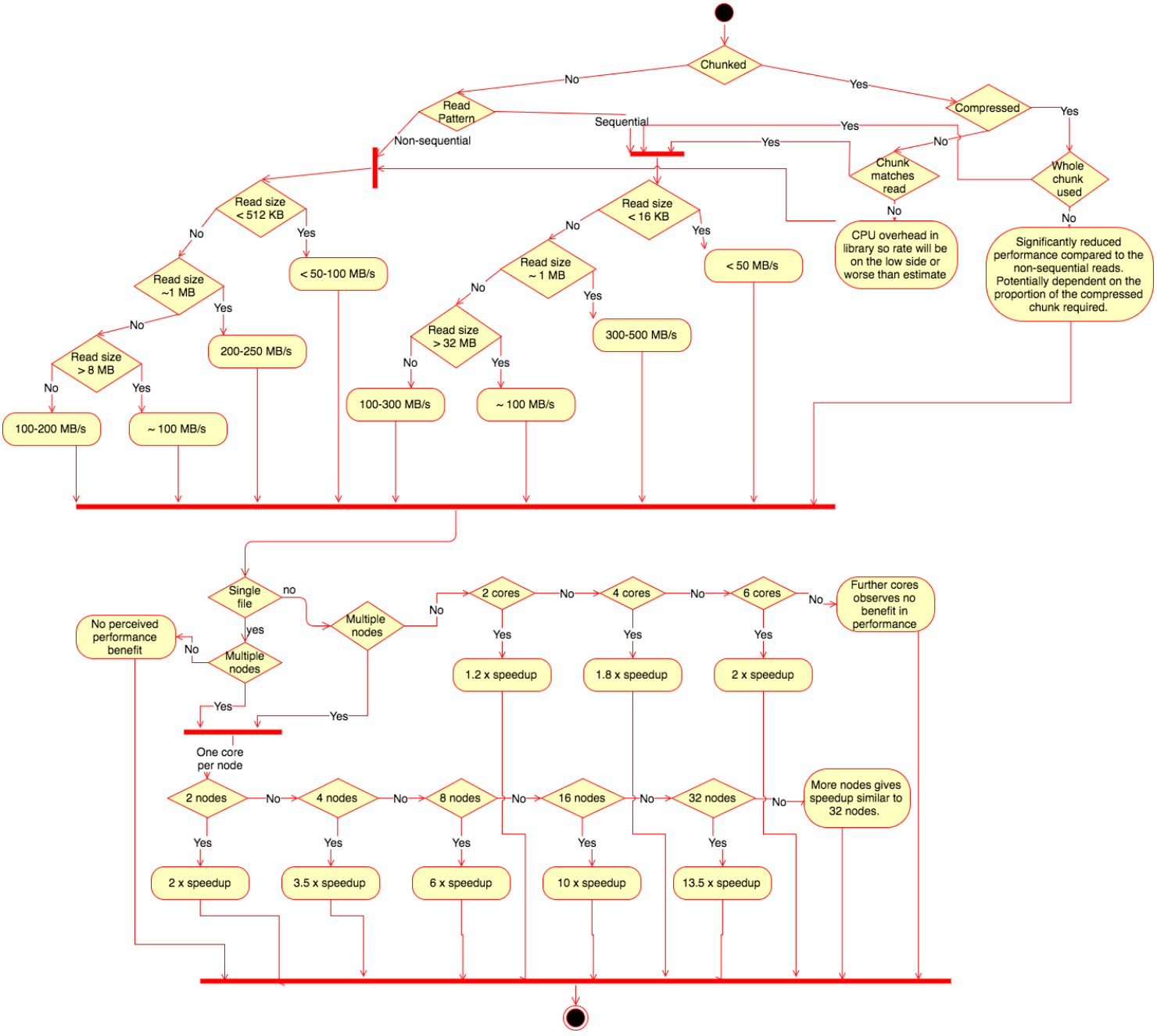


Figure 7.1: Cumulation of the results from all chapters in the thesis, into a flowchart model.

Similar in some ways to prefetching, preloading data either predicts what data is required and reads it ahead of time, or reads data using an intermediate stage. A burst buffer is generally located between the parallel filesystem and the HPC compute cluster. This intermediate storage can provide faster access and read rates than the main parallel filesystem. Another benefit of a burst buffer system could be that the data once read onto the burst buffer nodes, could be rearranged or reformatted to enable fast reads from the main filesystem, and faster transfers to the processing nodes. This would obviously be beneficial looking at the results from this thesis. An example of this method is BurstMem (Wang et al., 2015), which is an example of a burst buffer system. BurstMem showed significantly improved performance over a non-burst buffer system (over twice the bandwidth in some cases). This improved performance, along with the potential to reorganise data in the burst buffer, could significantly reduce the runtime for the STSA application in Chapter 6. This would give similar performance to, if not better than, the x-t chunked file (fastest read) while only storing one version of the data, thus keeping storage costs low. However, BurstMem does not currently support NetCDF and HDF5 files, so has limited usefulness in Atmopsheric science.

Another way of mitigating poor performance due to inefficient read patterns is to change the layout of the data on disk. Chapter 4 showed that simply using HDF5 chunking gives beneficial results for a single read pattern, but not multiple read patterns. Some studies which aim to solve this problem include:

- using the SDS framework (Dong et al., 2013),
- using PLFS to interface between the physical filesystem and a logical file view from an application,
- using the ADIOS BP file format,
- and using the Ceph filesystem to improve random access speed.

The SDS framework improves the performance of multiple read patterns by automatically reformatting data by analysing read patterns of applications, and automatically selects the best version of the dataset for a given application using metadata. This shows clear benefits with regard to reading of the reorganised data being 50 times faster. This is a similar speedup to the chunked files results from Chapter 6 (around 30 times faster comparing unchunked and x-t chunked files), with similar overhead (each requires multiple files). However, the SDS framework does this automatically and transparently to the user.

PLFS is a virtual filesystem which allows a mapping between the physical data and a virtual file which can have a different data order, giving a similar effect to HDF5 chunking. He et al. (2013) showed significant improvement when using PLFS to provide this mapping. However, it does not seem to provide any benefits that chunking does not.

The ADIOS BP format (see Section 2.8), organises data on disk to provide fast access for multiple read patterns (Lofstead et al., 2011). This gives similar benefits to having multiple chunked files, without the storage overhead. Applying this file type and data distribution to the STSA, the BP file format could provide around 20 times speedup, which is very similar to the speedup shown when comparing the unchunked files and x-t chunked files in Table 6.1. However, it is a specific format which would lose the benefits of NetCDF, such as its well established metadata conventions and tools.

Ceph is an object store based file system like Panasas and Lustre (see Appendix A). Unlike those two however, it removes the file allocation tables (metadata describing where files are stored) in favour of using an algorithm to locate data, significantly reducing metadata access time. The data across the object stores is also randomly distributed (Weil et al., 2006) as opposed to using a round-robin distribution which favours reading chunks in storage order. These two factors could significantly improve the performance of non-sequential read patterns. The distribution of data involves striping (see Section 2.7.2), which Chapter 4 showed can interact poorly with chunking. One way to avoid this would be to not chunk the NetCDF4 data, however this would eliminate the option of the inbuilt compression in the NetCDF4 library. Despite this drawback, using Ceph could be very beneficial to atmospheric science workflows, giving a similar level of speedup to ADIOS BP files, while still using NetCDF4.

The compression results from Chapter 4 give similar compression ratios to Kunkel (2017) which showed 40-70% compression ratios for mixed scientific data (including NetCDF4 files) with zlib compression. Kunkel (2017) showed that the decompression speed for these files was around 20 times slower than the observed bandwidth. The results from Chapter 4 show that the overhead for compressed NetCDF4 files, where the read matches the chunk shape, was around two thirds the uncompressed read rate. However, when the reads consist of multiple whole chunks, the overhead was negligible. This shows the importance of making good chunking decisions when compressing data. The reads from Kunkel (2017) likely included reads where the read shape and chunk shape were mismatched, increasing the mean uncompression time.

There are three European projects which aim to improve exascale data analysis: ESiWACE (Centre of Excellence in Simulation of Weather and Climate in Europe)[1], SAGE[2], and NEXTGenIO [3]. Part of the ESiWACE project involves developing software to handle analysis of large data sets. One interesting aspect of this work is on the fly rearrangement of data to improve subsequent access speeds to data sets, i.e. after the data has been accessed the first time, the access pattern is analysed, and the data reorganised to improve the read performance for subsequent reads. SAGE aims to utilise a storage hierarchy with object stores to improve I/O performance, along with supporting software. NEXTGenIO aims to bridge the gap between storage and compute using non-volatile memory (NVRAM), by designing appropriate hardware and software to take advantage of NVRAM.

Most of the studies discussed in this section, make the assumption that large data analysis is a problem. However, they do not seek to quantify the problem, and instead focus on providing solutions. While this is a valid approach to the problem, it is targeted to those who have already encountered problems associated with big data analysis. Quantifying the factors which affect the performance of data analysis makes scientists aware of which of these factors could affect their application, and avoidance of these problems can save time during development. The results in this thesis have quantified many of these factors.

7.3 Further work

There are five avenues of further work which would expand upon the work in this thesis.

Firstly, implementing more case studies would affirm the performance results in Figure 7.1. Three further case studies were identified which could provide useful results. The case studies were: k-means clustering, cyclone tracking, and a global energy or moisture budget. Clustering is a method of grouping data into different categories, which requires an iterative algorithm, differentiating it from the STSA case study significantly. Cyclone tracking is a method of finding the trajectories of a cyclone through data from simulations or satellite data, by searching through fields for specific circumstances, and connecting them between timesteps to form a trajectory. Global budget studies involve evaluating many different fields, and processing a large amount of data, including finite difference calculations; creating strong dimensional

[1] https://www.esiwace.eu/
[2] http://www.sagestorage.eu/
[3] http://www.nextgenio.eu/

dependencies in the data fields.

