Department of Information Engineering and Computer Science

Master's Degree in
Computer Science

Final Dissertation

Benchmarking Zarr file storage format on climate data in HPC Environments

Supervisor

Fiore Sandro Luigi

Student

Marchioro Nicola Giuseppe

External Supervisor

Anantharaj Valentine

Academic year 2022/2023

Acknowledgements

...thanks to...

Contents

Abstract

In the ever-evolving landscape of scientific research and data analysis, proficiently managing complex and multidimensional datasets is a paramount challenge. Diverse data requirements, intricately intertwined with hardware infrastructure, implementation tools, and the unique characteristics of scientific applications, necessitate innovative solutions. This thesis embarks on a journey into the realm of data format optimization and data management strategies within the field of meteorology.

High-Performance Computing clusters serve as indispensable resources for advancing data management practices, offering the computational capabilities required for groundbreaking discoveries. Key to these advancements is the selection of an appropriate scientific data format. The choice of format has far-reaching implications, significantly influencing data management and accessibility in the realm of scientific research. Various formats, including HDF5, netCDF4, and Zarr, each bring distinct advantages that cater to diverse use cases. With access to the advanced computing resources of Oak Ridge National Laboratory (ORNL), this research embarks on an exploration of the emerging Zarr data format. The focal point is the parallelized conversion of NetCDF files into Zarr format, conducted within HPC environments, notably the Summit supercomputer. This study aims to assess the viability of Zarr as an alternative to the widely-used NetCDF format.

This research has revealed valuable insights into the conversion of NetCDF to Zarr, shedding light on the significance of benchmarking data formats. The findings underscore their potential for innovation and have laid the foundation for future research to refine and expand upon these discoveries. As scientific data continues to grow in complexity and volume, the effective management and utilization of such data will be a defining factor in the success of scientific endeavors. Future work can build upon these insights, addressing questions related to Dask's overhead, bottlenecks in weak scaling experiments, and the comparative performance of NetCDF and Zarr in diverse HPC environments.

1 Introduction

In the realm of scientific research and data analysis, the efficient handling of complex and multi-dimensional data sets is a fundamental necessity. The choice of data format, storage methodology, and processing techniques is often a pivotal decision, as it significantly impacts the entire research workflow. However, this undertaking is not without its challenges. Data requirements can be highly specific, varying with the hardware infrastructure, implementation tools, and the nature of the scientific application. What works optimally for one dataset may not be suitable for another, making it crucial to tailor data management strategies to the unique demands of each project. Here, innovation can make a significant difference. Weather prediction is also continuing to evolve, becoming more accurate and offering insights into the future. Just as in scientific research, where novel data formats and storage solutions have become essential for handling ever-increasing volumes of data, the field of meteorology is making strides in improving its data management strategies. Today's standard for weather simulation, with a resolution scale of 10km, is already impressive, but there are ongoing efforts to reach the 1km scale, which promises even greater accuracy. The ECMWF[14] is at the forefront of medium-range weather prediction, introducing new methodologies and technologies to enhance forecasting capabilities. In scientific research and meteorology alike, the challenge extends to managing vast datasets efficiently. While in the past, achieving floating-point efficiency was a priority, the focus has now shifted to optimizing I/O operations and economizing data transfers. As we look toward the 1km scale in weather modeling, the data generated rapidly becomes immense, creating new challenges in data handling. For instance, CMIP5 simulations at 200km resolution already produce 2 petabytes of data[18], and transitioning to a 1km scale could quickly lead to the management of 80 exabytes of data.

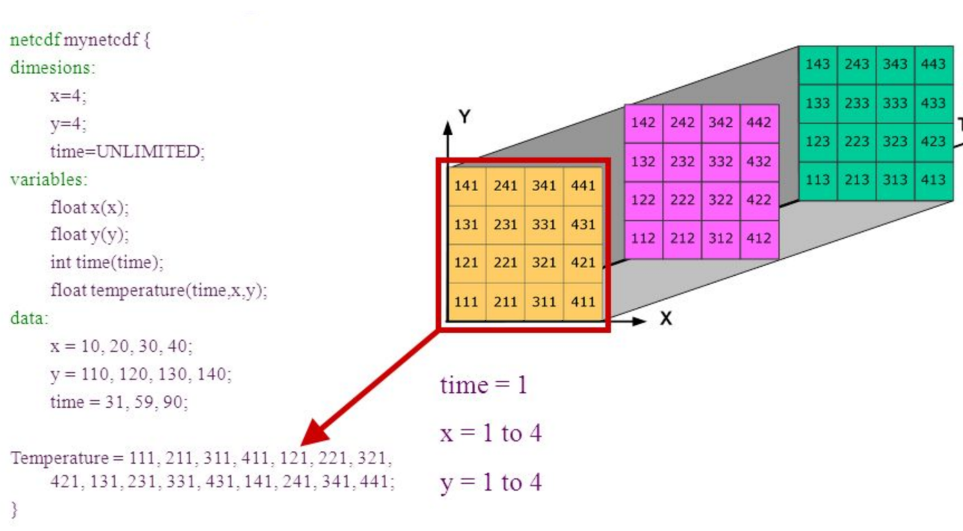
Just as High-Performance Computing (HPC) systems have revolutionized data handling in scientific research, these systems also play an indispensable role in meteorology. HPC clusters and supercomputers provide the computational power and resources needed to implement and innovate data management practices effectively, facilitating groundbreaking discoveries and insights. They expedite the processing of large-scale weather simulations, making real-time decisions and iterative experiments more achievable. In essence, HPC systems complement the importance of understanding the right data practices and formats, as they empower researchers in both fields to tackle data-intensive tasks, thus advancing our understanding of the world and the accuracy of our predictions.

1.1 Background

The choice of a scientific data format is crucial as it can significantly impact data management and accessibility in scientific research. Various formats, such as HDF5, NetCDF4, and Zarr, offer distinct advantages for different use cases. These formats are designed to store structured data efficiently and often include self-describing capabilities, making them versatile choices for scientific applications in various fields, from astronomy to medicine and physics. The ability to handle complex data structures, along with metadata, ensures that these formats cater to the diverse data storage and retrieval needs across the scientific community. In my experiment, the focus on Zarr and NetCDF4 as the two key data formats was guided by practical and technical considerations. One key reason was the acknowledgment that the NetCDF4 format's core is built upon HDF5. This inherent connection made NetCDF4 an attractive choice, benefiting from HDF5's strengths while providing additional specialized features for scientific data storage and access. Moreover, the dataset I had at my disposal was in the NetCDF4 format, streamlining data compatibility and integration. This allowed me to leverage the existing dataset structure and easily transition my research to explore the interaction of Dask and Zarr for benchmarking parallel execution performance.

1.1.1 NetCDF

The NetCDF (Network Common Data Form) data abstraction model offers a robust solution for managing scientific data, promoting accessibility, enhancing reusability, and transcending the confines of machine and implementation specifics.
NetCDF conceives scientific data sets as collections of named multidimensional variables, each enriched with coordinate systems and a set of named auxiliary properties. Each NetCDF file has three components: dimensions, variables, and attributes. Together they help capture the meaning of a scientific data set. Variables, the actual data, are assigned a specific data type, a shape, and possibly some attributes. These attributes provide critical context, making the data comprehensible and self-describing.



```
netcdf mynetcdf {
dimesions:
      x=4;
      y=4;
      time=UNLIMITED;
variables:
      float x(x);
      float y(y);
      int time(time);
      float temperature(time,x,y);
data:
      x = 10, 20, 30, 40;
      y = 110, 120, 130, 140;
      time = 31, 59, 90;

Temperature = 111, 211, 311, 411, 121, 221, 321,
      421, 131, 231, 331, 431, 141, 241, 341, 441;
}
```

Figure 1.1: Visual representation of a NetCDF file[2].

The NetCDF interface serves as a powerful and machine-independent data-access framework for programs engaged in the reading and writing of scientific data. Unlike traditional approaches reliant on machine-specific intricacies, NetCDF abstracts away these complexities, ensuring that data are accessed based on their inherent structure rather than the underlying hardware. NetCDF's design promotes efficient data access by allowing users to specify the precise section of data associated with a variable. This granular approach enhances the portability of data manipulation processes.
While NetCDF is a versatile tool for many scientific data applications, it does have certain limitations. It is best suited for applications dealing with homogeneous record structures, it primarily operates with array-based data and the generated files are not human-readable and require additional tools to permit access to the information.

1.1.2 Zarr

Zarr is an innovative and efficient data storage format designed to facilitate the management of large, multi-dimensional arrays. It was first introduced in the mid-2010s, with active development and refinement continuing in the subsequent years. It shares some goals with NetCDF, but shifts its focus in the domain of parallel and distributed computing environments, enhancing its usefulness for data analysis and processing at scale. Zarr's main advantage lies in its modern design and approach. It leverages the capabilities of contemporary data management tools and libraries, making it more suitable for today's data-intensive scientific applications and as a result it is increasingly gaining traction in the scientific community. Zarr is most exclusively implemented through python libraries, and integrates seamlessly with other widespread modules like NumPy, Dask, and Xarray, making it a popular choice for data storage and manipulation in Python-based scientific computing, although efforts were made to create bindings or wrappers to use Zarr in other programming languages. However, these may not provide the same level of functionality, performance, or convenience as using Zarr directly within Python.
In Zarr, data is natively organized in chunks, and this chunking behavior is a fundamental character-

istic of the format. Unlike some other data storage formats, where chunking is optional or specified on a per-variable basis, Zarr's design centers around the concept of chunks. In Zarr, all data is stored as a collection of fixed-size, multi-dimensional chunks.

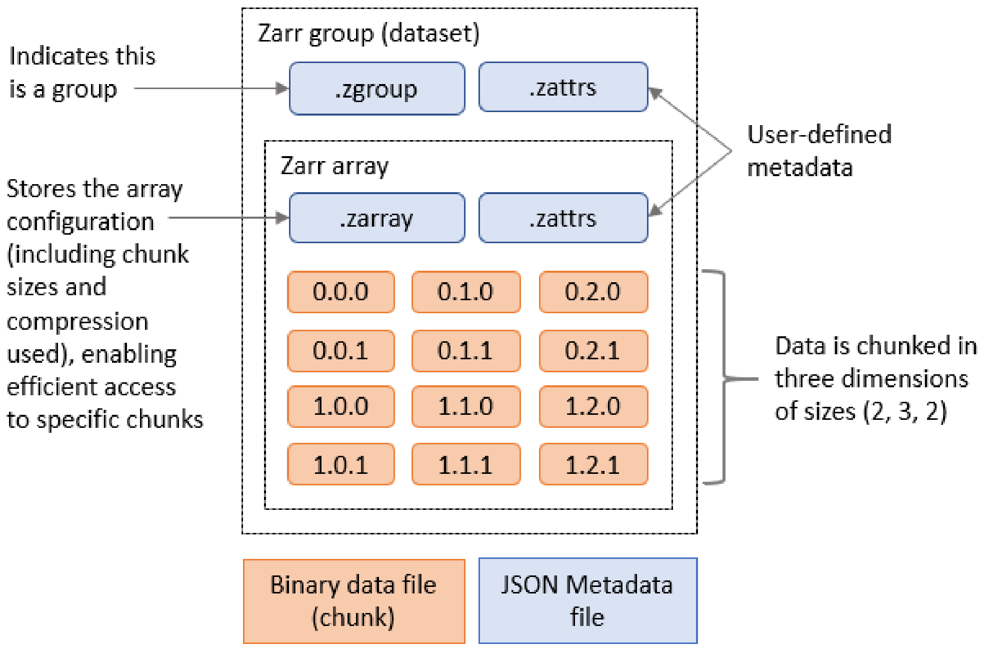

Zarr group (dataset)
Indicates this is a group
.zgroup
.zattrs
User-defined metadata
Zarr array
Stores the array configuration (including chunk sizes and compression used), enabling efficient access to specific chunks
.zarray
.zattrs
0.0.0
0.1.0
0.2.0
0.0.1
0.1.1
0.2.1
1.0.0
1.1.0
1.2.0
1.0.1
1.1.1
1.2.1
Data is chunked in three dimensions of sizes (2, 3, 2)
Binary data file (chunk)
JSON Metadata file

Figure 1.2: Visual representation of a zarr store structure[3].

1.1.3 Chunking

Chunking is a general concept in data storage formats, where large datasets are divided into smaller, fixed-size pieces called *chunks*. These chunks serve as the fundamental units of storage within the dataset. Chunking is a strategy that allows for the efficient organization and access of data in a way that optimizes the use of storage space and data retrieval without regard to specific format-specific details. Chunking was first introduced in NetCDF4, representing a significant evolution of the NetCDF data format. Unlike its predecessors, NetCDF4 incorporates chunking as a fundamental (but not mandatory) feature, enabling efficient storage and retrieval of large datasets. In NetCDF3, data was stored sequentially, which posed challenges when managing substantial volumes of information. The introduction of chunking in NetCDF4 revolutionized data organization by breaking it into smaller, fixed-size pieces. This architectural shift provided a host of advantages, including improved data access, compression, and support for parallel processing. Zarr data stores inherently implement chunking due to their structure and, since this feature is not optional anymore, efficient chunking techniques are essential to ensure optimal of performance.

1.2 Project Overview

Thanks to the Oak Ridge National Laboratory[7] I was given access access to both a powerful computing infrastructure and datasets to run the experiments. This work is part of a data hackathon[6] proposed by the laboratory to encourage scientific research on climate data at scale. Objective of the experiment was to evaluate both the performance of the storage systems used in the center and comparing different data formats. The scientific community has previously undertaken similar studies across a wide spectrum of heterogeneous systems. State-of-the-art literature and studies serve as invaluable resources, offering useful insights, methodologies, and benchmarks. The existing body of work serves as a complete reference for grasping the current state of the field. It is essential for new research efforts to draw from this valuable source of knowledge.

1.2.1 Pangeo benchmarking analysis

Pangeo represents a collaborative effort between scientists and software developers dedicated to facilitating interactive Big Data Geoscience analysis. This community-driven initiative aims to empower researchers to perform their analyses seamlessly, whether in the public cloud or on high-performance computing systems. Pangeo undertook a comprehensive study[19] with the objective of understanding how to choose between NetCDF and Zarr for scientific data management and analysis. The study delved deeply into the intricate nuances of these data formats and their compatibility with diverse storage infrastructures, aiming to scrutinize the strengths and limitations, particularly in the realm of I/O operations. By meticulously assessing the performance of both NetCDF and Zarr within these varying environments, the research seeks to provide invaluable insights into suggesting the most appropriate data format for different storage systems, thus equipping researchers with the knowledge to make informed decisions when managing and analyzing their data. They employed two main techniques called *strong* and *weak* scaling, in their weak scaling investigation, they assessed the read and write throughput while altering the number of nodes, all while maintaining a consistent dataset size per processor. They calculated the scaled read and write throughput by dividing the total dataset size by the overall runtime. In an ideal scenario, they would expect the throughput to exhibit a linear correlation with the number of cores, with a positive slope, indicating effective scaling.
On the other hand, in their strong scaling analysis, they concentrated solely on measuring read throughput, excluding any write operations. They varied the number of nodes while keeping the total dataset size constant. This approach allowed them to establish an upper limit on the scalability of object storage for the two distinct data formats by gauging the total read time for each configuration considered in the study.
The findings from the Pangeo research demonstrate notable variations in data read and write rates between NetCDF and Zarr on different storage systems. Weak scaling analysis indicates that Zarr exhibits significantly enhanced write rates on POSIX systems compared to NetCDF, attributed to its chunked data format and Dask support, which enhance parallelism and achieve optimal write speedup. However, with fewer processor counts, Zarr's read rates on POSIX systems are initially slower than those of NetCDF, but they converge as processor counts increase. Moreover, the read and write rates for Zarr on S3 object storage, while demonstrating scalability, tend to be slower compared to Zarr on POSIX systems, likely due to data transfer rate constraints. In contrast, the strong scaling analysis reveals that the read throughput for NetCDF and Zarr on S3 storage is comparable, offering geoscientists reassurance that transitioning to Zarr can maintain, or even enhance, read performance. This transition also brings several advantages, such as parallel output with compression options and flexible storage APIs, making Zarr a valuable asset for the geoscience community.