Secondly, investigating the reasons involved with the reduced performance due to partial reads of chunks in NetCDF4 files (when compressed and uncompressed) would be useful. Also, it would be useful to quantify in more detail the time involved with reformatting files for different read patterns. This would give a more complete picture of the effect on the workflow.

Thirdly, the creation of similar flowcharts for other HPC platforms with different filesystems (ARCHER, and the RDF) would be useful. Firstly, this would confirm that the method in this thesis is more widely applicable. Secondly, having the performance modelled for different platforms would be useful for users working on those platforms.

Adaptively changing the layout of data in the file to provide more efficient access to a specific application would drastically improve the read performance. This could either be done after the first run of an application (i.e. once the algorithm has learnt how to format the data for the application), while the user is in a batch queuing system waiting to run (the user would provide prompts to the system to tell it how to format the data), or this could be done when moving data into a burst buffer (discussed below). All of these approaches would improve the read rate without having to permanently store multiple version of the file, also with little overhead for the user.

Finally, there are other types of architecture which could improve the I/O performance compared to a HPC parallel file system, such as using burst buffers, using a Hadoop-type cluster (see Section 2.7.2), or using an alternative filesystem such as Ceph. A burst buffer enables the data to be staged onto high speed storage, typically solid state, reducing the overall read rate, especially when accessing the data multiple times. For certain workflows Hadoop and MapReduce could be the right solution for a workflow depending on the data distribution and layout. Ceph reduces the amount of metadata required for a filesystem by eliminating the file allocation table, and randomly distributes the striped data (as opposed to Panasas which uses a round-robin type distribution). Both of these attributes could improve the performance of reads, particularly non-sequential reads with random data distribution (it could provide similar speedup to the ADIOS BP file format). Using the methodology in this thesis could examine whether these architectures would be broadly beneficial for workflows in atmospheric science.

BIBLIOGRAPHY

Amdahl, G. M., 1967: Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the April 18-20, 1967, spring joint computer conference*, ACM, 483–485.

Asanovic, K., et al., 2006: The landscape of parallel computing research: A view from berkeley. Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`.

Baker, A. H., et al., 2014: A Methodology for Evaluating the Impact of Data Compression on Climate Simulation Data. *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, 203–214, doi:10.1145/2600212.2600217, URL `http://doi.acm.org.ezproxy.lib.utexas.edu/10.1145/2600212.2600217`.

Balaji, V., 2015: Climate Computing: The State of Play. *Computing in Science & Engineering*, **17 (6)**, 9–13, doi:10.1109/MCSE.2015.109, URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7274265`.

Balaji, V., et al., 2017: CPMIP: measurements of real computational performance of Earth system models in CMIP6. *Geoscientific Model Development*, **10 (1)**, 19.

Barton, E. and A. Dilger, 2015: *High Performance Parallel I/O*, chap. 8, 91–106. CRC Press.

Bartz, C., K. Chasapis, M. Kuhn, P. Nerge, and T. Ludwig, 2015: A Best Practice Analysis of HDF 5 and NetCDF- 4 Using Lustre. *High Performance Computing*, Springer International Publishing, 274–281.

Blank, T. and J. R. Nickolls, 1992: A grimm collection of mimd fairy tales. *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*, IEEE, 448–457.

Blower, J., A. Gemmell, G. Griffiths, K. Haines, A. Santokhee, and X. Yang, 2013: A Web Map Service implementation for the visualization of multidimensional gridded environmental data. Elsevier, URL http://centaur.reading.ac.uk/31396/12/ncWMS_paper_EMS_2013.pdf, doi: 10.1016/j.envsoft.2013.04.002 ¡http://dx.doi.org/10.1016/j.envsoft.2013.04.002¿.

Borrill, J., L. Oliker, J. Shalf, and H. Shan, 2007: Investigation of leading HPC I/O performance using a scientific-application derived benchmark. *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC '07)*, doi:10.1145/1362622.1362636.

Borrill, J., L. Oliker, J. Shalf, H. Shan, and A. Uselton, 2009: HPC global file system performance analysis using a scientific-application derived benchmark. *Parallel Computing*, **35 (6)**, 358–373, doi:10.1016/j.parco.2009.02.002, URL http://www.sciencedirect.com/science/article/pii/S0167819109000271.

Buck, J. B., N. Watkins, J. Lefevre, C. Maltzahn, and S. Brandt, 2011: SciHadoop : Array-based Query Processing in Hadoop Categories and Subject Descriptors. *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 66.

Byna, S., Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp, 2008: Parallel i/o prefetching using mpi file caching and i/o signatures. *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, IEEE Press, 44.

Caron, J., 2014: Compression by Bit-Shaving. URL http://unidata.ucar.edu/blogs/developer/entry/compression_by_bit_shaving.

Chasapis, K., M. F. Dolz, M. Kuhn, and T. Ludwig, 2014: Evaluating Power-Performance Benefits of Data Compression in HPC Storage Servers. *International Conference on Smart Grids, Green Communications and IO Energy-aware Technologies*.

Childs, H., E. Brugger, K. Bonnell, J. Meredith, M. Miller, B. Whitlock, and N. Max, 2005: A contract based system for large data visualization. *Visualization, 2005. VIS 05. IEEE*, IEEE, 191–198.

Cinquini, L., D. Crichton, A. Braverman, L. Kyo, T. Fuchs, and M. Turmon, 2014: Dawn: A simulation model for evaluating costs and tradeoffs of big data science architectures. *AGU Fall Meeting Abstracts*, Vol. 1, 03.

Cooley, J. W. and J. W. Tukey, 1965: An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, **19 (90)**, 297–301.

Crichton, D. J., et al., 2012: Sharing Satellite Observations with the Climate-Modeling Community: Software and Architecture. *IEEE Software*, **29 (5)**, 73–81, doi:10.1109/MS.2012.21, URL http://ieeexplore.ieee.org/document/6133265/.

Dean, J. and S. Ghemawat, 2008: MapReduce: simplified data processing on large clusters. *Communications of the ACM*, **51 (1)**, 107–113, URL http://dl.acm.org/citation.cfm?id=1327492.

del Rosario, J. M., R. Bordawekar, and A. Choudhary, 1993: Improved parallel I/O via a two-phase run-time access strategy. *ACM SIGARCH Computer Architecture News*, **21 (5)**, 31–38, doi:10.1145/165660.165667.

Dong, B., S. Byna, and K. Wu, 2013: Expediting scientific data analysis with reorganization of data. *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, 1–8, doi:10.1109/CLUSTER.2013.6702675, URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6702675.

Flynn, M. J., 1972: Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, **C-21 (9)**, 948–960, doi:10.1109/TC.1972.5009071, URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5009071.

Fox, G. C., S. Jha, J. Qiu, and A. Luckow, 2014: Towards an understanding of facets and exemplars of big data applications. *Proceedings of the 20 Years of Beowulf Workshop on Honor of Thomas Sterling's 65th Birthday*, ACM, 7–16.

Gao, K., C. Jin, A. Choudhary, and W. K. Liao, 2011: Supporting computational data model representation with high-performance I/O in parallel netCDF. *18th International Conference on High Performance Computing, HiPC 2011*, doi:10.1109/HiPC.2011.6152746.

Gustafson, J. L., 1988: Reevaluating amdahl's law. *Communications of the ACM*, **31 (5)**, 532–533.

Hager, G. and G. Wellein, 2010: *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 356 pp., URL http://books.google.com/books?hl=en&lr=&id=rkWPojgfeM8C&pgis=1.

Hayashi, Y., 1971: A Generalized Method of Resolving Disturbances into Progressive and Retrogressive Waves by Space Fourier and Time Cross-Spectral Analyses. *Journal of the Meteorological Society of Japan. Ser. II*, **49 (2)**, 125–128, doi:10.2151/jmsj1965.49.2_125, URL https://www.jstage.jst.go.jp/article/jmsj1965/49/2/49_2_125/_article.

Hayashi, Y., 1982: Space-time spectral analysis and its applications to atmospheric waves. *J. Meteor. Soc. Japan*, **27 (11)**, 156–171, URL https://140.208.31.101/bibliography/related_files/yh8201.pdf.

Hayashi, Y. and D. Golder, 1977: Space-time spectral analysis of mid-latitude disturbances appearing in a GFDL general circulation model. *Journal of the Atmospheric Sciences*, URL https://gfdl.noaa.gov/bibliography/related_files/yh7701.pdf.

He, J., J. Bent, A. Torres, G. Grider, G. Gibson, C. Maltzahn, and X.-H. Sun, 2013: I/O Acceleration with Pattern Detection. *HPDC '13 Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, 25–36, doi:10.1145/2462902.2462909.

Henty, D., A. Jackson, C. Moulinec, and V. Szeremi, 2015: Performance of Parallel IO on ARCHER. URL http://www.archer.ac.uk/documentation/white-papers/parallelIO/ARCHER_wp_parallelIO.pdf, 1–13 pp.

Hildebrand, D. and F. Schmuck, 2015: *High Performance Parallel I/O*, chap. 9, 91–106. CRC Press.

Hoyer, S., 2015: github/shoyer: Pythonic interface to netCDF4 via h5py. URL https://github.com/shoyer/h5netcdf.

Huang, X., Y. Ni, D. Chen, S. Liu, H. Fu, and G. Yang, 2016: Czip: A Fast Lossless Compression Algorithm for Climate Data. *International Journal of Parallel Programming*, **44 (6)**, 1248–1267, doi:10.1007/s10766-016-0403-z, URL http://link.springer.com/10.1007/s10766-016-0403-z.

Hübbe, N., A. Wegener, J. M. Kunkel, Y. Ling, and T. Ludwig, 2013: Evaluating lossy compression on climate data. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, **7905 LNCS**, 343–356, doi:10.1007/978-3-642-38750-0_26.

Jha, S., J. Qiu, A. Luckow, P. Mantha, and G. C. Fox, 2014: A Tale of Two Data-Intensive Paradigms: Applications, Abstractions, and Architectures. *Big Data (BigData Congress), 2014 IEEE International Congress on*, 645–652, URL http://arxiv.org/abs/1403.1528, 1403.1528.

Jiang, S., X. Ding, Y. Xu, and K. Davis, 2013: A prefetching scheme exploiting both data layout and access history on disk. *ACM Transactions on Storage (TOS)*, **9 (3)**, 10.

Jiang-Yu, M. and W. Zai-Zhi, 2008: Actual Tropical Waves Identified by the Wavenumber-Frequency Spectrum Analysis during Boreal Summer. *Chinese Physics Letters*, **1506**, URL http://iopscience.iop.org/0256-307X/25/4/092.

Jones, M., J. Blower, B. Lawrence, and A. Osprey, 2016: Investigating read performance of python and netcdf when using hpc parallel filesystems. *International Conference on High Performance Computing*, Springer, 153–168.

Kunkel, J., 2017: SFS: A Tool for Large Scale Analysis of Compression Characteristics. Tech. Rep. 4, Deutsches Klimarechenzentrum GmbH, Bundesstrae 45a, D-20146 Hamburg.

Lawrence, B., 2014: Jasmin – a data analysis environment, URL http://home.badc.rl.ac.uk/lawrence/static/2014/10/07/Lawrence_JASMIN.pdf, NERC ICT Current Awareness.

Lawrence, B., C. Maynard, A. Turner, X. Guo, and D. Sloan-Murphy, 2017: Partnership for Advanced Computing in Europe Parallel I/O Performance Benchmarking and Investigation on Multiple HPC Architectures. URL http://www.prace-ri.eu/IMG/pdf/WP236.pdf.

Lawrence, B. N., 2001: A gravity-wave induced quasi-biennial oscillation in a three-dimensional mechanistic model. *Quarterly Journal of the Royal Meteorological Society*, **127 (576)**, 2005–2021, doi:10.1002/qj.49712757608, URL http://doi.wiley.com/10.1002/qj.49712757608.

Lawrence, B. N., V. Bennett, J. Churchill, M. Juckes, P. Kershaw, P. Oliver, and M. Pritchard, 2012: The JASMIN super-data-cluster. arXiv:1204.3553v1.

Lawrence, B. N., et al., 2013: Storing and manipulating environmental big data with JASMIN. *IEEE Big Data 2013*.

Lee, C., M. Yang, and R. Aydt, 2008: NetCDF-4 Performance Report. Tech. rep., 1–22 pp. URL https://www.hdfgroup.org/pubs/papers/2008-06_netcdf4_perf_report.pdf.

Li, H., Y. Ruan, Y. Zhou, J. Qiu, and G. Fox, 2011: Design Patterns for Scientific Applications in DryadLINQ CTP. *in Proceedings of The Second International Workshop on Data Intensive Computing in the Clouds (DataCloud-2) 2011, The International Conference for High Performance Computing, Networking, Storage and Analysis (SC11)*, 12–18, URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.300.9577.

Liu, J., E. Racah, Q. Koziol, and R. S. Canon, 2016: H5spark: Bridging the i/o gap between spark and scientific data formats on hpc systems. *Cray User Group*.

Liu, S., X. Huang, H. Fu, G. Yang, and Z. Song, 2015: Data Reduction Analysis for Climate Data Sets. *International Journal of Parallel Programming*, **43 (3)**, 508–527, doi:10.1007/s10766-013-0287-0, URL http://link.springer.com/10.1007/s10766-013-0287-0.

Lofstead, J., M. Polte, G. Gibson, S. Klasky, K. Schwan, R. Oldfield, M. Wolf, and Q. Liu, 2011: Six degrees of scientific data: Reading Patterns for Extreme Scale Science IO. *Proceedings of the 20th international symposium on High performance distributed computing - HPDC '11*, ACM Press, New York, New York, USA, 49, doi:10.1145/1996130.1996139, URL http://dl.acm.org/citation.cfm?id=1996130.1996139.

Lofstead, J. F., S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, 2008: Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments - CLADE '08*, ACM Press, New York, New York, USA, 15, doi:10.1145/1383529.1383533, URL http://dl.acm.org/citation.cfm?id=1383529.1383533.

Mattmann, C. A., D. J. Crichton, N. Medvidovic, and S. Hughes, 2006: A software architecture-based framework for highly distributed and data intensive scientific applications. *Proceeding of the 28th international conference on Software engineering - ICSE '06*, ACM Press, New York, New York, USA, 721, doi:10.1145/1134285.1134400, URL http://portal.acm.org/citation.cfm?doid=1134285.1134400.

Méndez, S., D. Rexachs, and E. Luque, 2013: A Methodology to characterize the parallel I / O of the message-passing scientific applications. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA).*, 8.

Miller, M., 2015: Silo: A Genrral-Purpose API and Scientific Database. *High Performance Parallel I/O*, Prabhat and Q. Koziol, Eds., CRC Press, chap. 21, 249–258.

Palamuttam, R., R. M. Mogrovejo, C. Mattmann, B. Wilson, K. Whitehall, R. Verma, L. McGibbney, and P. Ramirez, 2015: SciSpark: Applying in-memory distributed computing to weather event detection and tracking. *2015 IEEE International Conference on Big Data (Big Data)*, IEEE, 2020–2026, doi:10.1109/BigData.2015.7363983, URL `http://ieeexplore.ieee.org/document/7363983/`.

Rew, R., 2013: Chunking Data: Why it Matters. URL `http://www.unidata.ucar.edu/blogs/developer/entry/chunking_data_why_it_matters`.

Rew, R., C. J. Caron, E. Hartnett, and D. Heimbigner, 2010: December. Advances in the NetCDF Data Model , Format , and Software.

Schmid, J. and J. Kunkel, 2016: Predicting i/o performance in hpc using artificial neural networks. *Supercomputing Frontiers and Innovations*, **3 (3)**, URL `http://superfri.org/superfri/article/view/105`.

Schnase, J. L., et al., 2016: Big Data Challenges in Climate Science: Improving the next-generation cyberinfrastructure. *IEEE Geoscience and Remote Sensing Magazine*, **4 (3)**, 10–22, doi:10.1109/MGRS.2015.2514192, URL `http://ieeexplore.ieee.org/document/7570342/`.

Shan, H., K. Antypas, and J. Shalf, 2008: Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, IEEE Press, 42, URL `http://dl.acm.org/citation.cfm?id=1413370.1413413`.

Silberschatz, A., P. Baer Galvin, and G. Gagne, 2013: *Operating System Concepts*. 9th ed., Wiley.

Stocker, T., 2014: *Climate change 2013: the physical science basis: Working Group I contribution to the Fifth assessment report of the Intergovernmental Panel on Climate Change*. Cambridge University Press.

Wang, T., S. Oral, Y. Wang, B. Settlemyer, S. Atchley, and W. Yu, 2015: BurstMem: A high-performance burst buffer system for scientific applications. *Proceedings - 2014 IEEE International Conference on Big Data, IEEE Big Data 2014*, 71–79, doi:10.1109/BigData.2014.7004215.

Weil, S. A., S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, 2006: Ceph: a scalable, high-performance distributed file system. *Proceedings of the 7th symposium on Operating systems design and implementation*, USENIX Association, 307–320, URL http://dl.acm.org/citation.cfm?id=1298455.1298485.

Welch, B., M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, 2010: White Paper Scalable Performance of the Panasas Parallel File System. *6th USENIX Conference on File and Storage Technologies (FAST '08)*, 1–22, May.

Yang, G.-Y., B. Hoskins, and J. Slingo, 2003: Convectively Coupled Equatorial Waves: A New Methodology for Identifying Wave Structures in Observational Data. *Journal of the Atmospheric Sciences*, **60 (14)**, 1637–1654, doi:10.1175/1520-0469(2003)060¡1637:CCEWAN¿2.0.CO;2, URL http://journals.ametsoc.org/doi/abs/10.1175/1520-0469(2003)060<1637:CCEWAN>2.0.CO;2.

Yang, G.-Y., B. J. Hoskins, and J. M. Slingo, 2011: Equatorial Waves in Opposite QBO Phases. *Journal of the Atmospheric Sciences*, **68 (4)**, 839–862, doi:10.1175/2010JAS3514.1, URL http://journals.ametsoc.org/doi/abs/10.1175/2010JAS3514.1.

Zender, C. S., 2016: Bit Grooming: Statistically accurate precision-preserving quantization with compression, evaluated in the netCDF Operators (NCO, v4.4.8+). *Geoscientific Model Development Discussions*, **(April)**, 1–18, doi:10.5194/gmd-2016-63, URL http://www.geosci-model-dev-discuss.net/gmd-2016-63/.

Appendix A

Platforms

This Appendix contains details on the three HPC platforms used in the thesis. All three were used for Chapter 3, and only JASMIN was used in the other chapters.