1.2.2 Comparison of HDF5, Zarr, and NetCDF4

In the paper[13], the researchers describe the development of a benchmark aimed at assessing the performance of three multipurpose scientific file formats: HDF5, NetCDF4, and Zarr. This benchmark was designed to evaluate the read and write speeds of these file formats by writing randomized data to specified datasets within a file and measuring the time taken to complete these operations. By doing so, the researchers aimed to provide an objective basis for comparing the performance of these file formats in various operations. The benchmark primarily focused on two critical types of operations: writing and reading. These operations are pivotal in testing a file format's capability as their efficiency directly impacts long-term data processing and end-user workflows. Faster write and read times not only reflect superior performance characteristics but also enhance user productivity by reducing time spent on non-essential operations. The research involved testing these three file formats on a computer running Ubuntu 18.04.5 with precise hardware specifications, including an Intel(R) Xeon(R) Silver 4215R CPU, 196 Gigabytes of RAM, and 960 Gigabytes of solid-state storage provided by a Micron 5200 Series SSD. Notably, this research was not conducted in a high-performance computing environment. The absence of an HPC setting is of particular interest, as it sets the stage for a comparison of the results obtained in this study with those from a fundamentally different and more complex environment. The outcomes of the benchmark analysis revealed distinct trends in the performance of the three file formats across different operations. Notably, when creating a dataset, NetCDF4 exhibited the shortest processing time, followed by HDF5, and Zarr came next. In terms of writing data to a file, HDF5 demonstrated the quickest write time, with Zarr following closely behind, while NetCDF4 required more than double the time on average compared to HDF5. The results of the read benchmark mirrored those of the write benchmark. NetCDF4 was the fastest in opening a dataset, followed by Zarr, and then HDF5. Furthermore, when reading the data and printing the dataset values to the standard output, HDF5 proved to be the fastest, followed by Zarr, and NetCDF4 exhibited a slightly slower performance. In summary, the research findings indicated that HDF5 excelled in reading and writing to a dataset, NetCDF4 was fastest in creating and opening a dataset, and Zarr closely trailed behind HDF5 in terms of performance.

1.2.3 Impact of Chunk Size on Read Performance of Zarr Data

In their paper[16], the research team specifically focused on the performance of various chunking strategies for multidimensional Earth science datasets, particularly emphasizing the use of Zarr as a storage format. The study was designed to resemble simplified use cases, providing insights into the impact of different strategies on processing time and memory usage when accessing data for common operations. The dataset examined in their research had dimensions of 5136 (time) x 1152 (longitude) x 721 (latitude) and was stored in an AWS S3 bucket. The study was conducted on a high-performance computing cluster on AWS, utilizing Python, Xarray, and matplotlib for analysis. To evaluate chunking strategies, the research team employed the Python package Rechunker, which facilitates efficient manipulation of Zarr datasets while preserving data integrity. They tested various chunk sizes for time, longitude, and latitude dimensions, covering a range of possibilities. The study focused on two common operations: extracting a time series at a coordinate and creating maps at specific datetimes. Performance was benchmarked for various chunking strategies, considering both metadata querying and data loading. The aim was to assess the I/O component of data access without parallel execution, using a single core. The research findings highlighted that the optimal chunking strategy varied depending on the access pattern. For extracting time series, larger time chunks and smaller spatial chunks performed best, while smaller time chunks and larger spatial chunks were ideal for creating maps. A strategy with intermediate-sized chunks that balanced both access patterns efficiently was identified. While wall time for responsiveness was a crucial factor, memory constraints also played a significant role in choosing an appropriate chunking strategy. The research emphasized the need for chunks that were large enough to minimize processing time yet small enough to fit into memory. This consideration was particularly relevant for Earth science datasets with multiple access patterns. The paper provided valuable insights into the impact of chunking strategies on data access performance and memory usage. The research findings can guide future studies and help establish efficient computing infrastructure for Earth science data in cloud-optimized formats.

2 HPC Cluster and software stack

2.1 High Performance Computing

High-Performance Computing represents the culmination of decades of technological evolution, with roots tracing back to the mid-20th century. It stands as a sophisticated solution for tackling intricate, data-intensive challenges that surpass the capacities of traditional computing systems. HPC environments, at their core, comprise intricate systems purposefully designed to unlock substantial computational power, predominantly realized through clusters of interconnected computers.

These clusters come together to create a distributed architecture that permits parallel processing, catering to the needs of high-throughput computing. Central to the functionality of HPC setups are the worker nodes, individual computing units that may be standalone computers or server nodes. Each worker node possesses its own dedicated central processing unit (CPU) and memory, often complemented by specialized hardware components, such as Graphics Processing Units (GPUs) or accelerators designed to perform specific types of computations efficiently. These worker nodes are interlinked, through high-speed interconnects, facilitating rapid data exchange and collaboration among the nodes. Integral to the efficient operation of HPC systems is the role of a job scheduler. The job scheduler meticulously manages the allocation of computational tasks across the worker nodes. It ensures the even distribution of workloads, thereby optimizing resource usage and minimizing idle time. Moreover, it takes into consideration task dependencies and priorities, ensuring a methodical and efficient execution of jobs. The underlying infrastructure of HPC environments includes a high-performance file system, often designed to be distributed and parallel. This file system plays a critical role in storing and granting access to the vast volumes of data generated and processed by HPC applications. Additionally, HPC setups demand robust cooling and power management systems, as the sheer computational power they harness generates substantial heat and consumes considerable amounts of energy. In the world of HPC, software plays an indispensable role. A comprehensive software stack includes the operating system tailored to the specific requirements of HPC, cluster management software for efficient resource utilization, compilers that enable the translation of code into machine-readable instructions, and libraries and tools optimized for parallel computing.

This software foundation empowers programmers to develop and run applications that can effectively leverage the parallel processing capabilities of the worker nodes. HPC's primary advantage lies in its unparalleled capability to process massive datasets and execute complex calculations at remarkable speeds. This attribute empowers scientists, engineers, and researchers to tackle a wide array of challenges, spanning weather modeling, drug discovery, climate simulations, nuclear research, and beyond. Cost-effectiveness often arises from the reduction in the time required for computations, which would otherwise consume substantial resources. However, HPC does bear some notable disadvantages. Foremost among them is the substantial cost associated with the acquisition, operation, and maintenance of HPC infrastructure, often rendering such systems inaccessible to smaller organizations. The complexity of programming for HPC systems is another hurdle, demanding specialized skills and expertise. Scaling HPC applications can be particularly challenging, frequently requiring fine-tuning of code and adaptation of algorithms to harness the full potential of the hardware. Additionally, energy consumption and the substantial cooling requirements of HPC data centers have raised environmental concerns, demanding innovative and sustainable solutions.

2.1.1 Infrastructure

A standard High-Performance Computing infrastructure is a complex and highly organized system designed to facilitate parallel processing and high-throughput computing. It typically consists of several key components, including login nodes, worker nodes, a high-performance file system, a network fabric, and a job scheduling system.

Login Nodes

The HPC infrastructure typically includes one or more login nodes. These nodes are the entry points for users and administrators to access the cluster. Users log in to the login nodes to submit jobs, manage their data, and interact with the HPC system. Login nodes are not designed for running computationally intensive tasks; instead, they serve as the interface to the cluster. This is why, when logging into an HPC system via Secure Shell (SSH), users generally provide a generic domain name, without the option to target a specific server. This procedure enables the HPC system to efficiently distribute and manage user access across multiple login nodes. As a result, the system's performance and responsiveness are safeguarded, ensuring that no single login node is overwhelmed with user requests, thereby enhancing the overall user experience.

Worker Nodes

The heart of the HPC cluster is comprised of worker nodes. These nodes are responsible for carrying out the actual computations, simulations, and data processing required by users and applications. Unlike login nodes, which serve as entry points for users, worker nodes are dedicated to performing the heavy lifting. Worker nodes in an HPC system are typically identical in terms of hardware configuration and highly specialized for their role. They are equipped with substantial processing power, memory, and often tailored hardware to handle the demanding workloads typical of HPC applications. The efficient coordination and utilization of worker nodes are central to the overall performance and productivity of an HPC system, ensuring that complex computations are executed rapidly and accurately.

File System

A high-performance file system assumes a pivotal role as an integral component of the infrastructure. Its primary function is to furnish users with a robust, high-speed storage solution, effectively serving as the repository for data and program files. What makes it particularly essential is its capability to provide shared storage that is not only rapid but also capable of accommodating the substantial volumes of data that HPC applications tend to generate and process. Furthermore, it's worth noting that this file system is often made accessible from both the login and worker nodes within the HPC environment. This unified access ensures that users, regardless of their position within the system, can seamlessly interact with the data they require, facilitating an integrated and cohesive user experience.

Network Fabric

High-Performance Computing clusters leverage a high-speed network fabric as a fundamental component to enable swift and efficient communication between the various nodes within the cluster. In this context, two prevalent choices for the network fabric are InfiniBand and high-speed Ethernet. This specialized network fabric forms the backbone of the HPC infrastructure, enabling worker nodes to seamlessly exchange data, collaborate on resource sharing, and coordinate the execution of parallel processing tasks. The choice between InfiniBand and high-speed Ethernet for an HPC network fabric depends on specific performance needs and cost considerations. Regardless of the choice, the network fabric's primary function is to facilitate efficient and reliable communication among worker nodes, contributing to the overall capability and effectiveness of the HPC cluster.

Job Scheduler

Job submission and management are crucial aspects of HPC. A job scheduling system plays a pivotal role in managing the allocation of computational tasks. It ensures an equitable distribution of workloads across worker nodes, optimizes resource utilization, and minimizes idle time. The job scheduler also takes into account dependencies and job priorities, ensuring that tasks are executed efficiently and in an organized manner.

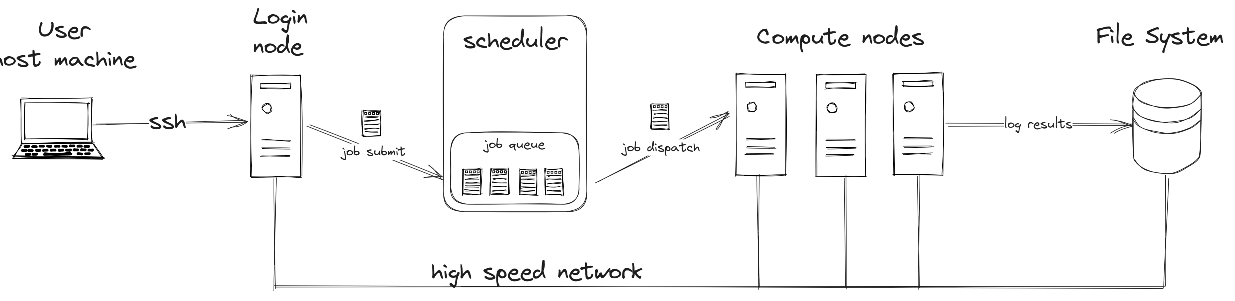


Figure 2.1: Schema of an HPC environment

2.2 HPC at Oak Ridge

The Oak Ridge Leadership Computing Facility aims to give researchers around the globe an opportunity to tackle problems that would be unthinkable on traditional systems. The laboratory hosts multiple supercomputers and storage systems able to serve the high throughput demands of thousands of concurrent jobs. During the thesis research I was given access to two HPC environments as a member of the XNR1K project.

2.2.1 Andes

Andes is a Linux cluster consisting of 704 compute nodes, primarily designed to facilitate extensive scientific research by handling the pre/post processing and analysis of simulation data generated on the Summit system. The 704 nodes are allocated to the batch partition, with each node being equipped with two 16-core 3.0 GHz AMD EPYC 7302 processors, incorporating AMD's Simultaneous Multithreading (SMT) Technology, and boasting 256GB of primary memory. Each CPU offers 16 physical cores, resulting in a collective total of 32 physical cores per node[1].

This machine was useful for getting accustomed to the HPC environment while providing a solid infrastructure on which to build the software pipeline. Access to the computing resources is granted through a SLURM batch scheduler accessed from one of the eight available login nodes.

SLURM scheduler

Simple Linux Utility for Resource Management (SLURM) is an open-source, fault-tolerant, and highly scalable cluster management and job scheduling system tailored for Linux clusters, capable of supporting thousands of nodes[9]. One of SLURM's notable strengths is its simplicity, making it accessible even to motivated end users who wish to explore its source code and potentially add their own functionality. The developers adhere to a philosophy of feature expansion only when those features hold broad appeal, ensuring the system remains efficient and uncluttered. Furthermore, SLURM is distributed freely under the GNU General Public License.Th scheduler is primarily written in the C language and employs the GNU autoconf configuration engine that helps creating portable and platform-independent source code[4]. Originally developed for Linux, it can be readily ported to other Unix-like operating systems. SLURM features a versatile "plugin" mechanism, enabling support for various infrastructures with ease. Designed for scalability, SLURM can effectively manage clusters comprising thousands of nodes. The SLURM controller for a 1000-node cluster consumes approximately 2 MB of memory and delivers excellent performance. Job resource requirements can be specified in various ways, including options and ranges. SLURM excels in handling diverse failure scenarios without disrupting workloads, including node controller crashes. User jobs can be configured to continue execution even if one or more nodes experience failures. The user command that governs job control can detach and reattach from parallel tasks at any time. Nodes allocated to a job become available for reuse once that job is completed. In the event of tardy node terminations due to hardware or software issues, only the scheduling of those nodes is impacted. While SLURM does not assume network security, it does presume that the entire cluster operates within a single administrative domain with a shared user base. Configuration in SLURM is straightforward and can be modified at any time without disrupting ongoing jobs. The system is proficient at managing heterogeneous nodes within a cluster, and its interfaces are script-friendly.

2.2.2 Summit

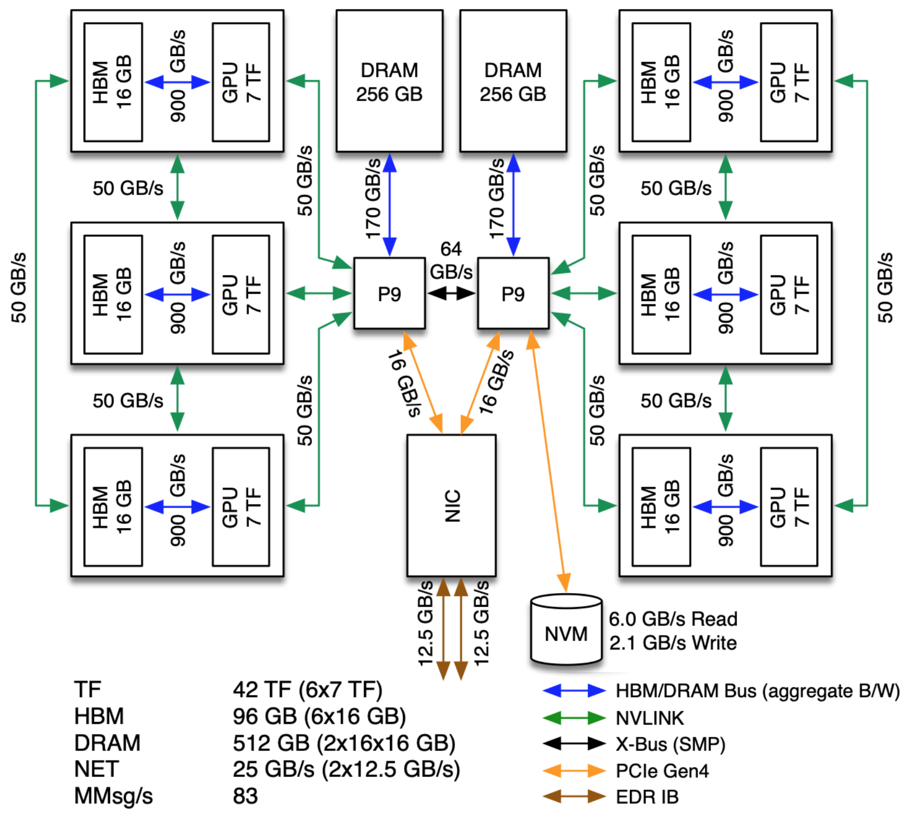
Summit, situated at the Oak Ridge Leadership Computing Facility, is an IBM system that ranks among the world's most powerful supercomputers[10], offering exceptional performance for a broad spectrum of traditional computational science applications. Its theoretical peak double-precision performance reaches approximately 200 petaflops. Remarkably, Summit is also renowned for its proficiency in deep learning applications, boasting mixed-precision capabilities exceeding 3 exaflops. The fundamental building block of Summit is the IBM Power System AC922 node. Each of these 4600 nodes incorporates two IBM POWER9 processors and six NVIDIA Tesla V100 accelerators, providing an impressive theoretical double-precision capacity of roughly 40 teraflops. Connecting the POWER9 processors are dual NVLINK bricks, each with a transfer rate of 25GB/s in both directions. In terms of memory, most Summit nodes are equipped with 512 GB of DDR4 memory for the POWER9 processors, along with 96 GB of High Bandwidth Memory (HBM2) for the accelerators. Additionally, they feature 1.6TB of non-volatile memory that can function as a burst buffer. Notably, a subset of nodes (54) are designated as "high memory" nodes, characterized by 2TB of DDR4 memory, 192GB of HBM2, and 6.4TB of non-volatile memory. The POWER9 processor is founded on IBM's SIMD Multi-Core (SMC) architecture, comprising 22 SMCs with individual 32kB L1 data and instruction caches. Pairs of SMCs share a 512kB L2 cache and a 10MB L3 cache, and SMCs support Simultaneous Multi-Threading (SMT) at up to level 4, which implies each physical core supports up to 4 Hardware Threads. To ensure efficient cooling, the POWER9 processors and V100 accelerators employ cold plate technology, while other components are cooled through more conventional means.
Summit comprises three primary node types: Login, Launch, and Compute, each serving distinct purposes while sharing similar hardware characteristics.