A.1 JASMIN (Panasas)

The JASMIN platform (see Figure A.1) at the Science and Technology Facilities Council (STFC) uses a Panasas storage system (Lawrence et al., 2013) . The Panasas sub-system (see Figure A.2) is composed of bladesets, each containing shelves, that in turn contain blades, each made up of two disks. The blades are connected to the shelf via 1Gb/s ports, and the shelves used connected to the network via one 10Gb/s port. This describes the initial JASMIN storage implementation, JASMIN1, which was used for the tests in the thesis. In JASMIN2, the newer section of the filesystem, the shelves are connected via a 2 10Gb/s connections. The compute cluster is composed of nodes, with each connected to the storage system and to each other via a non blocking 10Gb/s network. The Panasas file system handles how the objects are physically stored on disk and the most efficient way to access them, i.e. giving the shortest possible access time (Welch et al., 2010). This gives a theoretical bandwidth to a single processing node of 1.25GB/s. The compute nodes used here have 128GB RAM. In Panasas, files can be striped using the RAID6 method, in which a large file is split across multiple blades (small files are copied) with included redundancy (Silberschatz et al., 2013). This means that the file can be read from multiple different blades increasing the read performance, theoretically, by a factor of the number of blades and bladesets. The manager nodes have the metadata to where and how the files are stored (Welch et al., 2010).

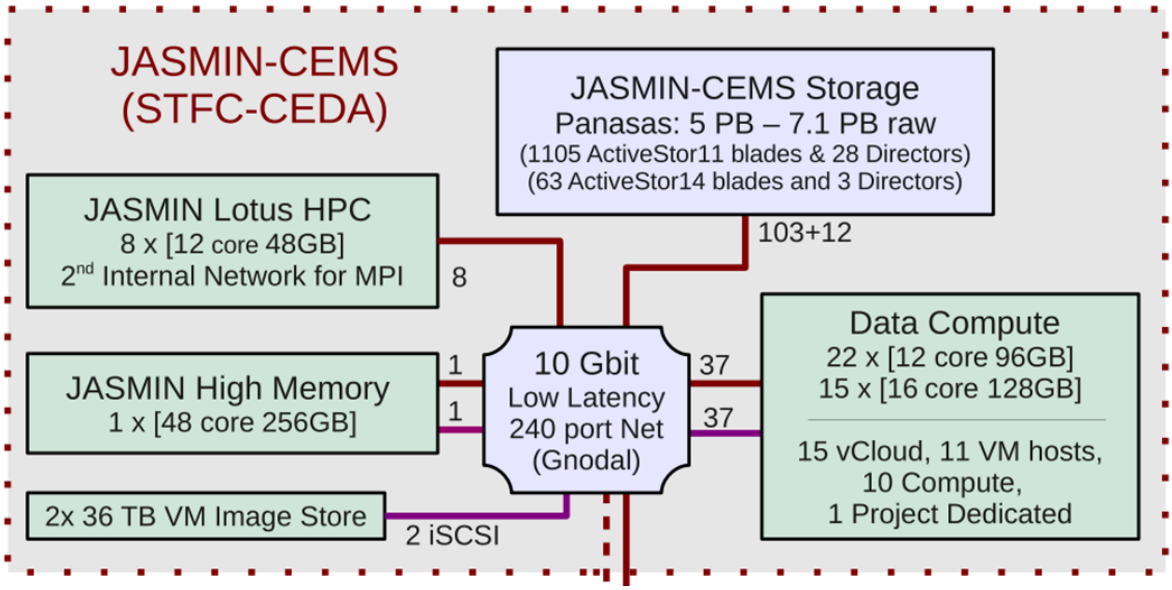


Figure A.1: Representation of the JASMIN platform. Adapted from Lawrence et al. (2013)

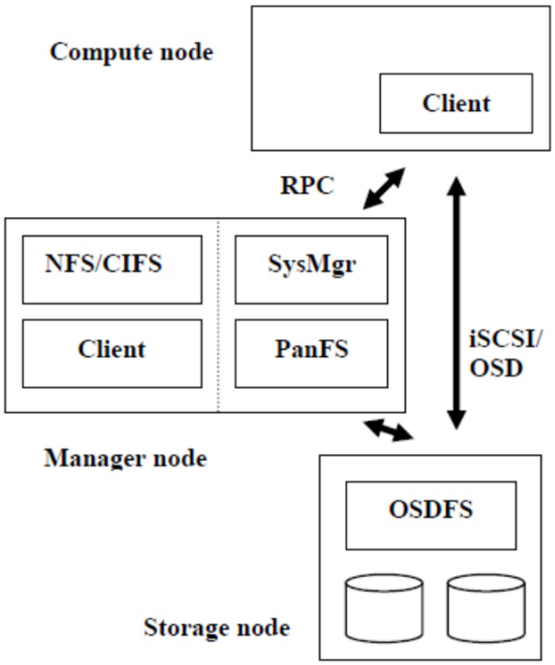


Figure A.2: Representation of the Panasas filesytem. Figure from Welch et al. (2010).

A.2 ARCHER (Lustre)

The ARCHER platform (a Cray XC30) uses a high performance Lustre file system (depiction in Figure A.3). The compute nodes used have 64GB RAM, and are connected by the Cray Aries interconnect[1]. Lustre is an open source parallel file system which uses many object storage servers with metadata servers to store data (Barton and Dilger, 2015). The object storage targets are connected to object storage servers which are all connected to the network, and there is a manager node which is connected to all the object storage severs. Metadata servers are connected directly to the network, and to the object storage serves via the manager node (Barton and Dilger, 2015).

A.3 RDF (GPFS)

The UK Research Data Facility (UK-RDF) HPC platform uses GPFS (General Parallel File System), with Infiniband connections between the storage and compute nodes. The compute nodes used have 128GB RAM. Filesystem metadata is stored on high speed hard disk drives (HDDs), with the data stored on HDDs in four storage arrays [2]. GPFS is a parallel file system that can increase bandwidth by exploiting multiple network shared disks (Hildebrand and Schmuck, 2015). The storage for GPFS is connected to network shared disk servers, which are connected to the network which is in turn connected to the processing nodes (Hildebrand and Schmuck, 2015).

A.4 Comparing the three platform's filesystems

There are a few differences between the Lustre, GPFS, and Panasas files systems. In terms of how files are stored, there are two methods used here – either block based storage, or object based storage. With storage blocks, the block only contains the data itself – it contains no metadata. On the other hand, an object in an object store has the data, metadata about the object, and a global identifier. With both block-based, and object-based storage striping can be used (see Section 2.7.2) – the files are composed of multiple objects or blocks, which can be stored on separate disks.

[1] http://www.archer.ac.uk/documentation/user-guide/
[2] http://www.archer.ac.uk/about-archer/hardware/

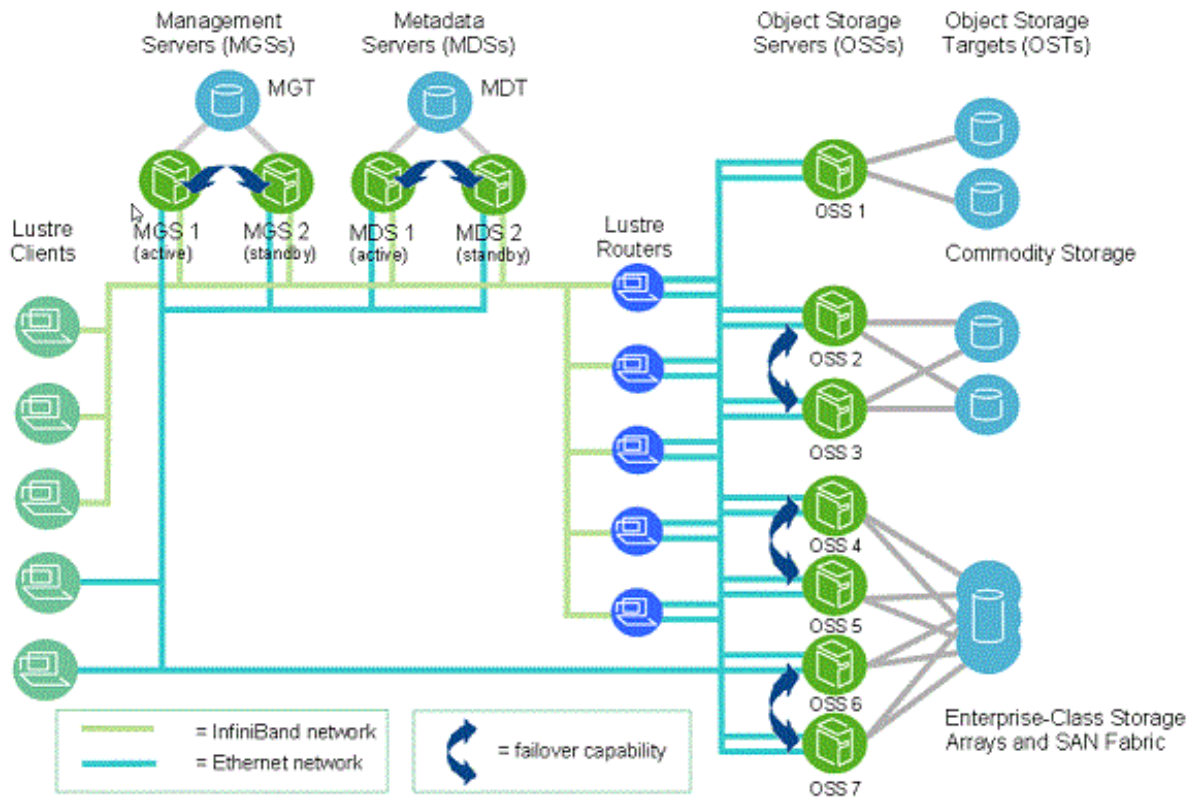


Figure A.3: Representation of a Lustre filesystem. Figure from http://lustre.org/about/

The second difference between the files systems is how metadata servers function. Metadata is needed in order to know where data is stored in the parallel file system. I/O requests from applications are required to query a metadata node before being able to executing reads or writes. This means that the metadata query part of a read could be a significant bottleneck with many users accessing the same file system, or as the filesystem grows in size. In GPFS and Panasas, the metadata servers are involved with file I/O operations (e.g. disc allocation, and data striping), as well as pathname and permissions checks. The metadata server on Lustre, on the other hand, are only involved with pathname and permissions checks in an attempt to avoid scalability bottlenecks on the metadata server. Note that, in all cases, once data has been located, the I/O operations do not have to pass through the metadata node, meaning the only bottleneck is the acquisition of the metadata, and with more metadata nodes the bottleneck is likely to be lower. Panasas in general has a metadata server to storage device ratio of around 1 to 10, as is the case with JASMIN. ARCHER has one metadata node per filesystem (with more recent versions of LUSTRE more metadata servers can be used). The RDF has two metadata nodes (the configuration of GPFS is very flexible are more could potentially be used in other implementations).

Appendix B

Tools

This section contains descriptions of the libraries used in the thesis.

B.1 NetCDF4 C library

The NetCDF4 C library[1] enables reading and writing to NetCDF4 files and the creation of files. Version 4.4.0 was used.

B.2 MPI

MPI is a message passing system which allows tasks to communicate during run time.

B.3 MPI-IO

MPI-IO[2] was developed to provide I/O support for MPI. It supports a number of different I/O modes, including independent and collective I/O (see Chapter 5), and with blocking and non-blocking I/O – this decides whether multiple tasks can access data at the same time.

B.4 Nccopy

Nccopy[3] is a utility which enables the copying of NetCDF4 files, with the ability to change its settings (such as chunking or compression).

[1] https://www.unidata.ucar.edu/software/netcdf/docs/
[2] http://beige.ucs.indiana.edu/I590/node86.html
[3] http://www.unidata.ucar.edu/software/netcdf/workshops/2011/utilities/Nccopy.html

B.5 Python libraries

The following sections contain descriptions Python libraries were used throughout the thesis, along with which version of the library was used.

B.5.1 NumPy

NumPy[4] is a library which allows fast calculations on arrays by implementing said calculations in C rather than natively in Python. It is a fundamental package for other applications because of the speed benefit it applies, and the powerful inbuilt functions, such as statistical calculations, array broadcasting and reshaping, and Fourier transforms. Version 1.13.0 was used.

B.5.2 netCDF4-python

NetCDF4-python[5] is the Python interface to the C NetCDF4 library above. Version 1.2.9 was used, a new version not yet released, 1.3.0, could significantly improve striding read performance[6].

B.5.3 H5py

H5py[7] is a Pythonic interface for reading HDF5 files. This library was not directly used in this work, but the h5netcdf library is built on this one.

B.5.4 H5netcdf

h5netcdf[8] uses the h5py library to read from NetCDF4 files, circumventing the NetCDF4 C library. The library claims to provide faster performance than the the netCDF4-python library, and the results from Chapter 3 back this claim up. Version 0.4.0 was used.

[4] http://www.numpy.org/
[5] https://pypi.python.org/pypi/netCDF4?
[6] https://github.com/Unidata/netcdf4-python/blob/master/Changelog
[7] http://www.h5py.org/
[8] https://pypi.python.org/pypi/h5netcdf/

B.5.5 Jug

Jug[9] was produced as a simple way to write easily parallel programs in Python and is particularly good for embarrassingly parallel problems. Embarrassingly parallel problems are ones in which processes can be immediately and easily split into independent parts which can be executed in parallel. Any workflow that has repeatable sections that do not depend on each other could be easily parallelised using jug. Some examples of this could be calculating spatial analysis for individual timesteps and calculating time based analysis at individual points in space. Jug works by creating a pool of tasks to be executed, and a number of workers to run these tasks – this means that the tasks can be run asynchronously, and any number of tasks and workers can be created, not necessarily the same number. It is possible in a jug script to have multiple pools of tasks and data can flow through them. This can enable a gathering of the data after a pool is finished to move onto another task. In order to ensure that the previous pool is finished barriers can be used in the script to pause the progress until everything before the barrier is finished. Jug is a flexible library which can be used for many different work flows which do not require communication between the tasks. All the control (e.g. which tasks have been run) and data is controlled via a directory jug creates on the file system. Version 0.9.6 was used.

[9] https://jug.readthedocs.io/en/latest/

Appendix C

Code

This section contains the code which was used to run tests in the thesis. It is broken down chapter by chapter and only contains the main bulk of the testing scripts, the excluded scripts are mainly to do with file creation, and submission; the latter enabling concurrent running of the scripts on the cluster.

C.1 Chapter 3, 4, and 5

C.1.1 Program to read using C from 'plain' binary files

Written by Jon Blower, with adaptations by the author. Used in Chapter 3.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/stat.h> /* Only present on POSIX systems I think */
#include <sys/time.h>

double rand2();

int main(int argc, char *argv[])
{
    char *file_name;       /* Name of the file to read */
    FILE *fin;             /* File pointer for the input file */
    off_t file_size;       /* Size of the input file */
    struct stat stbuf;     /* Will hold stats about the input file */

    char random;                        /* if we're reading from random positions in the file, 0 otherwise */
    int random_read_count;              /* The number of buffers requested in a random read */
    off_t *offsets;                     /* If reading randomly, we'll precalculate an array of random offsets */
    int buffer_size;                    /* Number of bytes to read in a single operation */
    char *buffer;                       /* Buffer to hold bytes read from the file */
    unsigned long long buffers_read;    /* The number of buffers actually read */
    size_t bytes_read;                  /* The bytes read in a single read operation */
    unsigned long long total_bytes_read; /* The total number of bytes read */

    int i;                 /* Loop control variables */
    int done;

    clock_t cpu_time_start;   /* Used to get CPU time spent reading the file */
    clock_t cpu_time_end;
    double cpu_time_spent;

    struct timeval wall_time_start; /* Used to measure wall-clock time spent reading the file */
    struct timeval wall_time_end;
    double wall_time_spent;

    double read_rate;    /* Average data read rate in megabytes per wall clock seconds */

    random = 's'; /* Assume we're reading sequentially unless we find otherwise */
    //srand(time(NULL)); /* Seed the random number generator */
```

```
    srand(1);

    /* Check there are enough command-line arguments */
    if (argc < 3 || argc > 5) {
        fprintf(stderr, "Usage: readfile <filename> <buffer size in bytes> [r|s|h] [count if random]\n");
        exit(EXIT_FAILURE);
    }

    /* Read the command-line arguments */
    file_name = argv[1];
    if (sscanf(argv[2], "%d", &buffer_size) <= 0) {
        fprintf(stderr, "Invalid buffer size %s\n", argv[2]);
        exit(EXIT_FAILURE);
    }
    if (argc > 3) {
        /* Third argument is a character indicating if reads should be random, sequential or hopping */
        sscanf(argv[3], "%c", &random);
        if (random != 'r' && random != 's' && random != 'h') {
            fprintf(stderr, "Must be \"r\", \"s\" or \"h\", found %s\n", argv[3]);
            exit(EXIT_FAILURE);
        }
    }
    if (random == 'r') {
        /* We need a count */
        if (argc < 5) {
            fprintf(stderr, "Must input a count when performing random reads\n");
            exit(EXIT_FAILURE);
        }
        if(sscanf(argv[4], "%d", &random_read_count) <= 0) {
            fprintf(stderr, "Invalid read count %s\n", argv[4]);
            exit(EXIT_FAILURE);
        }
    }

    /* Create buffer of the right size */
    printf("Buffer size = %d bytes\n", buffer_size);
    buffer = (char*) malloc(sizeof(char) * buffer_size);
    if (buffer == NULL) {
        fprintf(stderr, "Error allocating buffer of size %d bytes", buffer_size);
        exit(EXIT_FAILURE);
    }

    /* Find the size of the file, using code from
       https://www.securecoding.cert.org/confluence/display/c/FIO19-C.+Do+not+use+fseek%28%29+and+ftell%28%29+to+compute+the+
           ↪ size+of+a+regular+file */
    if ((stat(file_name, &stbuf) != 0) || (!S_ISREG(stbuf.st_mode))) {
        fprintf(stderr, "Error getting size of file %s\n", file_name);
        exit(EXIT_FAILURE);
    }
    file_size = stbuf.st_size;
    printf("File size: %llu bytes\n", file_size);

    if (random == 'r') {
        /* Precalculate the array of offsets, so we don't cost CPU time during the read operation
           (I'm not sure this really makes a difference, we could probably compute these on the fly) */
        offsets = (off_t*) malloc(sizeof(off_t) * random_read_count);
        if (offsets == NULL) {
            fprintf(stderr, "Error allocating space for %d precalculated random offsets\n", random_read_count);
            exit(EXIT_FAILURE);
        }
        for (i = 0; i < random_read_count; i++) {
            offsets[i] = rand2() * file_size;
            /* Uncomment the line below to simulate sequential read as a test */
            //offsets[i] = i * (off_t)buffer_size;
        }
    }

    /* Open the file */
    fin = fopen(file_name, "r");
    if (fin == NULL) {
        fprintf(stderr, "Error opening file %s\n", file_name);
        exit(EXIT_FAILURE);
    }

    /* Turn off underlying buffering used by fread()
       (Although I don't think it makes much difference) */
    setvbuf(fin, NULL, _IONBF, 0);

    /* Initialise count variables */
    buffers_read = 0;
    bytes_read = 0;
    total_bytes_read = 0;

    /* Make a note of the time */
    gettimeofday(&wall_time_start, NULL);
    cpu_time_start = clock();