- **Login Nodes:** These nodes are the initial connection point when accessing Summit. They are designated for code development, editing, compilation, data management, and job submission. It's crucial to note that running parallel jobs or threaded jobs on login nodes is discouraged, as these are shared resources utilized by multiple users simultaneously.
- **Launch Nodes:** When a batch script or interactive batch job initiates, it runs on a Launch Node.. All commands within the job script or those executed in an interactive job are processed on launch nodes. Similar to login nodes, launch nodes are shared resources, so they are not suited for executing multiprocessor or threaded programs.
- **Compute Nodes:** The majority of nodes on Summit fall into the Compute category, and these are where your parallel jobs are executed. Access to compute nodes is granted through the "jsrun" command.

Despite the categorization into different node types, it's essential to understand that all nodes on Summit share similar hardware. This homogeneity eliminates the need for cross-compilation when building on a login node. Since login nodes possess hardware resources comparable to compute nodes, any tests conducted during the build process, particularly those managed by utilities like autoconf and cmake, will have access to the same hardware as the compute nodes. This results in a streamlined development process without requiring interventions typical of non-homogeneous systems.

IBM Spectrum LSF

IBM Spectrum Load Sharing Facility (LSF) software is a premier enterprise-class solution that serves as both a scheduler and a resource management framework. LSF excels in distributing workloads across a range of heterogeneous IT resources, thereby establishing a shared, scalable, and fault-tolerant infrastructure. This, in turn, results in improved performance and reliability of workloads while



TF	42 TF (6x7 TF)
HBM	96 GB (6x16 GB)
DRAM	512 GB (2x16x16 GB)
NET	25 GB/s (2x12.5 GB/s)
MMsg/s	83

Figure 2.2: Schema of the IBM Power System AC922 node[1]

concurrently reducing operational costs. As a scheduler, LSF's core capabilities lie in load balancing, resource allocation, and seamless resource accessibility. It identifies job requirements, assigns the most suitable resources, monitors progress, and ensures compliance with host load and site policies, thereby guaranteeing an optimal and efficient execution environment for workloads.

The **bsub** command allows users to submit jobs to the LSF scheduler, specifying job requirements, resource needs, and other parameters. It provides a straightforward way to initiate workload execution within the LSF-managed environment. On the other hand, the **bjobs** command enables users to track and manage their submitted jobs. It provides real-time information about job status, resource allocation, and execution progress, facilitating efficient job monitoring and management.

In summary, LSF is a versatile and sophisticated tool that combines scheduling and resource management to optimize resource utilization, enhance workload performance, and efficiently manage an organization's IT infrastructure.

Network fabric

Summit's nodes are connected to a dual-rail EDR InfiniBand network with a node injection bandwidth of 23 GB/s. These nodes are organized in a Non-blocking Fat Tree topology, a standard and widely adopted architecture in high-performance computing. The Fat Tree topology is favored for several reasons, primarily its ability to provide robust and efficient interconnection. By employing a three-level structure with switches at the first level to connect nodes within cabinets and Director switches at the second and third levels to link cabinets together, it ensures both redundancy and high network throughput. This architecture is recognized for its scalability and ability to handle the demands of modern supercomputing, making it a common choice for large-scale computational clusters. However, it can be complex to set up and maintain, incurring higher costs, especially at scale.

2.2.3 Storage Systems

Alpine

The OLCF employ a center-wide POSIX-based IBM Spectrum Scale parallel filesystem called Alpine. Alpine, boasting a colossal maximum storage capacity of 250 petabytes (PB), is a robust storage infrastructure constructed around 77 IBM Elastic Storage Server GL4 nodes, all operating IBM Spectrum Scale 5.x, collectively referred to as Network Shared Disk servers. Each individual IBM ESS GL4 node represents a scalable storage unit and is composed of two dual-socket IBM POWER9 storage servers. These servers are integrated with a high-speed 4X EDR InfiniBand network, delivering up to 100 Gbit/sec of networking bandwidth. The ultimate performance capabilities of the production system are remarkable, reaching around 2.5 terabytes per second for sequential I/O and 2.2 TB/s for random I/O in FPP mode, where each process manages its own file. Notably, metadata operations exhibit impressive results, with a minimum of approximately 50,000 file access operations per second and an aggregated capacity of up to 2.6 million accesses, particularly with small 32KB files. It's important to note that achieving optimal I/O performance is contingent on specific conditions. When saving a single shared file with a non-optimal I/O pattern, the performance can be lower than the maximum attainable. Furthermore, the system's performance results are typically achieved under ideal conditions, assuming dedicated system usage with a specific number of compute nodes. In a shared file system where multiple users engage in extensive I/O operations and large-scale job executions, performance may fluctuate due to the stress on the interconnection network. Similarly, if the I/O pattern isn't properly aligned, it can significantly hamper disk operation performance. For optimal performance on the IBM Spectrum Scale file system, it's recommended to utilize large page-aligned I/O and implement asynchronous reads and writes. The file system's block size is 16MB, and the minimum fragment size is 16KB. When storing files smaller than 16KB, they will still occupy 16KB on the disk. Writing files larger than 16MB leads to improved performance. Additionally, all files are distributed across Logical Unit Numbers and striped across multiple IO servers.
Alpine provides accessibility from both the login and compute nodes, simplifying direct input and output operations on the file system. Nevertheless, it's worth mentioning that performance may experience constraints dependent on the system state. The shared environment, with numerous users running resource-intensive tasks and potentially straining the interconnection network, can lead to fluctuations in I/O performance, impacting the consistency of file operations. Consequently, multiple runs are often necessary to accurately evaluate performance. In testing, we observed a high degree of variance between different executions, occasionally resulting in significant fluctuations of up to 50% in execution times.

Burst Buffers

Each compute node on Summit features a high-speed 1.6TB Non-Volatile Memory (NVMe) storage device built on the XFS file system, capable of achieving a theoretical peak performance of 2.1 GB/s for writing and an impressive 5.5 GB/s for reading. To enhance access to frequently used libraries, 100GB of each NVMe is allocated as an NFS cache. When calculating the maximum usable storage size, it's essential to account for this cache and formatting overhead. As a result, the usable portion consists of a maximum storage capacity of 1.4TB.
The NVMe drives can significantly reduce application wait times for I/O operations. These drives serve as burst buffers, allowing data to be swiftly transferred to or from the drive before applications read or write files. This integration enables applications to benefit from the native high-speed performance of SSDs for a portion of their I/O requests. Notably, this mode of usage supports writing file-per-process or file-per-node but does not facilitate automatic "n to 1" file writing, where multiple nodes write to a single file. After a job is completed, the NVMe drives undergo a trimming process, permanently erasing data from the devices. Consequently, the file system is not suitable for long-term data storage but is fully operational, allowing users to evaluate its performance characteristics.

2.3 Software components

Summit and Andes employ the Lmod module system, based on Lua, to effectively manage environment modules. Lmod facilitates adjustments to the shell's variables, enabling users to modify the software available in their shell environment without introducing conflicts between different packages or versions. Lmod is a recursive module system, which means it's designed to be aware of module compatibility and actively modifies the environment to avoid conflicts. In cases where Lmod makes implicit environment changes, it issues error messages to keep users informed. Environment modules are structured hierarchically by compiler family, ensuring that packages built with a specific compiler are accessible only if the corresponding compiler family is initially present in the environment. This structure simplifies the management of software components, ensuring compatibility within the Summit computing environment. The importance of Lmod lies in its capacity to dynamically load and unload modules, providing a versatile tool for creating custom environments. This dynamic approach empowers users to craft tailored environments that suit their specific needs, enabling them to select the necessary software packages and versions for their projects. Furthermore, Lmod aids in effective version management, allowing users to switch between different versions of software as required. This efficiency extends to resource management, as loading only essential modules optimizes resource utilization, and unloading them when they are no longer needed reduces memory and processing overhead. In shared high-performance computing environments, Lmod actively manages environment variables to prevent conflicts between software packages and versions, ensuring seamless compatibility. This flexibility is invaluable in collaborative research settings where multiple users with diverse requirements share computing resources, promoting productivity and resource-efficient computing practices.

2.3.1 Python

Thanks to Lmod, users have the capability to load the Python module, enabling them to harness its versatile functionality. Python is a widely used high-level programming language known for its simplicity and readability, making it an ideal choice for both beginners and experienced programmers. Python's appeal in the realm of scientific research is underscored by several key factors. One of the foremost reasons is its extensive ecosystem of libraries and frameworks tailored for scientific endeavors. These resources, including NumPy, SciPy, and Pandas, simplify tasks related to data analysis, modeling, and visualization, facilitating research across diverse scientific domains. Moreover, Python's open-source nature promotes collaboration and innovation within the scientific community, as researchers can freely access, modify, and distribute scientific tools. Python's versatility transcends disciplinary boundaries, making it accessible and applicable in fields as varied as physics, biology, data science, and engineering.

Miniconda

Miniconda plays a pivotal role in efficiently managing Python environments and their dependencies, making it an indispensable tool for developers, data scientists, and researchers. Setting up a Python environment with Miniconda is a straightforward process. By installing Miniconda, users gain the ability to create isolated, self-contained environments where they can tailor Python and its libraries to specific project requirements. This isolation ensures that different projects do not interfere with one another, avoiding version conflicts and maintaining a clean, reproducible development or research environment. Conda offers a vast repository of packages, enabling users to quickly access and integrate tools and libraries relevant to their projects. Furthermore, it excels in resolving complex dependency issues, simplifying the otherwise challenging task of managing package compatibility. Miniconda's lightweight nature ensures minimal system footprint, making it an excellent choice for creating streamlined and efficient Python environments tailored to specific needs.

In particular the Conda environment employed in the experiments comprises a selection of libraries and packages:

- *Channel*: channels are repositories or sources from which the Conda package manager retrieves software packages and libraries for installation. These channels serve as distribution points for

Conda packages and play a crucial role in managing software dependencies and environments.

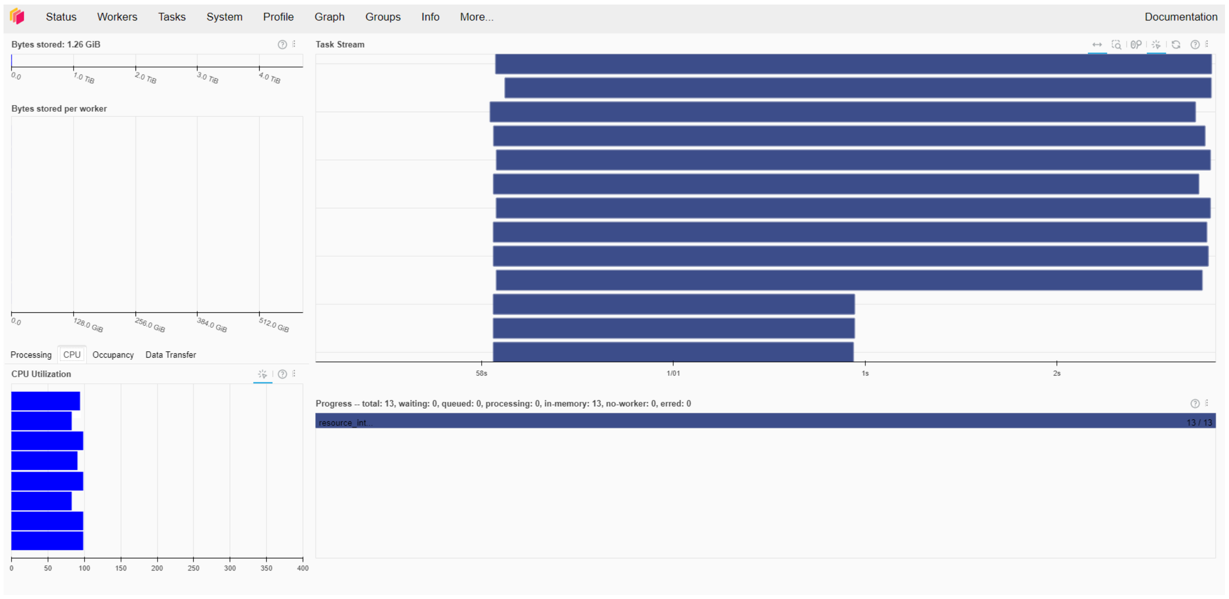
  - **conda-forge:** a community-driven Conda channel known for its extensive and diverse package collection, featuring packages from various domains, including data science and scientific computing. Maintained by a dedicated community of contributors, Conda-forge provides up-to-date and well-packaged software through a rigorous review process. It operates under an open and inclusive model, welcoming contributions from a diverse range of users, making it a valuable resource for those seeking reliable and comprehensive packages for their projects.

- *Dependencies*
  - **Python**=3.10.12: Python is a versatile high-level programming language widely utilized in various domains, including web development, scientific computing, data analysis, and artificial intelligence.
  - **Xarray**=2023.8.0: Xarray is a Python library tailored for handling labeled, multi dimensional arrays. Its primary purpose lies in simplifying data analysis tasks related to climate science, meteorology, and geophysics. It streamlines the process of working with complex, labeled data.
  - **NumPy**=1.26.0: NumPy is a fundamental Python library that empowers numerical computing. It provides efficient support for large multi-dimensional arrays and matrices, along with an extensive collection of mathematical functions. These capabilities are invaluable for scientific and data-intensive applications.[1]
  - **Pandas**=2.0.3: Pandas is a powerful Python library designed for data manipulation and analysis. It introduces essential data structures, such as DataFrames and Series, simplifying the handling and exploration of tabular data. Researchers and analysts benefit from its versatility.[1]
  - **NetCDF4**=1.6.4: NetCDF4 is a Python library crucial for working with the NetCDF data format, widely adopted in scientific computing, climate modeling, and geospatial data storage. It provides the necessary tools for efficient data handling in these domains.[1]
  - **Bottleneck**=1.3.7: Bottleneck is a Python package specialized in optimizing specific operations on NumPy arrays. Its primary goal is to accelerate array-based computations, particularly when dealing with large datasets. Researchers benefit from enhanced computational efficiency.[1]
  - **Zarr**=2.16.1: Zarr is a Python library that addresses the storage and retrieval of chunked, compressed, and multi-dimensional arrays. Its purpose is to facilitate efficient data storage and retrieval in scientific and data-intensive applications, improving data access speed and resource utilization.[1]
  - **Dask-Jobqueue**=0.8.2: Dask-Jobqueue is a Python package designed to seamlessly integrate the Dask library with job queuing systems like SLURM and LSF. It enables distributed computing across clusters of nodes, making it an asset for parallel data processing and computational workloads.
  - **Dask**=2023.6.0: Dask is a flexible Python library focusing on parallel and distributed computing. It allows users to construct custom high-level task scheduling and data processing workflows, ensuring efficient utilization of computational resources.
  - **Distributed**=2023.6.0: Distributed is utilized for distributed computing and parallel processing, enhancing the overall performance of computational tasks.
  - **IPykernel**=6.14.0: IPykernel is a Python package essential for Jupyter notebooks. It enables seamless interaction with different Python versions within Jupyter environments, facilitating the execution of Python code in various notebook setups. This library is not necessary, but can help during the debugging phases to find problems that may become cumbersome to discover during a bulk job execution.