    /* Now read the data from the file */
    if (random == 's') {
        /* Read sequentially from the start until the end of the file */
        printf("Reading whole file sequentially...\n");
        while (!feof(fin)) {
            bytes_read = fread(buffer, sizeof(char), buffer_size, fin);
            buffers_read++;
            total_bytes_read += bytes_read;
        }
    } else if (random == 'r') {
        /* Read the specified number of times from random positions within the file */
        printf("Reading data from file randomly...\n");
```

```
        for (i = 0; i < random_read_count; i++) {
            //printf("Offsets[%d] = %llu\n", i, offsets[i]);
            /* Seek to the precomputed offset */
            if (fseeko(fin, offsets[i], SEEK_SET) < 0) {
                fprintf(stderr, "Error seeking to %llu, aborting due to internal error\n", offsets[i]);
                exit(EXIT_FAILURE);
            }
            /* Now read a buffer from this location */
            bytes_read = fread(buffer, sizeof(char), buffer_size, fin);
            buffers_read++;
            total_bytes_read += bytes_read;
        }
    } else {
        /* Read in a "hopping" pattern reading a block and then skipping three blocks */
        printf("Reading file in \"hopping\" pattern...\n");
        done = 0;
        while (!done) {
            /* Read a buffer */
            bytes_read = fread(buffer, sizeof(char), buffer_size, fin);
            buffers_read++;
            total_bytes_read += bytes_read;
            /* Skip three buffers' worth of data */
                if (fseeko(fin, buffer_size * 3, SEEK_CUR) < 0) {
                    fprintf(stderr, "Error in seek, aborting due to internal error\n");
                    exit(EXIT_FAILURE);
                }
                if (bytes_read < buffer_size) {
                    done = 1;
                }
        }
    }

    /* Note the time again. */
    cpu_time_end = clock();
    gettimeofday(&wall_time_end, NULL);

    /* Calculate the wall-clock and CPU time spent */
    cpu_time_spent = (double) (cpu_time_end - cpu_time_start) / CLOCKS_PER_SEC;
    wall_time_spent = (double)(wall_time_end.tv_usec - wall_time_start.tv_usec) / 1e6 +
                      (double)(wall_time_end.tv_sec  - wall_time_start.tv_sec);

    /* Calculate the read rate in MB per sec (10^6 bytes per wall clock second) */
    read_rate = total_bytes_read / (wall_time_spent * 1e6);

    /* Output statistics as comma separated values */
    /* File size (bytes) | Bytes read | Buffer size (bytes) | No. buffers | Seq or random | CPU time (s) | Wall time (s) |
        ↪ Read rate (MB/s) */
    printf ("%llu,%llu,%d,%llu,%s,%f,%f,%f\n",
        file_size,
        total_bytes_read,
        buffer_size,
        buffers_read,
        random == 'r' ? "random" : (random == 's' ? "sequential" : "hopping"),
        cpu_time_spent,
        wall_time_spent,
        read_rate
    );

    /* Close the file */
    fclose(fin);
    free(buffer);
    if (random == 'r') {
        free(offsets);
    }
    return 0;
}

/* Return a random number between 0.0 and 1.0 inclusive.
   Must seed the random number generator before this is called. */
double rand2()
{
        return (double)rand() / (double)RAND_MAX;
}
```

C.1.2 Program to read using C from NetCDF4 files

Adapted from C.1.1 by Annette Osprey, and the author. This was used in Chapter 3, and Chapter 5.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/stat.h> /* Only present on POSIX systems I think */
#include <sys/time.h>
#include <netcdf.h>

#define ERRCODE 2
#define ERR(e) {printf("Error: %s\n", nc_strerror(e)); exit(ERRCODE);}

double rand2();
```

```
int main(int argc, char *argv[])
{
    char *file_name;        /* Name of the file to read */
    int ncid;               /* Id of netcdf file */
    int dimid;              /* Id of dimension "dim1" */
    size_t dimlen;          /* Size of the dimension (number of elements in data variable) */
    int varid;              /* Id of data variable to be read from file */
    int retval;             /* Return code used to check status of netcdf function calls */

    char random;                        /* if we're reading from random positions in the file, 0 otherwise */
    unsigned long long random_read_count;               /* The number of buffers requested in a random read */
    size_t *offsets;                    /* If reading randomly, we'll precalculate an array of random offsets */
    int buffer_size;                    /* Number of bytes to read in a single operation */
    double *buffer;                     /* Buffer to hold data read from the file */
    unsigned long long buffer_len;                      /* Number of elements in buffer (= buffer_size/8) */
    unsigned long long num_buffers;     /* The number of buffers to be read in */
    unsigned long long total_bytes_read; /* The total number of bytes read */
    unsigned long long pos;                             /* Position of data to be read */
    size_t start[1];                    /* Start location for each netCDF read */
    size_t count[1];                    /* Size of each netCDF read */

    int i;                  /* Loop control variables */

    clock_t cpu_time_start;   /* Used to get CPU time spent reading the file */
    clock_t cpu_time_end;
    double cpu_time_spent;

    struct timeval wall_time_start; /* Used to measure wall-clock time spent reading the file */
    struct timeval wall_time_end;
    double wall_time_spent;

    double read_rate;    /* Average data read rate in megabytes per wall clock seconds */

    random = 's'; /* Assume we're reading sequentially unless we find otherwise */
    srand(time(NULL)); /* Seed the random number generator */

    /* Check there are enough command-line arguments */
    if (argc < 3 || argc > 5) {
        fprintf(stderr, "Usage: readfile <filename> <buffer size in bytes> [r|s|h] [count if random]\n");
        exit(EXIT_FAILURE);
    }

    /* Read the command-line arguments */
    file_name = argv[1];
    if (sscanf(argv[2], "%d", &buffer_size) <= 0) {
        fprintf(stderr, "Invalid buffer size %s\n", argv[2]);
        exit(EXIT_FAILURE);
    }
    if (argc > 3) {
        /* Third argument is a character indicating if reads should be random, sequential or hopping */
        sscanf(argv[3], "%c", &random);
        if (random != 'r' && random != 's' && random != 'h') {
            fprintf(stderr, "Must be \"r\", \"s\" or \"h\", found %s\n", argv[3]);
            exit(EXIT_FAILURE);
        }
    }
    if (random == 'r') {
        /* We need a count */
        if (argc < 5) {
            fprintf(stderr, "Must input a count when performing random reads\n");
            exit(EXIT_FAILURE);
        }
        if(sscanf(argv[4], "%d", &random_read_count) <= 0) {
            fprintf(stderr, "Invalid read count %s\n", argv[4]);
            exit(EXIT_FAILURE);
        }
    }

    /* Create buffer of the right size */
    printf("Buffer size = %llu bytes\n", buffer_size);
    buffer_len = buffer_size/8;                     /* use this variable for netcdf reads which are in elements */
    buffer = (double*)malloc(sizeof(double)*buffer_len);
    if (buffer == NULL) {
        fprintf(stderr, "Error allocating buffer of size %d bytes", buffer_size);
        exit(EXIT_FAILURE);
    }

    /* Open the file and get the data variable id - assuming this is called "var" */
    if ((retval = nc_open(file_name, NC_NOWRITE, &ncid)))
       ERR(retval);
    if ((retval = nc_inq_varid(ncid, "var", &varid)))
       ERR(retval);

    /* Find the dimension of the data variable - assuming this is called "dim1" */
    if ((retval = nc_inq_dimid(ncid, "dim1", &dimid)))
       ERR(retval);
    if ((retval = nc_inq_dimlen(ncid, dimid, &dimlen)))
       ERR(retval);
    printf("Data size: %llu bytes\n", dimlen*8);

    if (random == 'r') {
        /* Precalculate the array of offsets, so we don't cost CPU time during the read operation
           (I'm not sure this really makes a difference, we could probably compute these on the fly) */
        offsets = (size_t*) malloc(sizeof(size_t) * random_read_count);
        if (offsets == NULL) {
            fprintf(stderr, "Error allocating space for %d precalculated random offsets\n", random_read_count);
            exit(EXIT_FAILURE);
        }
        for (i = 0; i < random_read_count; i++) {
```

```
            offsets[i] = rand2() * (dimlen-buffer_len);
            /* Uncomment the line below to simulate sequential read as a test */
            //offsets[i] = i * (off_t)buffer_size;
        }
    }

    /* Initialise position and count variable */
    pos = 0;
    count[0] = buffer_len;

    /* Make a note of the time */
    gettimeofday(&wall_time_start, NULL);
    cpu_time_start = clock();

    /* Now read the data from the file */
    if (random == 's') {
        /* Read sequentially from the start until the end of the file */
        printf("Reading whole file sequentially...\n");
        num_buffers = dimlen / buffer_len;
        printf("Reading %d buffers\n", num_buffers);
        for (i = 0; i < num_buffers; i++){
           start[0] = pos;
           if ((retval = nc_get_vara_double(ncid, varid, start, count, &buffer[0])))
              ERR(retval);
           pos = pos + buffer_len;
        }
        total_bytes_read = num_buffers * buffer_size;

    }  else if (random == 'r') {
        /* Read the specified number of times from random positions within the file */
        printf("Reading data from file randomly...\n");
        printf("Reading %d buffers\n", random_read_count);
        for (i = 0; i < random_read_count; i++) {
          start[0] = offsets[i];
          if ((retval = nc_get_vara_double(ncid, varid, start, count, &buffer[0])))
             ERR(retval);
        }
        total_bytes_read = random_read_count * buffer_size;

    } else {
        /* Read in a "hopping" pattern reading a block and then skipping three blocks */
        printf("Reading file in \"hopping\" pattern...\n");
        num_buffers = (dimlen / buffer_len) / 4;
        printf("Reading %d buffers\n", num_buffers);
        for (i = 0; i < num_buffers; i++){
           start[0] = pos;
           if ((retval = nc_get_vara_double(ncid, varid, start, count, &buffer[0])))
              ERR(retval);
           pos = pos + buffer_len*4;
        }
        total_bytes_read = num_buffers * buffer_size;
    }

    /* Note the time again. */
    cpu_time_end = clock();
    gettimeofday(&wall_time_end, NULL);

    /* Calculate the wall-clock and CPU time spent */
    cpu_time_spent = (double) (cpu_time_end - cpu_time_start) / CLOCKS_PER_SEC;
    wall_time_spent = (double)(wall_time_end.tv_usec - wall_time_start.tv_usec) / 1e6 +
                      (double)(wall_time_end.tv_sec  - wall_time_start.tv_sec);

    /* Calculate the read rate in MB per sec (10^6 bytes per wall clock second) */
    read_rate = total_bytes_read / (wall_time_spent * 1e6);

    /* Output statistics as comma separated values */
    /* File size (bytes) | Bytes read | Buffer size (bytes) | No. buffers | Seq or random | CPU time (s) | Wall time (s) |
        ↪ Read rate (MB/s) */
    printf ("%llu,%llu,%d,%llu,%s,%f,%f,%f\n",
        dimlen*8,
        total_bytes_read,
        buffer_size,
        num_buffers,
        random == 'r' ? "random" : (random == 's' ? "sequential" : "hopping"),
        cpu_time_spent,
        wall_time_spent,
        read_rate
    );

    /* Close the file */
    if ((retval = nc_close(ncid)))
      ERR(retval);
    free(buffer);
    if (random == 'r') {
        free(offsets);
    }
    return 0;
}

/* Return a random number between 0.0 and 1.0 inclusive.
   Must seed the random number generator before this is called. */
double rand2()
{
        return (double)rand() / (double)RAND_MAX;
}
```

C.1.3 Program to read using Python from 'plain' binary files

Written by the author. Used in Chapter 3.

```
#!/usr/bin/env python2.7

import os
import sys
import numpy as np
from time import time, clock
from numpy.random import random


def hop_read(fpath, block_size, file_size):
    """
    Hopping read mode: reads one block size then skips three times the block size.

    :param fpath: path to the file
    :param block_size: size of the blocks to read
    :param file_size: total size of the file
    :return: wall_time: wall time for the total read
    :return: cpu_time: cpu time for the total read (on windows system this will also be wall time)
    :return: total_bytes_read: total number of bytes read from the file
    :return: buffers_read: total number of blocks read from the file
    """

    num_buffers = np.floor(file_size/block_size)
    hop_size = 4
    wall_time = time()
    cpu_time = clock()
    total_bytes_read = 0L
    buffers_read = 0
    f = open(fpath, 'rb')
    # read one buffer, skip three then read the next
    for i in np.arange(0,num_buffers,hop_size):
        f.seek(i*block_size)
        bytes_read = len(f.read(block_size))
        total_bytes_read += bytes_read
        buffers_read += 1
    f.close()

    wall_time = time()-wall_time
    cpu_time = clock()-cpu_time

    return wall_time, cpu_time, total_bytes_read, buffers_read


def rand_read(fpath, block_size, file_size, block_num):
    """
    Random read mode: read the specified number of blocks from the file at random positions.
    Uses uniform random numbers.

    :param fpath: path to the file
    :param block_size: size of the blocks to read
    :param file_size: total size of the file
    :param block_num: number of blocks to read from file
    :return: wall_time: wall time for the total read
    :return: cpu_time: cpu time for the total read (on windows system this will also be wall time)
    :return: total_bytes_read: total number of bytes read from the file
    :return: buffers_read: total number of blocks read from the file
    """
    # Generate random numbers
    rand_starts = [np.ceil(x) for x in random(block_num)*file_size]
    f_rand = open('rand_nums','a')
    print >> f_rand, rand_starts
    f_rand.close()
    wall_time = time()
    cpu_time = clock()
    total_bytes_read = 0L
    buffers_read = 0

    f = open(fpath, 'rb')
    for i in rand_starts:
        f.seek(i)
        bytes_read = len(f.read(block_size))
        total_bytes_read += bytes_read
        buffers_read += 1

    f.close()
    wall_time = time()-wall_time
    cpu_time = clock()-cpu_time

    return wall_time, cpu_time, total_bytes_read, buffers_read


def seq_read(fpath, block_size, file_size):
    """
    Sequential read mode: reads whole file sequentially.

    :param fpath: path to the file
    :param block_size: size of the blocks to read
    :param file_size: total size of the file
    :return: wall_time: wall time for the total read
    :return: cpu_time: cpu time for the total read (on windows system this will also be wall time)
    :return: total_bytes_read: total number of bytes read from the file
    :return: buffers_read: total number of blocks read from the file
```

```
    """
    num_buffers = file_size/block_size
    total_bytes_read = 0L
    wall_time = time()
    cpu_time = clock() # returns cpu time on linux, wall time on windows
    full_block_read = True
    f = open(fpath, 'rb')
    while full_block_read:
        #for i in np.arange(num_buffers): #~~~ change to while
        bytes_read = len(f.read(block_size))
        full_block_read = bytes_read == block_size
        total_bytes_read += bytes_read

    f.close()

    wall_time = time()-wall_time
    cpu_time = clock()-cpu_time

    return wall_time, cpu_time, total_bytes_read, num_buffers


def main(fpath, read_mode, block_size, block_num):

    # Find the size of the file
    file_size = float(os.path.getsize(fpath))

    if read_mode == 's':
        read_time, cpu_time, bytes_read, num_buffers = seq_read(fpath, block_size, file_size)
        print_read_mode = 'sequential'
    elif read_mode == 'r':
        read_time, cpu_time, bytes_read, num_buffers = rand_read(fpath, block_size, file_size, block_num)
        print_read_mode = 'random'
    elif read_mode == 'h':
        read_time, cpu_time, bytes_read, num_buffers = hop_read(fpath, block_size, file_size)
        print_read_mode = 'hopping'
    else:
        raise ValueError('Read_mode_error.\n\tUsage:_python_readfile.py_filename_s|r|h_blocksize_[randnum]\n')

    rate = (bytes_read/read_time)/1000**2

    print '%d,%d,%d,%d,%s,%f,%f,%f' % (file_size, bytes_read, block_size, num_buffers, print_read_mode, cpu_time, read_time,
        ↪ rate)

if __name__ == '__main__':
    # Usage: python readfile.py filename readmode blocksize randnum
    if len(sys.argv)<3:
        raise ValueError('Not_enough_arguments.\n\tUsage:_python_readfile.py_filename_s|r|h_blocksize_[randnum]\n')
    fpath = sys.argv[1]
    read_mode = sys.argv[2]
    block_size = long(sys.argv[3])
    if len(sys.argv) == 5:
        block_num = int(sys.argv[4])
    else:
        block_num = None

    if read_mode == 'r' and block_num == None:
        raise ValueError('Needs_number_of_buffers_for_random_mode.\n\tUsage:_python_readfile.py_filename_s|r|h_blocksize_[
            ↪ randnum]\n')

    main(fpath, read_mode, block_size, block_num)
```

C.1.4 Program to read using Python from NetCDF4 files

Written by the author. Used in Chapters 3, 4, and 5.

```
#!/usr/bin/env python2.7

import sys
import numpy as np
from netCDF4 import Dataset
from time import time, clock

def seq_read_1d(fid, num_elements):

    f = Dataset(fid,'r')
    var = f.variables['var']

    num_reads = var.shape/num_elements

    bytes_read = 0
    start = time()
    cpu_time = clock()
    for i in xrange(num_reads):
        ind_slice0 = long(i*num_elements)
        ind_slice1 = long(i*num_elements+num_elements)
        bytes_read += 8*len(var[ind_slice0:ind_slice1])

    cpu_time = clock() - cpu_time
    wall_time = time()-start
    rate = bytes_read/wall_time

    print '%s,%s,%s,%s,sequential,%s,%s,%s'\
```

```
        % (var.shape[0]*8, bytes_read, num_elements*8, num_reads[0], cpu_time, wall_time, rate/1000**2)

def hop_read_1d(fid, num_elements):

    f = Dataset(fid,'r')
    var = f.variables['var']

    num_reads = var.shape/num_elements

    bytes_read = 0
    start = time()
    cpu_time = clock()
    for i in xrange(0,num_reads,4):
        ind_slice0 = long(i*num_elements)
        ind_slice1 = long(i*num_elements+num_elements)
        bytes_read += 8*len(var[ind_slice0:ind_slice1])

    cpu_time = clock() - cpu_time
    wall_time = time()-start
    rate = bytes_read/wall_time

    print '%s,%s,%s,%s,hopping,%s,%s,%s'\
        % (var.shape[0]*8, bytes_read, num_elements*8, num_reads[0], cpu_time, wall_time, rate/1000**2)

def rand_read(fid, num_elements, rand_num):

    f = Dataset(fid,'r')
    var = f.variables['var']


    rand_starts = [int(np.ceil(x)) for x in np.random.random(rand_num)*(var.shape-num_elements)]
    if rand_starts <0:
        rand_starts = 0

    bytes_read = 0
    start = time()
    cpu_time = clock()
    for ind1 in rand_starts:
        ind2 = int(ind1+num_elements)
        bytes_read += 8*len(var[ind1:ind2])

    cpu_time = clock() - cpu_time
    wall_time = time()-start
    rate = bytes_read/wall_time

    print '%s,%s,%s,%s,random,%s,%s,%s'\
        % (var.shape[0]*8, bytes_read, num_elements*8, rand_num, cpu_time, wall_time, rate/1000**2)

def readfile_1d(fid, readmode, readsize, rand_num):
    # Readsize in elements
    num_elements = np.ceil(readsize/8.)

    if readmode == 's':
        seq_read_1d(fid, num_elements)
    elif readmode == 'h':
        hop_read_1d(fid, num_elements)
    elif readmode == 'r':
        assert rand_num != None, 'For random read the number of reads needs to be specified'
        rand_read(fid, num_elements, rand_num)

if __name__ == '__main__':
    fid = sys.argv[1]
    readmode = sys.argv[2]
    readsize = float(sys.argv[3])
    if len(sys.argv) == 5:
        rand_num = int(sys.argv[4])
    else:
        rand_num = None

    readfile_1d(fid, readmode, readsize, rand_num)
```