[1]The library is either mandatory or suggested for a seamless and optimal operation of Xarray.[11]

2.3.2 Dask

Dask is an open-source, flexible, and powerful parallel computing framework in Python that allows you to efficiently perform parallel and distributed computing. It is designed to scale from single machines to large clusters, and it is particularly well-suited for handling tasks that involve large datasets or complex computations.Dask was created to address the limitations of Python's Global Interpreter Lock (GIL)[8], which can hinder multi-threaded parallelism. It provides a solution for efficiently utilizing multiple cores and distributed computing resources. It can be used on a single machine or distributed across a cluster of machines, making it highly versatile. It is designed for tasks that exceed the available memory on a single machine or require more computational power than a single CPU can provide. Dask introduces several core data structures, known as "Dask collections", which include Dask Arrays, Dask DataFrames, and Dask Bags. These collections extend popular libraries like NumPy and Pandas to work on larger-than-memory or distributed datasets. Dask's task scheduling is a fundamental aspect of the Dask framework, essential for enabling parallel and distributed computation. It revolves around a directed acyclic graph of tasks, where each task represents a unit of work and dependencies between tasks are tracked. Dask follows a lazy evaluation approach, meaning it defers task execution until the results are explicitly requested, allowing for optimization and efficient scheduling. A task scheduler, inherent to Dask, is responsible for determining when and how tasks in the graph are executed, taking into account resource management, such as CPU cores and memory allocation. The scheduler can apply various optimizations to improve efficiency, such as minimizing data movement and task execution. Task priority assignment helps determine the order in which tasks are executed, and in a distributed environment, Dask's distributed task scheduler facilitates large-scale parallel and distributed computing across multiple machines.



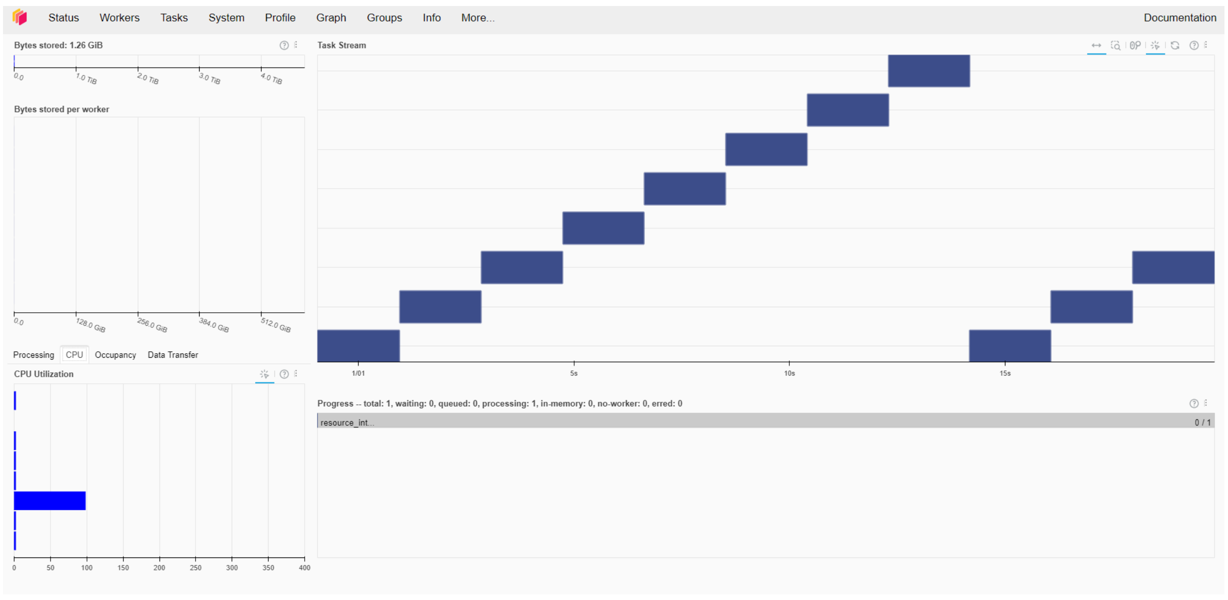
Progress -- total: 13, waiting: 0, queued: 0, processing: 0, in-memory: 13, no-worker: 0, erred: 0

Figure 2.3: Dashboard view when executing a parallel resource-intensive task, the execution is automatically distributed to workers in a round-robin fashion

This scheduler incorporates fault tolerance mechanisms to handle worker failures and includes a web-based dashboard for real-time monitoring of task progress, resource utilization, and performance diagnostics. The Dask dashboard is a web-based interface that's typically available at a specific URL, the default one often being **http://localhost:8787**[1].The Dask dashboard interacts with the Dask scheduler to gather information about the cluster's status and worker utilization. Internally, Dask uses communication protocols such as TCP/IP, HTTP(S), and WebSocket to facilitate these interactions. The Dask distributed cluster web page provides comprehensive information about the cluster's status, task execution, and resource allocation. The *STATUS* page offers insights into task execution

[1]Note that in a distributed environment it is accessible in this only from within the machine running the scheduler. From other machines on the same network users need to use the correct IP address of the server running the scheduler.

and distribution, offering a concise view of resource allocation for each node and the total memory allocated for running tasks. The *GRAPH* page visually represents computations through nodes (representing tasks) and edges (denoting task dependencies), showcasing the flow of data in Dask computations. Dask's lazy evaluation constructs the graph when computations are defined but defers execution until explicitly requested, enabling optimization. Parallelism is a central feature, as tasks execute independently when unburdened by dependencies, making efficient resource utilization while respecting dataflow dependencies. The Dask Task Graph is invaluable for understanding, monitoring, and optimizing Dask computations, especially in complex workflows and distributed computing scenarios. Additionally, the *WORKERS* page lists all connected workers, providing resource utilization details and easy access to further information with a simple click on a worker. These represent just a selection of the insights provided by the dashboard, which offers over 30 distinct views for in-depth cluster monitoring and analysis[2]. Dask provides a high degree of flexibility and customization when it comes to task execution, but with that power comes a responsibility to design and manage the workflow effectively. Design errors in the executed scripts can have significant impacts on execution, performance, and even resource utilization as demonstrated by Figure 2.4.
For instance, let's consider an example where a Dask script is designed to execute a task that requires a substantial amount of memory. If the script doesn't manage memory efficiently or has an error in data partitioning, it can lead to excessive memory usage and potentially crash the system. Additionally, poor task scheduling can lead to inefficient CPU utilization, slowing down the entire computation.



Progress -- total: 1, waiting: 0, queued: 0, processing: 1, in-memory: 0, no-worker: 0, erred: 0

Figure 2.4: Dashboard view when running a script similar to the one run in Figure 2.3, but with an intentional design flaw that serialize execution rather than parallelizing it. Dask still distribute tasks, but their execution is deferred

It's important for programmers working with Dask to have a solid understanding of Dask's capabilities and best practices. This includes proper data chunking, careful management of dependencies, efficient use of resources, and the judicious selection of Dask's various scheduling strategies. Dask offers numerous configuration options to fine-tune its behavior, but improper use can lead to unintended consequences.

[2]More information can be found at https://docs.dask.org/en/latest/dashboard.html

3 Benchmarks

3.1 Workflow

The experiment workflow is a well-defined process that initiates from an LSF script, which is submitted from a login node, thereby specifying the resource allocation for the experiment run. It defines parameters such as walltime, the number of nodes, and the quantity of workers needed for the task. This LSF script serves as the foundation for the entire workflow, orchestrating the creation of temporary folders, handling file transfers when required, and launching essential components, including a Dask scheduler instance and the requested number of workers. Subsequently, the Python script responsible for the actual computation takes center stage.
The Python script is designed to accept four crucial command-line parameters. First and foremost, it requires the location of the Dask scheduler file, which is generated at the launch of the Dask scheduler itself. This file provides crucial information regarding the scheduler, notably its networking parameters for enabling interaction. Secondly, the script necessitates the path to the input NetCDF file, specifying the dataset to be processed. It also mandates a destination folder for the output storage, which is where the results of the computation will be stored in Zarr format. Lastly, the Python script expects to be provided with the number of workers, which is pivotal in ensuring that all workers are connected and ready before initiating the computation.Once the Python script is launched, it commences by establishing a connection to the Dask cluster and diligently waits for all workers to connect, ensuring a synchronized start of the computation.

The Dask documentation outlines two primary methods for parallelizing input/output operations. The first approach involves passing the job to Dask, simplifying the workflow by leveraging Dask's capabilities, particularly when the dataset is already in memory. In this scenario, Dask can subdivide the execution and load the dataset into memory, making it a straightforward choice for the task. Conversely, the second approach requires the programmer to explicitly partition the input data into smaller regions, with each region assigned to an individual worker responsible for processing its allocated portion of data. While this method provides fine-grained control to the programmer, it may not always yield the most efficient results.

Upon the successful completion of the computation, the Python script carries out the important task of closing all workers and, in the final step, gracefully concludes by shutting down the scheduler, ensuring a well-organized and efficient conclusion of the experiment workflow.

3.2 Dynamic testing submission

Conducting comprehensive benchmarks in an HPC (High-Performance Computing) environment, with a specific focus on key parameters such as file system type, number of nodes/allocation strategy, number of Dask workers, and file dimensions, holds great importance in advancing our understanding of HPC systems and optimizing their utilization in scientific research and computational tasks. These benchmarks play a crucial role in advancing our understanding of HPC systems and optimizing their utilization in scientific research and computational tasks. Here, we delve into the in-depth importance of each of these parameters and why they are essential in HPC research:

Burst Buffers vs POSIX file system

The choice of file system type plays a pivotal role in determining the efficiency of data storage and retrieval within HPC systems. It has a direct impact on how data is accessed, manipulated, and utilized during computational tasks. Comparing Burst Buffers to POSIX file systems offers valuable insights into data management, which is a cornerstone of HPC applications. Burst Buffers, as a specialized type of file system, are designed to provide high-speed, temporary storage solutions. These buffers are essential for enhancing I/O (Input/Output) operations, particularly in scenarios where scientific simulations involve massive datasets. Their primary advantage lies in their ability to minimize I/O latency, making them indispensable for real-time or near-real-time simulations where data access speed is paramount. On the other hand, POSIX file systems are more traditional and provide persistent storage. They are well-suited for long-term data preservation and general-purpose file management.

Nodes allocation strategy

The number of nodes and allocation strategy profoundly impacts the scalability and resource utilization of HPC applications. Evaluating the effect of varying the number of nodes and employing different allocation strategies provides valuable insights into the optimal use of computational resources. Efficient resource utilization ensures that HPC resources are used effectively, which is especially important for minimizing operational costs. Figure 3.1 displays various allocation strategies for distributing processes in an HPC environment. Specifically, we focused on two configurations available in Summit. In these configurations, the SLURM scheduler was directed to consistently allocate entire nodes when requested via a script. This ensured that each process had exclusive access to all the resources provided by the node, without any competition from other processes. However, for the specific experiments I conducted, I did not observe a significant difference in execution times between the two allocation strategies.

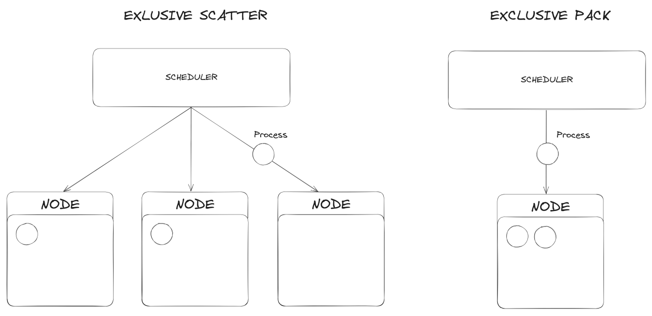


Figure 3.1: Schema of a scheduler distributing processes among nodes, with the scatter strategies processes are distributed one per node. Pack involves assigning all the processes to the same node. The "exclusive" keyword means that nodes are entirely allocated for this job, and no other jobs can interfere

Dask workers

Dask workers are instrumental in parallel processing and distributed computing in HPC environments. Varying the number of Dask workers allows researchers to explore how parallelism affects computational performance and scalability. In-depth analysis of Dask workers' role in workload distribution, load balancing, and system efficiency is indispensable. It helps in configuring HPC environments optimally for diverse scientific applications, ensuring that the system can handle increasingly complex workloads.

File dimension

File dimensions have a significant impact on data size, I/O operations, and computational performance. By experimenting with different file dimensions, researchers can evaluate the scalability of applications with respect to data size. This is particularly crucial when dealing with scientific simulations that generate large datasets. Understanding how data size affects processing times, I/O efficiency, and storage requirements is fundamental for optimizing computational tasks and resource management.

LSF script generator

A single bash script manages both the creation and execution of LSF scripts. This script takes as input a sequence of worker numbers (provided as a bash array like (1 2 4 8 16 32)). It initiates the process by generating a temporary script folder and an output folder, and then proceeds to iterate through the worker numbers. For each worker count, it calculates the required nodes for allocation, generates an LSF script with the appropriate scheduler parameters and execution commands, saving it in the temporary folder. Subsequently, it makes the script executable and submits it to the scheduler. In consecutive iterations, the bsub command for submission includes an additional flag, **-w 'ended($previous_job_id)'**, where **$previous_job_id** represents the scheduler-assigned ID of the previously submitted job. This flag is crucial for resource management and ensuring test serialization, allowing job execution only after the prior one has completed, regardless of its completion state. This configuration ensures that output files are correctly stored inside the appropriate folders.

3.3 Dataset

Oak Ridge National Laboratory granted me access to a valuable climate dataset, sourced from another project hosted on their robust system. This dataset comprises multiple resolutions of the ERA5 model, specifically organized into four distinct datasets with resolutions of 0.25, 1.4, 2.8, and 5.6 degrees. Within each of these datasets, data is stored in NetCDF format, meticulously organized by year, spanning from 1941 to 2022, and further categorized by variable, including but not limited to humidity, wind velocity, and temperature which example is in appendix A. The datasets presented here originate from a previous experiment[17] designed to create benchmark datasets for data-driven medium-range weather forecasting. In such endeavors, having robust benchmarks with well-established baselines is essential to drive swift advancements in problem-solving. In this context, a specific benchmark has been formulated and the provided datasets are pre-processed to facilitate their utilization in machine learning models.

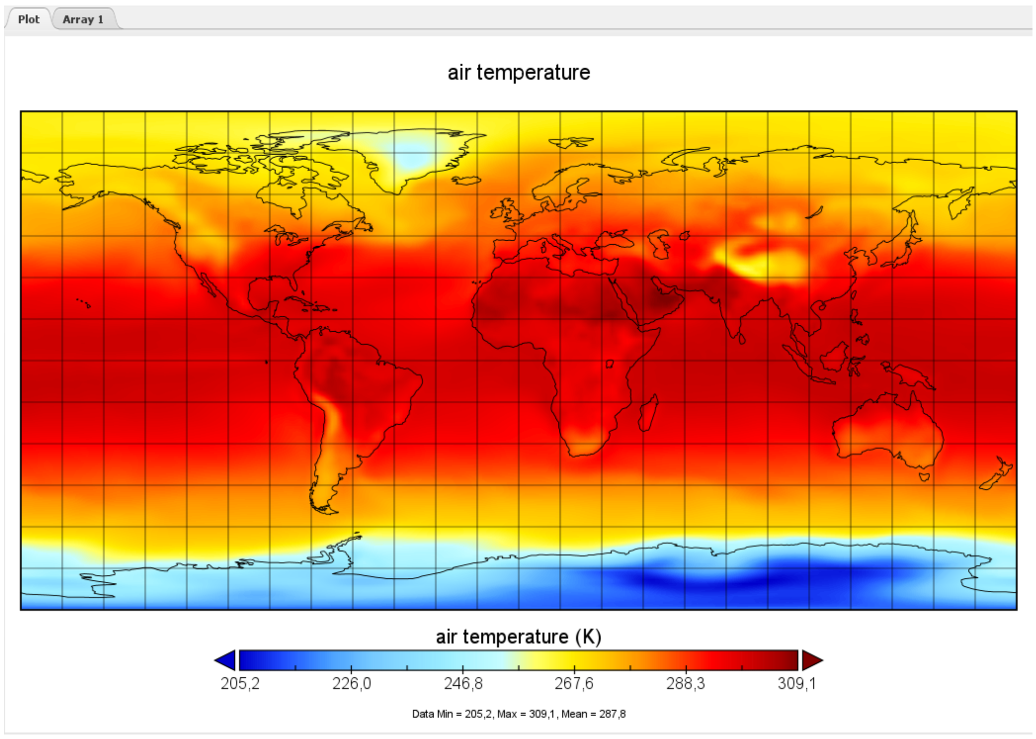


Figure 3.2: Visual representation of temperature values extracted from a NetCDF file using Panoply[5]

For the scope of my experiment, I focused solely on the temperature variable. However, it's important to note that due to the well-structured nature of the dataset, the same experimental procedures and results are potentially extendable to the other variables within the dataset. Each NetCDF file within these datasets contains data points, with the number of points varying based on the resolution of the dataset. These datasets capture data at hourly intervals, a detail that results in a rapid increase in the dataset's dimensions, particularly as the resolution becomes finer. This rich and comprehensive dataset offers a remarkable opportunity to delve into climate data analysis, shedding light on critical insights and trends across various temporal and spatial scales.

3.4 Benchmarking Zarr

In this section, I will showcase the results derived from the application of a variety of strategy combinations during the benchmarking process. It's essential to note that for a precise and comprehensive understanding, conducting tens of experiments would be necessary. However, the results presented here are valuable in providing a general overview and insights into the impact of these strategies on the performance and efficiency of HPC systems.

Let's begin by addressing the essential terminology of *speed-up* and *efficiency* in the context of High-Performance Computing.
In the field of HPC, *speed-up* refers to the improvement in the execution time of a parallel program compared to its sequential counterpart. It quantifies how much faster a parallel application runs when more resources, such as processors or nodes, are allocated.

$$Speedup = \frac{T_s}{T_p}$$

On the other hand, *efficiency* in HPC indicates how effectively the available resources are utilized to solve a problem. It measures the ratio of actual speed-up achieved to the maximum potential speed-up given the number of resources employed. In other words it can be used to test whether speedup increases linearly with respect to the number of workers employed.

$$Efficiency = \frac{\frac{T_s}{T_p}}{p} = \frac{T_s}{p{*}T_p}$$

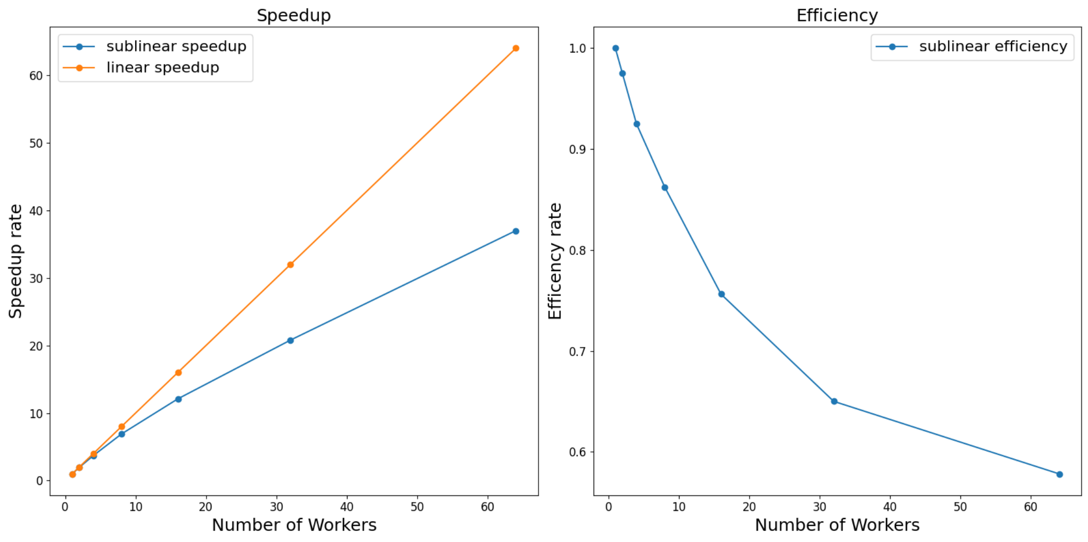


Figure 3.3: The plot illustrates the theoretical speedup achievable in parallel computations, typically expected to follow a linear trend. However, in practical scenarios, factors like overheads and specific implementations often result in a sublinear trend in real-world experiments.

For a thorough examination of parallel computation, it is crucial to acknowledge the significance of sequential execution. Therefore, I initiate the analysis by presenting the time required for a conventional sequential program, employing a solitary worker, to create the Zarr dataset. As expected, we observe a consistent pattern where the time increases in correlation with the file dimension. These initial time measurements establish the foundational benchmark for assessing the degree to which parallelization can enhance the overall efficiency of our computational processes.

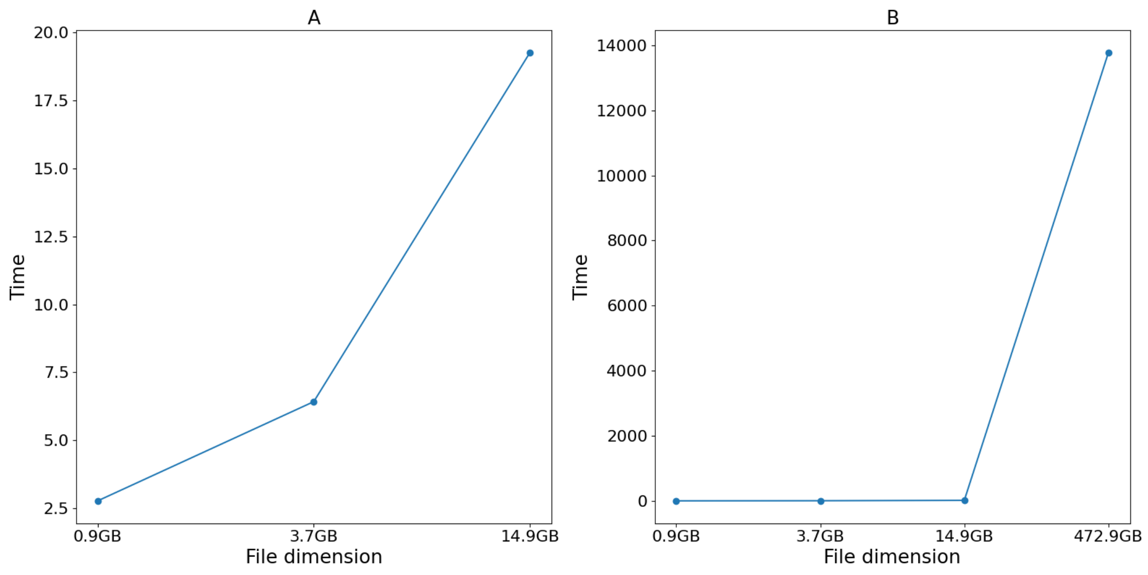


Figure 3.4: Execution time for the creation of Zarr store using 1 worker. Figure A presents a zoomed-in view of the time needed for converting the file with a single process, focusing on smaller file dimensions (0.9GB to 14.9GB). Figure B provides the complete view of the same data, including the larger file dimension (472.9GB). This dual representation offers a more detailed examination of conversion times for varying file sizes.

The initial test aimed to explore the two primary options for parallel output, as outlined in the xarray documentation. The first approach involved the creation of a Python script capable of partitioning the NetCDF dataset, distributing each region to a separate worker, and tasking them with writing the Zarr store. This method, although providing control to the programmer, placed the responsibility of ensuring data consistency on their shoulders.
To address the issue of consistency, two methods were considered:

- **Process Synchronizer:** This approach involved utilizing a process synchronizer. Zarr offers support for process synchronization via file locking, assuming that all processes have access to a shared file system and that the underlying file system supports file locking. However, experimentation revealed that this solution resulted in a low degree of parallelization, with storage access being significantly constrained.

- **Dataset Subdivision:** The second approach involved a careful partitioning of the dataset into distinct regions, with a focus on ensuring that the boundaries of data chunks aligned precisely with these subdivisions. This method require the creation of a template store, the template contains metadata regarding the variables and no actual data. It must be created employing dedicated xarray API and this contribute to the execution time with a variable overhead (3-25%). While this approach proved to be challenging yet achievable, it was ultimately surpassed in performance by the native Dask approach. When a dataset is loaded into memory, Xarray can utilize Dask utilities to optimize parallelization[12]. Handling datasets becomes more complex when dealing with large files because of memory constraints. In such cases, data needs to be explicitly loaded[1] and scattered among Dask workers. Once this data distribution is in place, Xarray is prepared to segment and parallelize tasks by invoking Dask APIs. In the experiment, it's important to note that loading and data distribution times are not taken into consideration. The focus is solely on tracking write operations that occur after the data is already in memory.

After establishing the superior parallelization approach, I proceeded to examine various file system configurations. In-depth testing was conducted on three distinct configurations: standard POSIX file system access, POSIX file system access with the maximisegpfs flag, and the utilization of burst

[1]https://docs.xarray.dev/en/stable/generated/xarray.Dataset.persist.html

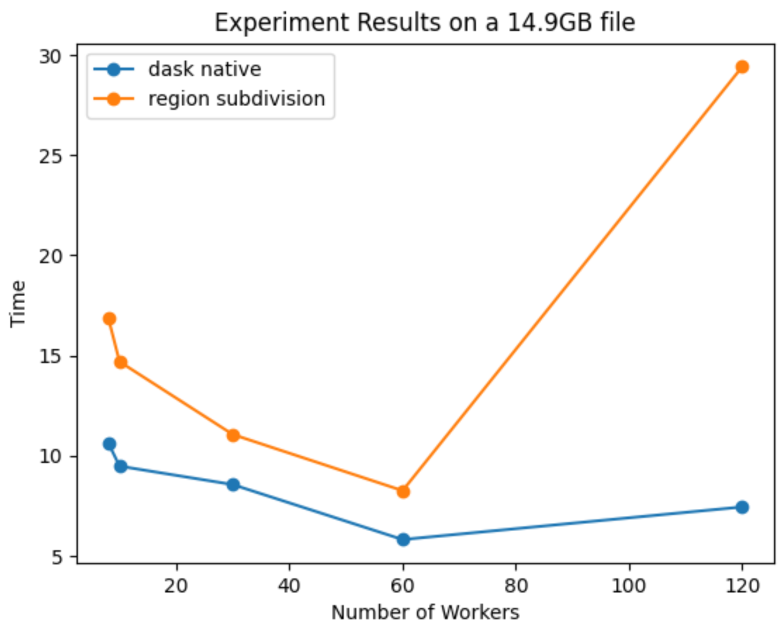


Figure 3.5: The Figure shows execution times obtained by applying two different distribution techniques for NetCDF to Zarr conversion tasks.

buffers. The adoption of the maximisegpfs flag was a direct outcome of a prior system study[15], which unveiled the potential for enhanced performance when this flag is utilized, particularly in scenarios where applications are constrained by I/O operations. This comprehensive analysis was aimed at gaining valuable insights into their varying performance characteristics. After conducting multiple runs, the results consistently demonstrate that burst buffers offer significantly improved performance. However, it's worth noting that they come with certain limitations, particularly in terms of the number of workers that can be effectively employed. Each Summit compute node is equipped with 42 physical CPU cores. Deploying more Dask workers on a node than there are physical CPU cores does not align with efficient resource utilization. This mismatch in worker-to-core ratio can result in resource contention, where the workers compete for the limited CPU and memory resources. This limitation is especially relevant when working with burst buffers, which are storage systems accessible exclusively within the node environment. In such cases, maintaining a worker count in line with the number of physical cores is the most effective approach for maximizing the utilization of the available resources. To illustrate performance comparison, a sample run is depicted in Figure 3.6.

Once I determined the optimal parameters for testing performance, I initiated the generation of LSF scripts to execute a serialized sequence of tests. The only parameter I needed to select was the array of workers to provide as input to the script. I opted for the values (1, 2, 4, 8, 15, 30, 40). Notably, these values deviate from the conventional approach often used in similar cases, where worker counts typically increase exponentially with a base of 2 (1, 2, 4, 8, 16, and so on). The reason behind this choice was to standardize dataset chunking. My employed chunking technique for efficient data storage exclusively involved subdividing the dataset along the time dimension, which has 8760 data points, and following the conventional approach would have resulted in an uneven distribution for certain standard worker counts. The process is straightforward, after loading the netcdf dataset, the xarray API for writing zarr will split the task among the workers connected to the dask scheduler, maximizing parallelization. At this stage, it is crucial to monitor the execution using the Dask dashboard, as it provides valuable insights into the run, including the task allocation among workers and their respective CPU usage. These tools are essential for monitoring that parallelization is truly effectively and efficiently managed. The Dask dashboard is accessible via HTTP connections, and the web server for the dashboard is launched at the Dask scheduler's startup, exposing several endpoints for users to monitor the execution's state. However, since the scheduler doesn't run on the

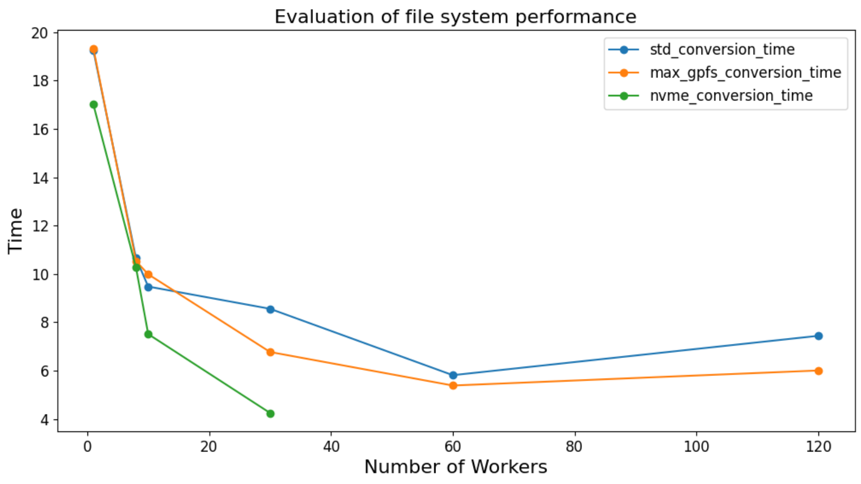


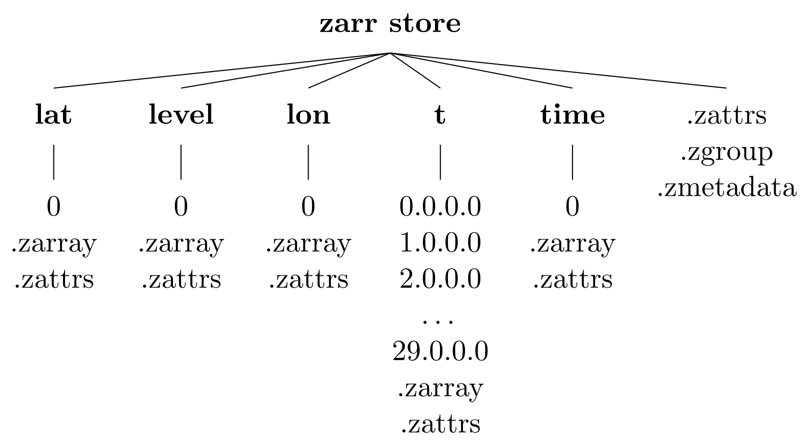
Figure 3.6: The figure illustrates a single execution of the program utilized for assessing File System performance. Notably, in the case of burst buffer (nvme) executions, the worker count is capped at 30, as exceeding this limit is not feasible due to operational constraints.

login node, it's necessary to utilize port tunneling to access the web server. The correct approach involves a double tunneling process. First, a tunnel is established during the connection from the user's host to the HPC login node, redirecting requests from a chosen port on the host PC to one on the login node. After launching the LSF script, we wait for the Dask cluster to initiate, retrieve its connection information, and then set up another port forward from the login node to the node hosting the Dask scheduler. This forwarding is done to the port specified in the Dask scheduler configuration file, enabling access to the Dask dashboard. Given that the login and compute nodes are on the same network, a simpler but less structured approach is to launch the scheduler, manually retrieve the connection details from the file, and then establish a new SSH connection from the host PC to the login node. During this connection setup, you can specify that requests to a particular port on the host machine should be redirected directly to the <IP, port> of the web server. This streamlined method allows for direct access to the Dask dashboard, given the network proximity, without the need for complex tunneling configurations. The first approach simplifies the process of tracking multiple runs, as the forwarding is set up from scratch each time. However, it's important to consider that this level of tracking may not be necessary, as once we've confirmed that the script works initially, we can reasonably assume that it will continue to function without the need for detailed tracking for every subsequent run. The result of a run consists of three main components:

- **Execution standard Error:** This file will contain the any error messages generated by the job during its execution. It's a way to capture and record information about any issues that may have occurred during the job's runtime, allowing for later analysis and troubleshooting. It serves the function of the stderr stream in a classic execution.

- **Execution standard Output:** Similarly to the previous one it is produced and managed by the job scheduler. It collects content written to the standard output stream during job execution. Output may be buffered and delayed, especially in the case of Dask clusters, where worker-generated output may only appear at the end of the computation.