C.2 Chapter 6

C.2.1 Read only test program

Written by the author.

```
#!/usr/bin/env python2.7

# Imports
from netCDF4 import Dataset
import numpy as np
from jug import TaskGenerator, barrier
from sys import argv
from time import time, clock

# File name depending on test file
fname = '/group_workspaces/jasmin/hiresgw/vol1/mj07/IO_testing_files/comp_test_u_c1bg.nc'#argv[1]
```

```
# open file
nc = Dataset(fname)
var = nc.variables['u']

# inputs
jworkers = argv[1]

@TaskGenerator
def read_section(read_start):
    # Get test number
    runfile = open('runnum','r')
    test = int(runfile.read().strip())
    runfile.close()

    # start timer
    start_time = time()
    start_clock = clock()
    data = np.zeros([240*40,48,1024])
    io_time = time()
    io_clock = clock()
    # var is dimensioned (240, 180, 768, 1024)
    # we're dividing y=768 into 16 tasks, each with 48 lat members.
    for io in range(40):
        # n is an index into latitude,
        # i is an index into the file to find those latitudes
        # we stride through using height as an extra pseudo time
        # dimension ... to make longer FFTS (nearly four years)
        for n,i in enumerate(range(read_start,read_start+48)):
            # we read 40x48x(240,1,1,1024) so 240 longitude circles
            # 48x40x(240x1024x8 bytes per read 2MB with 240 reads
            # internally to fill the buffer since only x is contigous)
            data[io*240:(io+1)*240,n,:] = var[:,io,i,:]
    io_end = time()-io_time
    io_clockend = clock()-io_clock
    cpu_start = time()
    cpu_clock = clock()
    # should do fft over longitude first, but we do time
    # here for historical reasons and it doesn't matter because
    # this is only a timing test.
    stg1 = np.fft.fft(data,axis=0)
    # then it should be fft over time (but we do longitude ... but heh!)
    stg2 = np.fft.fft(stg1,axis=2)
    cpu_end = time()-cpu_start
    cpu_clockend = clock() - cpu_clock
    logfile = open('output.csv','a')
    print >> logfile, '%s,CPU,%s,%s,%s,%s,%s,%s,%s,%s,%s' % (test, fname, jworkers, read_start, time()-start_time, clock()-
        ↪ start_clock, io_end, io_clockend, cpu_end, cpu_clockend)
    logfile.close()


@TaskGenerator
def starttime(innum):
    runfile = open('runnum','r')
    new = int(runfile.read().strip())+1
    runfile.close()
    runfile = open('runnum','w')
    print >> runfile, new
    runfile.close()
    wallfile = open('walltimes','a')
    print >> wallfile, '%s,start:%s,CPU,%s,%s' % (new,time(),fname, jworkers)
    wallfile.close()

@TaskGenerator
def endtime(innum):
    runfile = open('runnum','r')
    test = int(runfile.read().strip())
    runfile.close()
    wallfile = open('walltimes','a')
    print >> wallfile, '%s,end:%s,CPU,%s,%s' % (test,time(),fname, jworkers)
    wallfile.close()

map(starttime,[1])

barrier()

task_starts = np.arange(16)*48

map(read_section,task_starts)

barrier()

map(endtime,[1])
```

C.2.2 Test program including STSA calculations

Adapted from C.2.1 with STSA calculation code written by Bryan Lawrence.

```
#!/usr/bin/env python2.7

# Imports
from netCDF4 import Dataset
```

```
import numpy as np
from jug import TaskGenerator, barrier
from sys import argv
from time import time, clock

# File name depending on test file
fname = '/group_workspaces/jasmin/hiresgw/vol1/mj07/IO_testing_files/comp_test_u_c0.nc'#argv[1]

# open file
nc = Dataset(fname)
var = nc.variables['u']

# inputs
jworkers = argv[1]

# Using Bryan's hayashi code from https://bitbucket.org/bnlawrence/stsalib/src/7d01e689f0c7dd0c4f9b848af6ba6b185d8477b7/
    ↪ hayashi.py?at=master&fileviewer=file-view-default
def farm_x(array):
    ''' Find and remove the zonal mean of an array '''
    mm=np.mean(array,axis=1)
    #newaxis? makes the one dimensional mean two dimensional so a broadcast works
    return array-mm[:,np.newaxis],mm

def abft(series,**kw):
    ''' Return the cosine and sine coefficients of a Fourier Transform
    in a,b notation '''
    ff=np.fft.rfft(series,**kw)
    a=np.real(ff)
    b=np.imag(ff)
    return a,b

def hayashi(array):
    ''' This method calcluates the eastward and westward power from
    a data set which has been constructed by stacking longitudinal
    sections into an array, so that it has a shape something like:
      (365,180) - for 365 instances of 180 longitudinal points.

    The code is based on my Hayashi01 code, which itself has the
    following references:
                Venne Thesis p 26
                Hayashi, 1971, J Met Soc 49, p 127
                Me, Appendix 2, pages 227... '''

    #first remove the daily zonal means
    array,dayzm=farm_x(array)

    # next, we need to calculate ffts to get the spatial fourier coefficients
    c,s=abft(array)

    #now we need to remove the mean in those
    c,cm=farm_x(c)
    s,sm=farm_x(s)

    #now do the Fourier Transforms
    BigA,BigB=abft(c,axis=0)

    SmallA,SmallB=abft(s,axis=0)

    stationary=np.sqrt(cm**2+sm**2)/2

    n2=array.shape[1]/2

    # ------------------------------------------------------------------
    # In the Fortran era I used these, even though they weren't "right",
    # according to my analysis, but probably an artifact of the fft
    # routines I used then (and checked with tests at the time).
    # Eastward=np.sqrt( (BigA-SmallB)**2 + (-BigB-SmallA)**2 )/2
    # Westward=np.sqrt( (BigA+SmallB)**2 + (BigB-SmallA)**2 )/2
    # Now in 2012, my versions (which follow) pass unit tests with numpy fft,
    # so happy to got to something I have algebra for:
    #
    Eastward=np.sqrt( (BigA+SmallB)**2 + (BigB-SmallA)**2 )/2
    Westward=np.sqrt( (BigA-SmallB)**2 + (BigB+SmallA)**2 )/2
    #
    # ------------------------------------------------------------------

    return stationary,Eastward,Westward


@TaskGenerator
def read_section(read_start):
    # Get test number
    runfile = open('runnum','r')
    test = int(runfile.read().strip())
    runfile.close()

    # start timer
    start_time = time()
    start_clock = clock()
    data = np.zeros([240*40,48,1024])
    io_time = time()
    io_clock = clock()
    # var is dimensioned (240, 180, 768, 1024)
    # we're dividing y=768 into 16 tasks, each with 48 lat members.
    for io in range(40):
        # n is an index into latitude,
        # i is an index into the file to find those latitudes
        # we stride through using height as an extra pseudo time
        # dimension ... to make longer FFTS (nearly four years)
```

```
        for n,i in enumerate(range(read_start,read_start+48)):
            # we read 40x48x(240,1,1,1024) so 240 longitude circles
            # 48x40x(240x1024x8 bytes per read 2MB with 240 reads
            # internally to fill the buffer since only x is contigous)
            data[io*240:(io+1)*240,n,:] = var[:,io,i,:]
    io_end = time()-io_time
    io_clockend = clock()-io_clock
    #start cpu timers
    cpu_start = time()
    cpu_clock = clock()
    stationary = 0
    Eastward = 0
    Westward =0
    for i in range(48):
        stationarytmp,Eastwardtmp,Westwardtmp = hayashi(data[:,i,:])
        stationary += stationarytmp
        Eastward += Eastwardtmp
        Westward += Westwardtmp
    # take mean for each
    stationary = stationary/48.
    Eastward = Eastward/48.
    Westward = Westward/48.
    # end cpu timers
    cpu_end = time()-cpu_start
    cpu_clockend = clock() - cpu_clock
    logfile = open('output.csv','a')
    print >> logfile, '%s,CPU,%s,%s,%s,%s,%s,%s,%s,%s,%s' % (test, fname, jworkers, read_start, time()-start_time, clock()-
        ↪ start_clock, io_end, io_clockend, cpu_end, cpu_clockend)
    logfile.close()


@TaskGenerator
def starttime(innum):
    runfile = open('runnum','r')
    new = int(runfile.read().strip())+1
    runfile.close()
    runfile = open('runnum','w')
    print >> runfile, new
    runfile.close()
    wallfile = open('walltimes','a')
    print >> wallfile, '%s,start:%s,CPU,%s,%s' % (new,time(),fname, jworkers)
    wallfile.close()

@TaskGenerator
def endtime(innum):
    runfile = open('runnum','r')
    test = int(runfile.read().strip())
    runfile.close()
    wallfile = open('walltimes','a')
    print >> wallfile, '%s,end:%s,CPU,%s,%s' % (test,time(),fname, jworkers)
    wallfile.close()

map(starttime,[1])

barrier()

task_starts = np.arange(16)*48

map(read_section,task_starts)

barrier()

map(endtime,[1])
```