- **Zarr Storage:** It represent the dataset that has been converted from NetCDF to the Zarr format. In Zarr, the hierarchical structure features individual folders at the top level (one for each dataset variable or dimension), accompanied by files that store overarching metadata related to the dataset. This metadata includes attributes from the original NetCDF file, as well as additional properties required for the Zarr format. The **.zmetadata** file is particularly important, as it serves as a collection of all the attributes for the storage. This file is formatted in JSON and includes information about chunking, compression type, data types, dimensions, and default values for each variable. Within each variable's subfolder, you will find a metadata file specific to that variable, as well as a file containing the actual data. For variables that have been chunked, there is a separate file for each chunk, and the file names are used to identify the chunk's ID. Data chunking is essential to Dask parallelization, this is reflected in how tasks are distributed to workers, in particular every chunk of a dataset is handled by one worker. A natural consequence of this behavior is dividing the dataset (along the time dimension) by the number of worker, and pass each portion to one of them. This approach works well until the region dimension is less than 2147483647 bytes[2], this represent the maximum buffer dimension and raises an error. To cope with the problem some configurations may need a number of chunks larger than the number of workers. This have an impact on how data is processed by the system, but shouldn't worsen performance. This chunking strategy is a fundamental part of how Zarr organizes and stores data efficiently.



To summarize, each experiment involves the conversion of a particular NetCDF file at a specified resolution, employing varying numbers of workers. Following the completion of the last job, a script performs result analysis tasks, which include verifying the proper construction of the Zarr storage and ensuring data consistency with the original NetCDF file. Additionally, it scans for errors within the stderr files and extracts execution times from the output files. These times are subsequently used to compute speedup and efficiency metrics, which are then visualized through plotting.

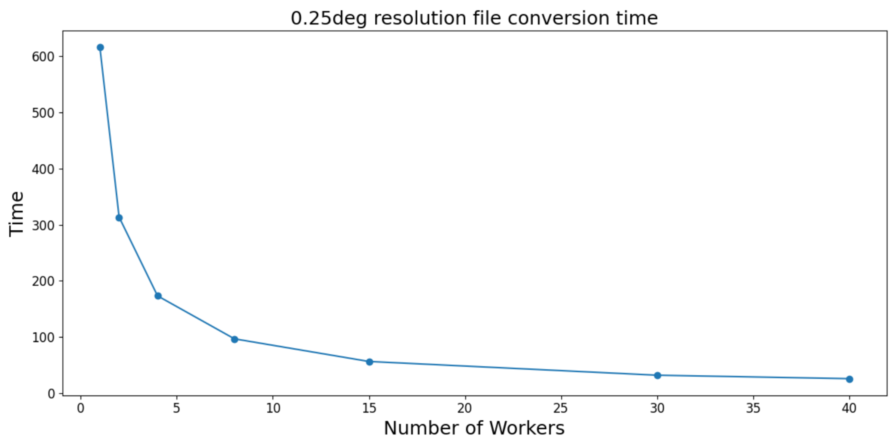


Figure 3.7: The plot illustrates the execution times for converting a full-resolution file from NetCDF to Zarr, varying the number of workers.

From the plots, it's evident that employing parallelization offers significant time savings in execution. The curve consistently shows a downward trend, indicating that the script hasn't reached its full

[2]This value corresponds to the maximum positive integer that can be stored in a 32-bit signed integer.

potential, and increasing the number of workers would likely lead to further reductions in execution times. This trend is similarly observed in the regular file system, where additional tests reveal a similar pattern for values up to 120 workers. Further testing would be required to assess how many workers can be employed before execution times level off. This conclusion is supported by the speedup graph, which demonstrates the ratio between serial and parallel computations. The data points align with the expected sublinear curve for this type of computation. However, when examining efficiency in Figure 3.8, it becomes apparent that the latter decreases notably as the number of workers increases.

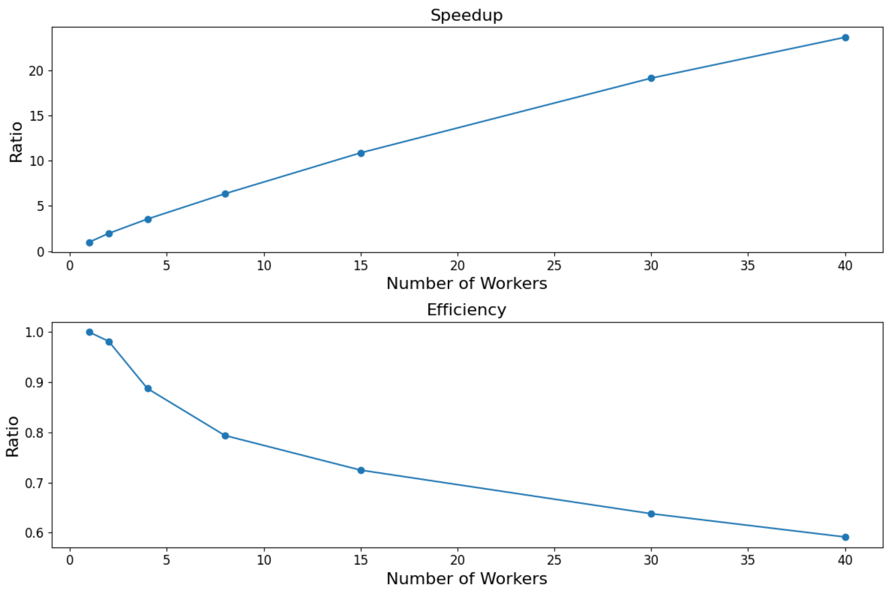


Figure 3.8: Plots of speedup and efficiency relative to the number of workers used. Execution times used for calculating the metrics are the same as in Figure 3.7

Experiments conducted across files with various resolutions consistently demonstrate that parallelization exhibits improvements that are directly proportional to the files' (and chunks) sizes as summarized in Figure 3.9 and more in details in Appendix B. Moreover, executions on larger files also tend to display greater consistency in terms of the time required for execution. Execution times are closely linked with HPC system performance, renowned for its intricate infrastructure and maintenance demands. Throughout the experiments, I had visibility into job scheduler information, revealing the presence of up to 200,000 concurrent processes operating simultaneously. This significant volume of jobs produces a considerable amount of data, potentially overwhelming both the storage system and network fabric. Consequently, performance is often affected by the concurrent execution of numerous processes, resembling the resource contention within a local machine. With a high number of parallel operations, these disparities can be smoothed out, leading to enhanced result reliability, particularly when working with larger files. Notably, during the experiments, it was observed that execution times could vary by up to 70% across multiple runs.

In the last strong scaling experiment, a different approach was taken regarding the chunk dimensions used for all worker configurations. Previously, the focus was on evenly distributing tasks to workers with the least number of chunks, which required adapting the chunk sizes based on the number of workers. For example, when running on a 1.4-degree file with 8 workers, the chunks had dimensions $\{time$: (1095,), $plev$: (13,), lat: (128,), lon: (256,)$\}$, resulting in a disk size of 1.8 GB. In contrast, when running with 40 workers on the same file, the chunks had dimensions $\{time$: (219,), $plev$: (13,), lat: (128,), lon: (256,)$\}$, reducing the disk size to 373 MB. This approach optimized the chunk sizes based on the number of workers and the file size.However, in the subsequent set of runs, the chunk dimensions were fixed to the maximum dimensions that the system could handle. This approach was particularly meaningful for larger files, as it prevented situations where a high number of workers would have most of them idling due to a small number of chunks. As a result these tests were conducted on the dataset with the highest resolution, resulting in fixed chunk dimensions of $\{time$:

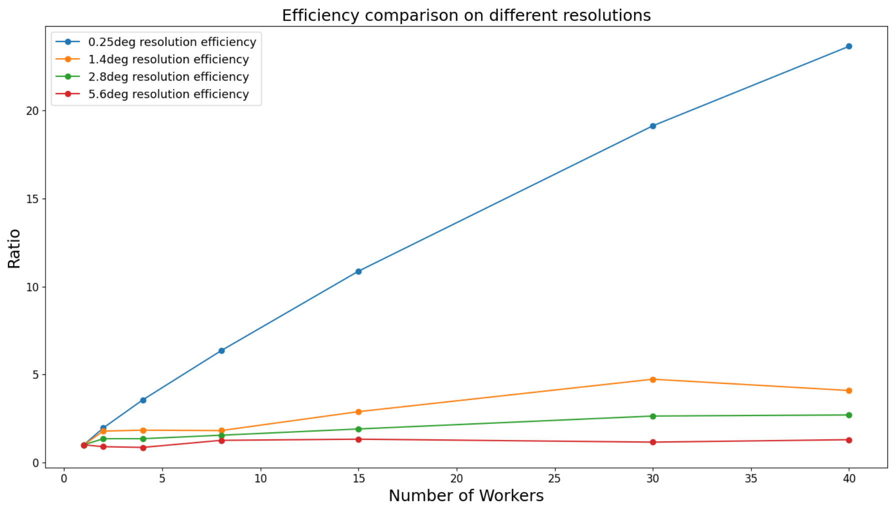


Figure 3.9: The figure depicts efficiency trends calculated under the same experimental conditions, with variations in the input file. It's worth noting that the 0.25-degree resolution exhibits a notably steeper slope compared to the others. This discrepancy arises from a substantial resolution change, which is seven times the factor of the second-best resolution, while the remaining resolutions only exhibit a change by a factor of two.

(820,), *plev*: (13,), *lat*: (103,), *lon*: (480,)}, equal to 2,108,121,600 bytes[3]. Figure 3.10 illustrates the comparison between the standard approach and the fixed max-size chunk. It is evident from the figure that the system performs better when chunk sizes are smaller and tailored to the number of workers involved in the execution.

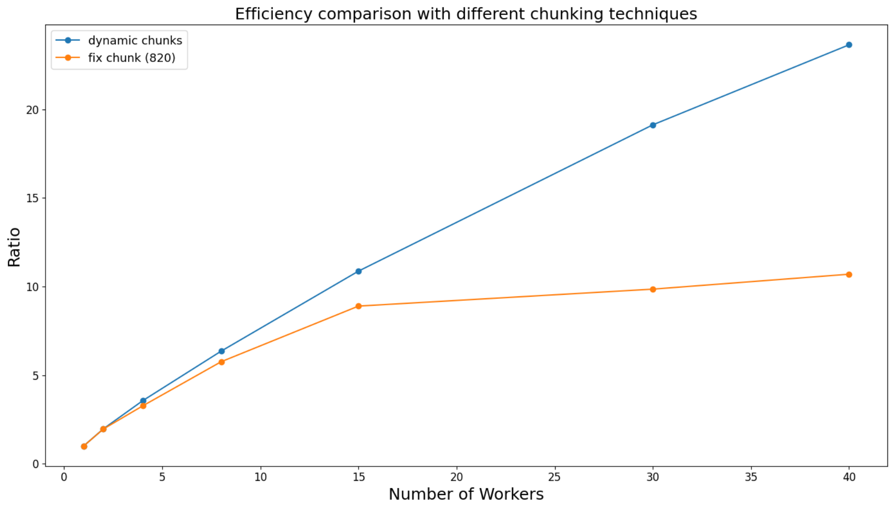


Figure 3.10: The figure shows efficiency trend when using automatic (dynamic) chunk detection and chunks with fixed dimension as previously described.

3.4.1 Weak scaling

Weak scaling involves increasing the overall workload while maintaining it constant for each worker, similar in concept to increasing dataset resolution. To perform benchmarking optimally in an environment with varying NetCDF file sizes, a customizable approach was needed. Since the file sizes at Oak Ridge didn't exhibit consistent growth, the solution was to generate sample data within the script for scalability testing. The generated datasets consist of a 3-dimensional float-type variable. The matrix

[3]Using this approach the full-resolution dataset, with a disk size of approximately 472 GB, is split into around 224 chunks

size remains fixed in two dimensions, {x:(1024,),y:(1024,)}, while the third dimension scales with the number of workers, {z:(256*nWorkers,)}. This approach enables the automatic generation of datasets with varying sizes, making it convenient for testing scalability under different workloads. The experiment involved running the script with varying worker counts, specifically, (1, 2, 4, 8, 16, 32, 40) on the NVMe file system and (1, 2, 4, 8, 16, 32, 64, 128) on the Alpine file system. The output is analogous to the one generated with the other scripts and times are once again extracted and plotted by an external utility. Weak scaling analysis plays a crucial role in assessing the parallelization capabilities of a program, especially when dealing with large datasets and distributed computing. Typically, in weak scaling experiments, one may expect a decrease in efficiency as more workers are involved in the computation. This decline in efficiency can often be attributed to communication overhead, where workers need to coordinate and exchange data, leading to synchronization delays. In this specific case, communication overhead is less of a concern because the workers can operate independently on their respective data portions. However, the performance plots indicate an unusual deterioration in execution time as more workers are added, expecially when reaching node saturation.

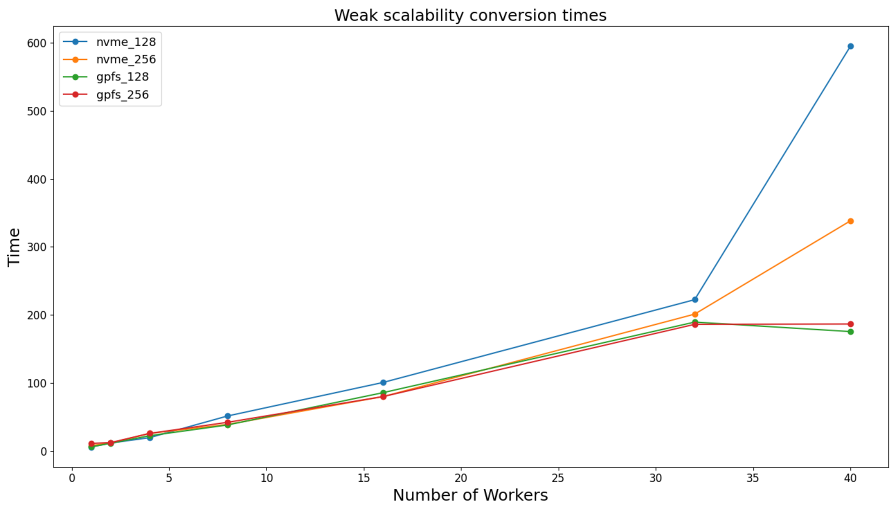


Figure 3.11: The figure depicts execution times observed during the weak scaling experiments. The lines on the plot are labeled with 128 and 256, corresponding to the chunking dimension employed in the experiments. These plots were generated subsequent to the runs to align the two experiments based on the same number of workers, irrespective of the storage system utilized

Examining the execution graph generated by Dask and observing the dashboard, it becomes apparent that tasks are being effectively split among the workers and that they start and execute concurrently. Nevertheless, the variability in execution times between tasks (shown in Figure 3.12) raises questions. While it's challenging to gain a deep understanding of the inner workings of the Dask library at a low level, the gathered information suggests that there might be a bottleneck in the execution process. Since the bottleneck doesn't appear to be related to task distribution or inter-worker communication, the focus turns to the file system. The issue could be attributed to either limitations in the file system's ability to handle parallel writes or the specific method employed by Dask.

3.4.2 Accessing Zarr

Up to this point, our primary focus has been on the conversion procedure that generates the Zarr stores. To gain a comprehensive understanding of Zarr's capabilities, I shifted my attention to reading operations. Xarray provides two distinct functions for reading Zarr and NetCDF files. However, it's important to note that both functions return a Dataset object as a result. This means that once the data is loaded into memory, the underlying format becomes transparent, and both datasets can be handled in the same way. In my evaluation, I meticulously tracked the timing for two distinct phases: opening the datasets and actually loading the data. Xarray's ability to employ lazy evaluation for data loading was a key advantage in this process. This means that I could create a reference to the

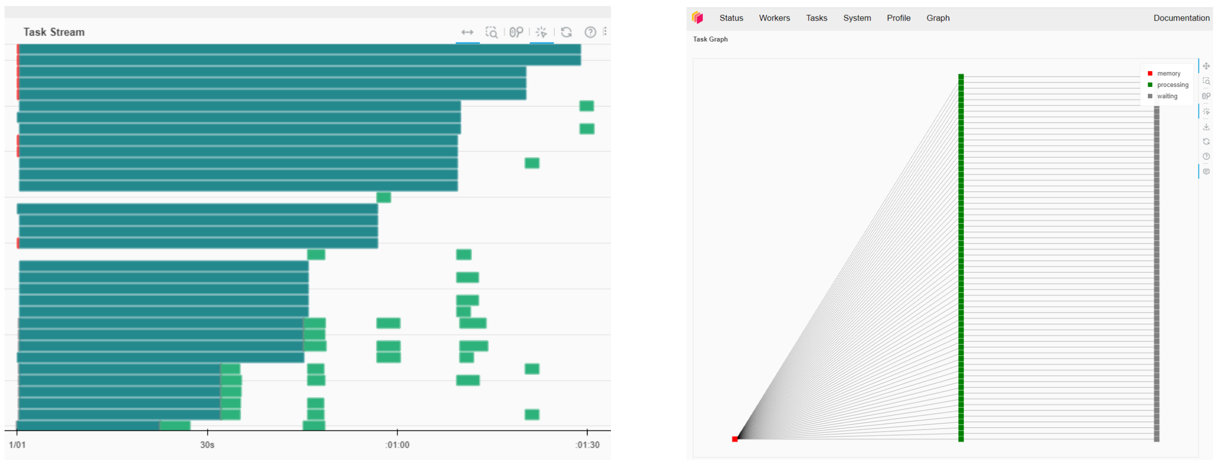


Figure 3.12: On the Dask dashboard, it’s noticeable that the tasks all commence simultaneously. However, considering that they collectively entail identical operations on equivalent subsets, it appears unusual that the majority of tasks exhibit significantly longer execution times, with some taking three times as long to complete. On the other hand the Dask graph illustrates that tasks are entirely self-contained and have no interdependencies on one another.

dataset first and then choose to load only the subsection I required for the subsequent computation. I’ts important to highlight that thanks to the underlying HDF5 implementation, upon which NetCDF is built, I was able to apply this approach to both NetCDF and Zarr formats. The evaluation process was straightforward, I initiated the function to load the dataset based on its format, and then I recorded the times required for subsetting it using the **isel** function. Following this subsetting operation, I invoked the **compute** function on the new dataset. This step explicitly instructed Xarray to load the data into memory, which was sufficient for calculating read performance. The separation of these metrics and the ability to make selective data loads allowed for a comprehensive assessment of the dataset opening and data loading processes.

```python
def read_dataset(data_path,form):
    start=time.time()
    if form=="zarr":
        ds= xr.open_zarr(data_path)
    elif form=="netCDF4":
        ds = xr.open_dataset(data_path)
    else:
        raise ValueError("Data␣format␣not␣supported")
    print("open␣time",time.time()-start)
    start=time.time()
    subset=ds["t"].isel(time=slice(0, 8760)).compute()
    print("read␣time",time.time()-start)
```

The results displayed in Table 3.1 showcase a remarkable contrast in read performance. This has led to some notable questions and uncertainties, especially in comparison to findings in numerous papers on the same subject. It’s important to acknowledge that the approach closely resembled those of other researchers, and thus the cause behind these results likely lies elsewhere. One critical factor that cannot be understated is the difference in the infrastructure used for benchmarking. Variations in hardware, storage systems, and network configurations can indeed lead to substantial performance variations. However, beyond infrastructure disparities, there are several other potential causes to consider.

- **Compression:** The selection of a compression method plays a pivotal role. Opting for the standard Blosc compression in Zarr, when other research papers do not explicitly detail their compression techniques, can exert a substantial influence on read performance.Compression algorithms vary in terms of speed and efficiency, and this can lead to divergent results.

- **Caching:** Caching mechanisms, whether implemented at the application, file system, or hardware level, possess the capacity to exert a considerable impact on read performance. It's worth noting that if other research studies explicitly incorporated caching strategies or had caching mechanisms integrated into their benchmarking environments, this could explain the disparities in results. Nonetheless, it's essential to emphasize that in my research, I purposefully opted for the employment of burst buffers. This decision aligns with the specific objectives and methodology of the study, as it was aimed at obtaining unadulterated data without the interference of caching mechanisms.

- **Concurrency:**The level of concurrency in benchmarking experiments might differ from that in other studies. Concurrent access and read operations can affect overall performance, especially in shared storage environments.

Workers	**Chunk dims**	**Slice dims**	**Slice size (B)**	**Zarr times (s)**	**NetCDF4 times (s)**	**Notes**
4	(1095, 13, 128, 256)	(1, 13, 128, 256)	1703936	7.463	0.002	random access to time slice
4	(1095, 13, 128, 256)	(3, 13, 128, 256)	5111808	11.484	0.002	time slices on different chunks
4	(1095, 13, 128, 256)	(3, 13, 128, 256)	5111808	6.784	0.003	time slices on same chunk
4	(1095, 13, 128, 256)	(1, 1, 1, 1)	4	7.491	0.002	single data point
4	(1095, 13, 128, 256)	(8760, 13, 128, 256)	14926479360	18.945	2.058	whole dataset
4	(597, 6, 68, 137)	(1, 13, 128, 256)	1703936	9.349	0.002	random access to time slice
4	(597, 6, 68, 137)	(3, 13, 128, 256)	5111808	8.992	0.003	time slices on different chunks
4	(597, 6, 68, 137)	(3, 13, 128, 256)	5111808	6.990	0.003	time slices on same chunk
4	(597, 6, 68, 137)	(1, 1, 1, 1)	4	7.066	0.002	single data point
4	(597, 6, 68, 137)	(8760, 13, 128, 256)	14926479360	20.553	2.038	whole dataset
4	(1, 13, 128, 256)	(1, 13, 128, 256)	1703936	7.098	0.002	random access to time slice
4	(1, 13, 128, 256)	(3, 13, 128, 256)	5111808	8.760	0.003	time slices on different chunks
4	(1, 13, 128, 256)	(1, 1, 1, 1)	4	6.441	0.002	single data point
4	(1, 13, 128, 256)	(8760, 13, 128, 256)	14926479360	90.749	2.511	whole dataset

Table 3.1: Table comparing read times for NetCDF4 and Zarr Datasets using 4 workers

4 Conclusions

4.1 Darshan

To better understand the results observed in the weak scaling and reading experiments, which may differ from other benchmarks conducted within different settings, it becomes imperative to introduce additional tools that facilitate a more in-depth analysis. These tools play a pivotal role in the comprehensive examination, offering valuable insights and enhancing our understanding of the experiments. These disparities may stem from various sources, and there are two primary avenues for investigation. The first path involves an in-depth exploration of the intricacies of the HPC system itself. Factors within the HPC environment, such as system configuration, resource allocation, and system load, can exert a substantial influence on performance. This angle of investigation necessitates a meticulous study of the HPC system's components and settings, aiming to identify potential sources of performance disparities. The second avenue of inquiry directs attention to the implementation of the conversion and data reading processes. Understanding the intricacies of how these processes are executed can reveal critical insights into the observed variations. It is at this juncture that the utility of Darshan, a comprehensive I/O profiling tool, comes into play. Darshan is a powerful tool that facilitates the profiling and analysis of I/O operations within an application or program. It tracks and records detailed information about file access, including the number of I/O operations, data transfer sizes, and the duration of these operations. This tool offers a granular view into the interactions between the application and the underlying file system, providing valuable data on I/O patterns and access behaviors. By leveraging Darshan, comprehensive insights can be obtained into how the NetCDF to Zarr conversion and data reading operations interact with the file system at a low level. Darshan's insights can highlight potential bottlenecks or inefficiencies in the I/O operations, shedding light on issues that may not be readily apparent through traditional performance monitoring. For instance, Darshan can reveal if there are recurrent patterns of small, inefficient I/O operations or if certain system calls are being frequently invoked, suggesting areas for optimization. It can pinpoint whether data access patterns are suboptimal, leading to performance degradation. In essence, Darshan acts as a forensic tool for scrutinizing the runtime behavior of I/O operations. By integrating Darshan into research endeavors, hidden factors contributing to disparities in the weak scaling and reading experiments can be uncovered. This comprehensive analysis, combining insights from system-level examination and Darshan's granular I/O profiling, offers a robust methodology for diagnosing and addressing performance anomalies. It not only contributes to a more complete understanding of the observed results but also paves the way for fine-tuning the implementation, ultimately bringing research closer to the state of the art.

4.2 Final remarks

Throughout this thesis, the primary focus has been on the parallelized conversion of NetCDF to Zarr files, with a particular emphasis on executing these conversions in HPC environments, notably on the Summit supercomputer at Oak Ridge National Laboratory (ORNL). The motivation behind this research was to evaluate the emerging Zarr data format in contrast to the widely-used NetCDF format, which may be showing signs of obsolescence. The investigation into concurrent writes demonstrated the significant impact of the choice of the environment on execution times. Python and Dask were employed to facilitate task distribution during the testing process, but it is essential to note that the potential overhead introduced by Dask should be examined further in future work, as the parallelization of the conversion task may not warrant its use. The research findings pertaining to burst buffers revealed impressive parallelization efficiency, with values exceeding 70% up to approximately 20 workers. Even beyond that, performance remained relatively high, at around 60%, until node saturation was reached. These results indicate that the conversion problem scales well, especially in an HPC environment. However, the outcomes of weak scaling experiments hinted at potential bottlenecks in the conversion pipeline. Identifying these bottlenecks can be challenging, but the inclusion of additional tools such as Darshan may provide deeper insights into low-level operations and their impact on execution. Lastly, the study delved into read operations, which revealed that in the specific HPC environment considered, with the compression settings used and the reading methods employed, NetCDF4 reads were considerably faster than Zarr reads.

The benchmarking and experimentation conducted in this thesis have far-reaching implications. They underscore the importance of assessing and comparing data formats such as NetCDF and Zarr in HPC environments. The ability to efficiently convert and work with scientific data is of paramount significance, particularly in domains where large datasets are the norm. As scientific research increasingly relies on high-performance computing, the choice of data format can have a profound impact on the speed and efficiency of data processing, analysis, and sharing. While this research has provided valuable insights, there are avenues for future work that can build upon the foundation laid in this thesis. First and foremost, the question of Dask's overhead in simple parallelization tasks warrants further investigation. Future research should delve into optimizing the use of Dask for this specific application, balancing its benefits with potential computational costs. Additionally, addressing the bottlenecks observed in the weak scaling experiments remains a challenge. The incorporation of tools like Darshan can shed light on the low-level operations responsible for slowdowns and lead to strategies for mitigation. Furthermore, it is essential to recognize that the choice between NetCDF and Zarr may not be one-size-fits-all. The research has pointed out that in the specific environment and with the configurations used, NetCDF4 outperformed Zarr for read operations. Further comparative studies in diverse HPC environments and with varying compression and reading methods can provide a comprehensive understanding of when to employ each format optimally. In conclusion, this thesis has made significant strides in exploring the conversion of NetCDF to Zarr in HPC environments. It has underscored the importance of benchmarking data formats, highlighted their potential for innovation, and laid the groundwork for future research to refine and expand upon these findings. As scientific data continues to grow in complexity and volume, the effective management and utilization of such data will be a defining factor in the success of scientific endeavors.

The results of this research will be published in the upcoming petascale data hackathon report, which is significant for the scientific community. This report not only summarizes the findings from our study but also serves as a hub for sharing knowledge and fostering innovation. By sharing our research, we aim to improve our collective understanding of data management in high-performance computing. This, in turn, can lead to advancements in fields like weather and climate prediction, making our predictions more accurate. Additionally, it can inspire the development of new technologies, like artificial intelligence and machine learning applications, as well as data assimilation systems, which can help us gather and interpret data from Earth observations and extreme weather events.

Bibliography

[1] Andes user guide. https://docs.olcf.ornl.gov/systems/andes_user_guide.html. last acces 28/09/2023.

[2] Datacarpentry netcdf guide. https://datacarpentry.org/python-oceanography-lesson. last acces 23/09/2023.

[3] Decrease geospatial query latency from minutes to seconds using zarr on amazon s3. https://aws.amazon.com/blogs/publicsector/decrease-geospatial-query-latency-minutes-seconds-using-zarr-amazon-s3/. last acces 23/09/2023.

[4] Gnu autoconf. https://www.gnu.org/software/autoconf. last acces 28/09/2023.

[5] Nasa panoply tool. https://www.giss.nasa.gov/tools/panoply. last acces 09/10/2023.

[6] Oak ridge national laboratory data hackaton. https://www.olcf.ornl.gov/ifs-nr-data-hackathon. last acces 23/09/2023.

[7] Oak ridge national laboratory website. https://www.olcf.ornl.gov/. last acces 23/09/2023.

[8] Python multithreading performance. http://www.dabeaz.com/python/GIL.pdf. last acces 3/10/2023.

[9] Slurm: Simple linux utility for resource management. https://slurm.schedmd.com/slurm_design.pdf. last acces 28/09/2023.

[10] Top500 supercomputer list. https://www.top500.org/lists/top500/2023/06. last acces 28/09/2023.

[11] Xarray installation page. https://xarray.pydata.org/en/v0.14.1/installing.html. last acces 29/09/2023.

[12] Xarray suggestion for handling zarr store. https://docs.xarray.dev/en/stable/user-guide/io.html#appending-to-existing-zarr-stores. last acces 29/09/2023.

[13] Sriniket Ambatipudi and Suren Byna. A comparison of hdf5, zarr, and netcdf4 in performing common i/o operations. 2023.

[14] Lorenzo Dell'Osso. High-resolution experiments with the ecmwf model: A case study. *Monthly Weather Review*, 112(9):1853 – 1884, 1984.

[15] Veronica Melesse Vergara, Wayne Joubert, Michael Brim, Reuben Budiardja, Don Maxwell, Matthew Ezell, Christopher Zimmer, Swen Boehm, Wael Elwasif, Sarp Oral, Christopher Fuson, Daniel S. Pelfrey, Oscar Hernandez Mendoza, Dustin B. Leverman, Jesse Hanley, Mark Berrill, and Arnold Tharrington. Scaling the summit: Deploying the world's fastest supercomputer. 6 2019.

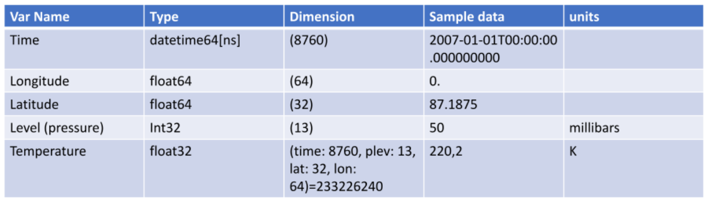
[16] Dieu Nguyen, Johana Cortes, Marina Dunn, and Alexey Shiklomanov. Impact of chunk size on read performance of zarr data in cloud-based object stores. 03 2023.

[17] Stephan Rasp, Peter D. Dueben, Sebastian Scher, Jonathan A. Weyn, Soukayna Mouatadid, and Nils Thuerey. Weatherbench: A benchmark data set for data-driven weather forecasting. *Journal of Advances in Modeling Earth Systems*, 12(11):e2020MS002203, 2020. e2020MS002203 10.1029/2020MS002203.

[18] Karl E. Taylor, Ronald J. Stouffer, and Gerald A. Meehl. An overview of cmip5 and the experiment design. *Bulletin of the American Meteorological Society*, 93(4):485 – 498, 2012.

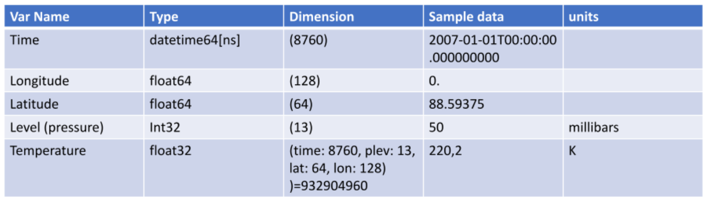
[19] Haiying Xu, Kevin Paul, and Anderson Banihirwe. Pangeo benchmarking analysis: Object storage vs. posix file system. In *2020 IEEE/ACM Fifth International Parallel Data Systems Workshop (PDSW)*, pages 40–45, 2020.

Appendix A Summary of dataset resolutions details



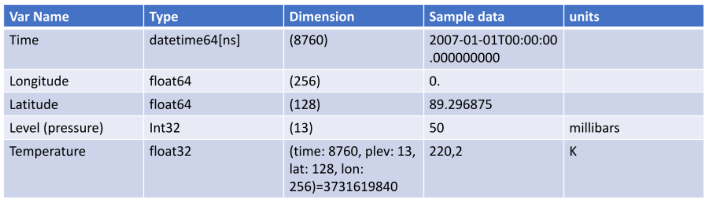
Var Name	Type	Dimension	Sample data	units
Time	datetime64[ns]	(8760)	2007-01-01T00:00:00 .000000000	
Longitude	float64	(64)	0.	
Latitude	float64	(32)	87.1875	
Level (pressure)	Int32	(13)	50	millibars
Temperature	float32	(time: 8760, plev: 13, lat: 32, lon: 64)=233226240	220,2	K

Figure A.1: File details for the temperature variable at a resolution of 5.6 degrees (0.9GB)



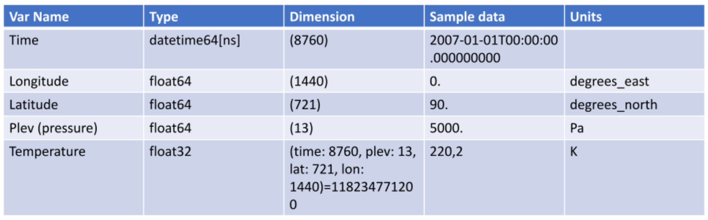
Var Name	Type	Dimension	Sample data	units
Time	datetime64[ns]	(8760)	2007-01-01T00:00:00 .000000000	
Longitude	float64	(128)	0.	
Latitude	float64	(64)	88.59375	
Level (pressure)	Int32	(13)	50	millibars
Temperature	float32	(time: 8760, plev: 13, lat: 64, lon: 128))=932904960	220,2	K

Figure A.2: File details for the temperature variable at a resolution of 2.8 degrees (3.7GB)



Var Name	Type	Dimension	Sample data	units
Time	datetime64[ns]	(8760)	2007-01-01T00:00:00 .000000000	
Longitude	float64	(256)	0.	
Latitude	float64	(128)	89.296875	
Level (pressure)	Int32	(13)	50	millibars
Temperature	float32	(time: 8760, plev: 13, lat: 128, lon: 256)=3731619840	220,2	K

Figure A.3: File details for the temperature variable at a resolution of 1.4 degrees (14.9GB)



Var Name	Type	Dimension	Sample data	Units
Time	datetime64[ns]	(8760)	2007-01-01T00:00:00 .000000000	
Longitude	float64	(1440)	0.	degrees_east
Latitude	float64	(721)	90.	degrees_north
Plev (pressure)	float64	(13)	5000.	Pa
Temperature	float32	(time: 8760, plev: 13, lat: 721, lon: 1440)=11823477120 0	220,2	K

Figure A.4: File details for the temperature variable at a resolution of 0.25 degrees (472.9GB)

Appendix B Benchmarks with different data resolution

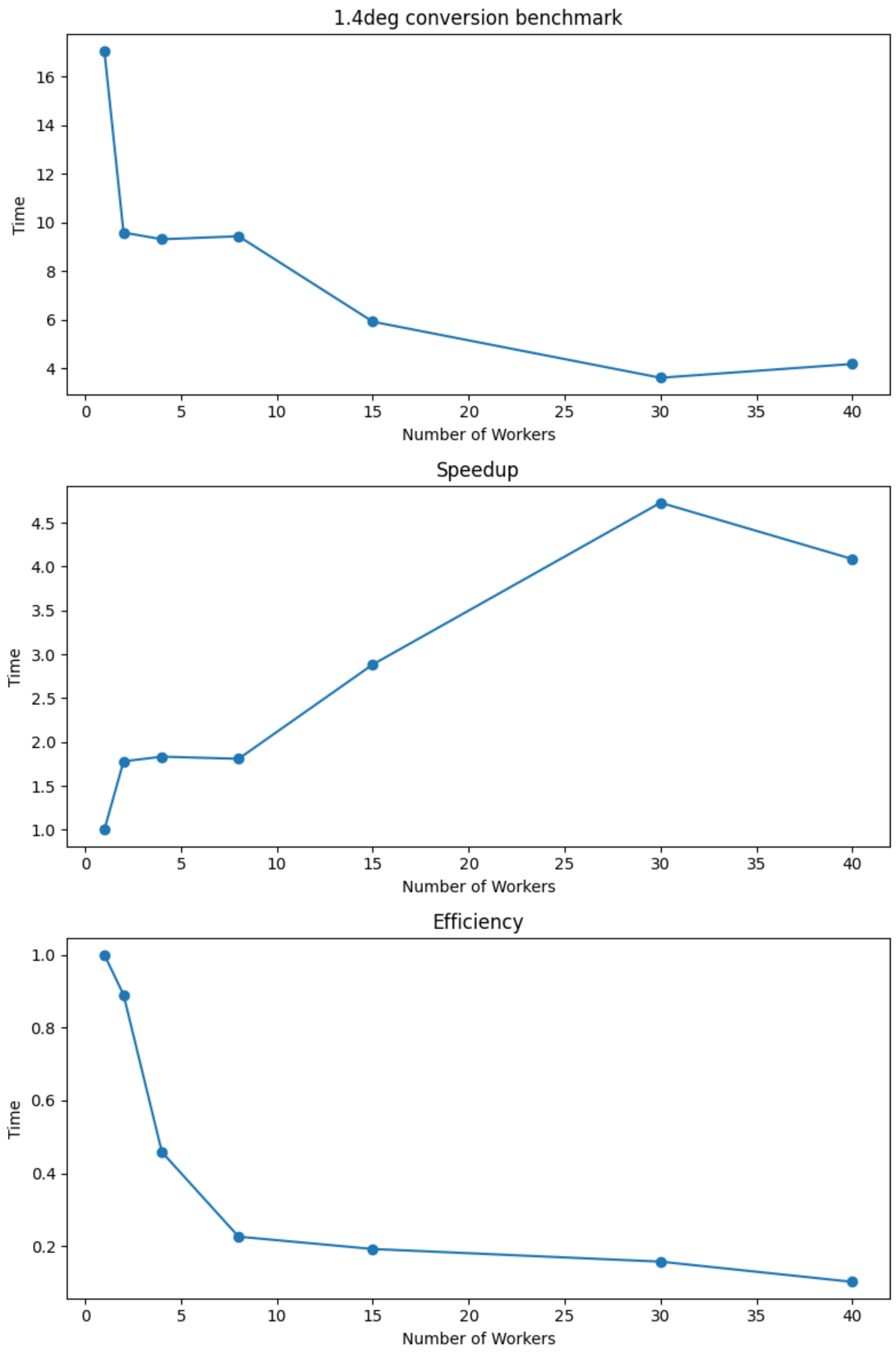


Figure B.1: Data obtained by converting a NetCDF Dataset with 1.4deg resolution

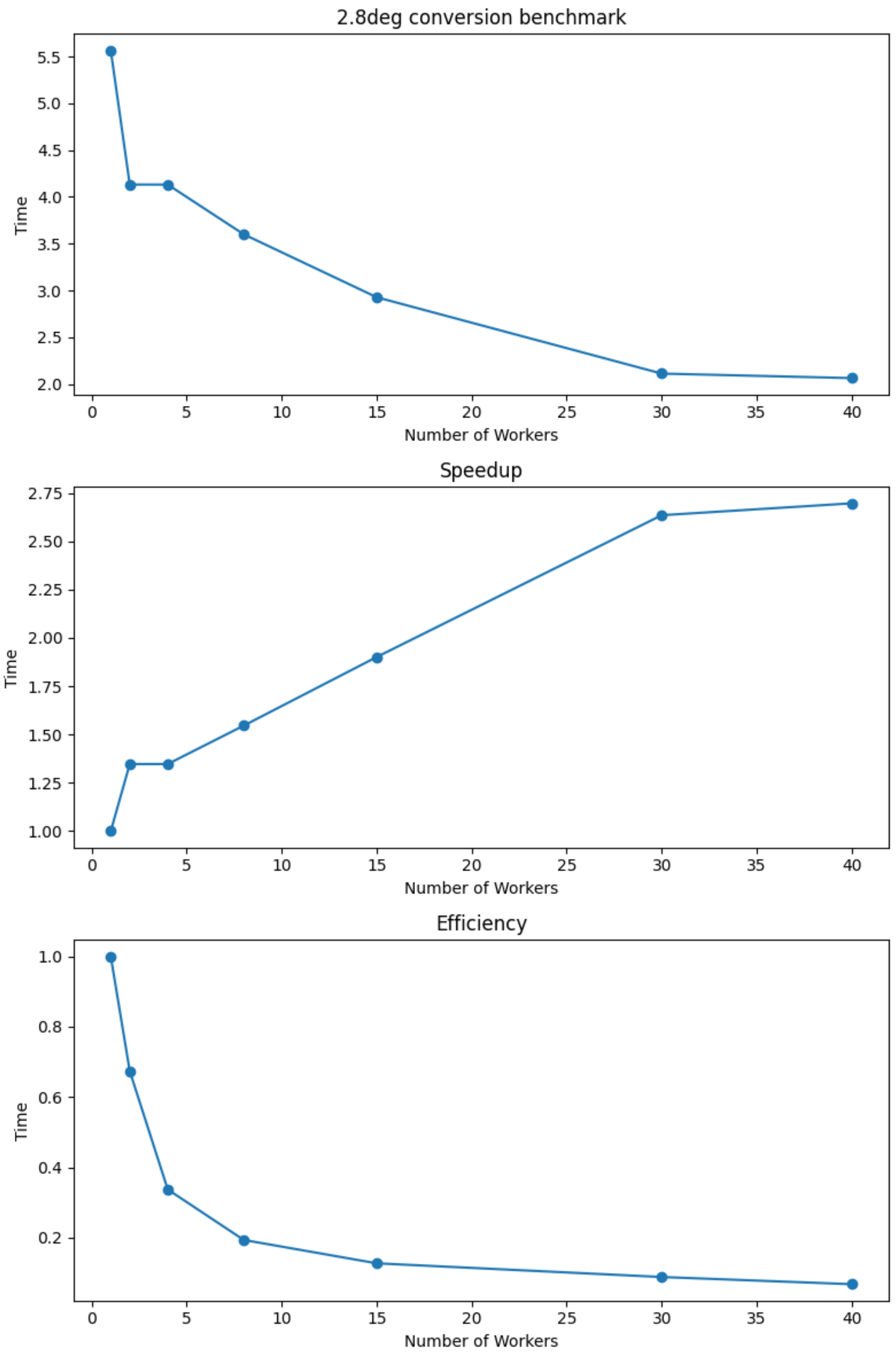


Figure B.2: Data obtained by converting a NetCDF Dataset with 2.8deg resolution

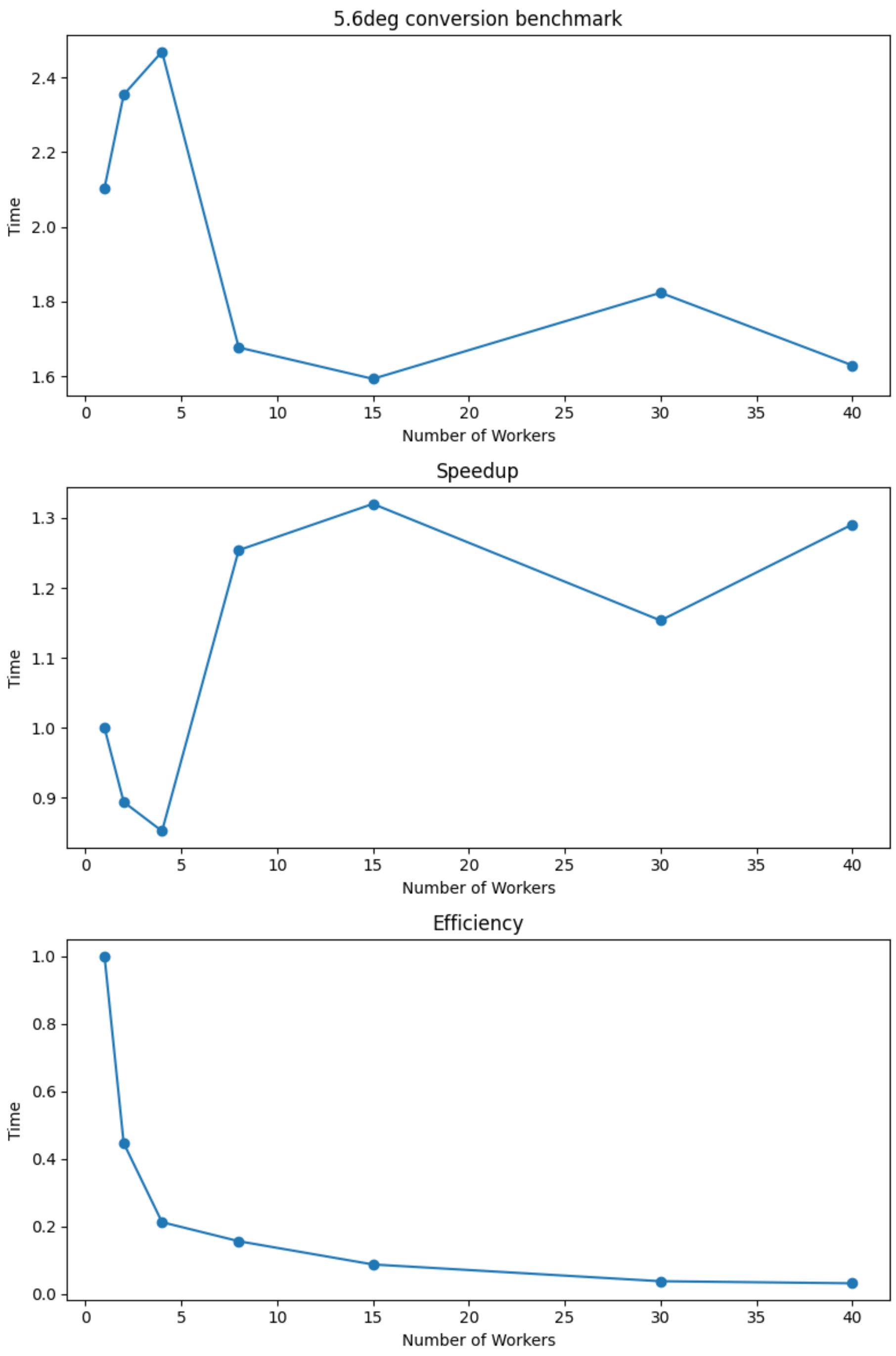


Figure B.3: Data obtained by converting a NetCDF Dataset with 5.6deg resolution

Appendix C Times for additional Read experiments

# Worker	chunk dimension	slice dimension	slice size (B)	zarr time (s)	Netcdf4 time (s)	notes
1	(1095, 13, 128, 256)	(1, 13, 128, 256)	1703936	7.056	0.002	random access to time slice
1	(1095, 13, 128, 256)	(3, 13, 128, 256)	5111808	10.154	0.002	time slices on different chunks
1	(1095, 13, 128, 256)	(3, 13, 128, 256)	5111808	6.818	0.003	time slices on same chunk
1	(1095, 13, 128, 256)	(1, 1, 1, 1)	4	6.831	0.002	single data point
1	(1095, 13, 128, 256)	(8760, 13, 128, 256)	14926479360	29.376	2.075	whole dataset
1	(597, 6, 68, 137)	(1, 13, 128, 256)	1703936	9.324	0.002	random access to time slice
1	(597, 6, 68, 137)	(3, 13, 128, 256)	5111808	8.742	0.003	time slices on different chunks
1	(597, 6, 68, 137)	(3, 13, 128, 256)	5111808	7.016	0.003	time slices on same chunk
1	(597, 6, 68, 137)	(1, 1, 1, 1)	4	6.667	0.002	single data point
1	(597, 6, 68, 137)	(8760, 13, 128, 256)	14926479360	32.860	2.029	whole dataset
1	(1, 13, 128, 256)	(1, 13, 128, 256)	1703936	6.797	0.002	random access to time slice
1	(1, 13, 128, 256)	(3, 13, 128, 256)	5111808	8.752	0.004	time slices on different chunks
1	(1, 13, 128, 256)	(1, 1, 1, 1)	4	7.076	0.002	single data point
1	(1, 13, 128, 256)	(8760, 13, 128, 256)	14926479360	366.309	2.475	whole dataset
8	(1095, 13, 128, 256)	(1, 13, 128, 256)	1703936	7.148	0.002	random access to time slice
8	(1095, 13, 128, 256)	(3, 13, 128, 256)	5111808	10.150	0.002	time slices on different chunks
8	(1095, 13, 128, 256)	(3, 13, 128, 256)	5111808	6.798	0.003	time slices on same chunk
8	(1095, 13, 128, 256)	(1, 1, 1, 1)	4	6.818	0.002	single data point
8	(1095, 13, 128, 256)	(8760, 13, 128, 256)	14926479360	18.869	2.057	whole dataset
8	(597, 6, 68, 137)	(1, 13, 128, 256)	1703936	7.006	0.002	random access to time slice
8	(597, 6, 68, 137)	(3, 13, 128, 256)	5111808	8.726	0.003	time slices on different chunks
8	(597, 6, 68, 137)	(3, 13, 128, 256)	5111808	7.303	0.003	time slices on same chunk
8	(597, 6, 68, 137)	(1, 1, 1, 1)	4	7.136	0.002	single data point
8	(597, 6, 68, 137)	(8760, 13, 128, 256)	14926479360	21.278	2.036	whole dataset
8	(1, 13, 128, 256)	(1, 13, 128, 256)	1703936	6.797	0.002	random access to time slice
8	(1, 13, 128, 256)	(3, 13, 128, 256)	5111808	9.018	0.003	time slices on different chunks
8	(1, 13, 128, 256)	(1, 1, 1, 1)	4	6.860	0.002	single data point
8	(1, 13, 128, 256)	(8760, 13, 128, 256)	14926479360	87.174	2.484	whole dataset