Efficient Delivery of Geospatial Data for Web Visualization

A Major Qualifying Project (MQP) Report
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the Degree of Bachelor of Science in

Computer Science

By:

Deepti Gosukonda
Nadav Konstantine
Nicholas Markou
Patrick Salisbury

Project Advisor:

Lane Harrison

Sponsored By:

Tetra Tech

Date: March 21st 2024

Abstract

Geospatial data is a widely used format for visualization and analysis of various location-embedded datasets applied in fields such as climatology, geology, and oceanography. Analyzing and rendering this data is computationally expensive; optimized processes and caching solutions are crucial in keeping this service responsive in modern cloud-based environments. This project benchmarked a widely used Web Map Service (ncWMS) against Xpublish-wms, a prototype Web Map Service. The team systematically identified and optimized inefficient code sections, and implemented caching methods to increase performance and decrease the experienced latency for end users. These changes result in reduced processing power to visualize the data and lower operational costs in the cloud.

Acknowledgments

The opportunity to participate in experimenting with web map service models and data delivery optimizations was made possible through the support of Worcester Polytechnic Institute's Innovation and Entrepreneurship Center for Major Qualifying Projects. This center, along with its network of affiliated and non-affiliated individuals, facilitated connections with the project sponsor, Tetra Tech.

To Professor Lane Harrison, our advisor, we would like to thank you for your invaluable support in building up our efforts to accomplish the work on this project. As we improve our technical and professional skills, we hope to continue following your guidance and wisdom post-completion of our academic careers. We could not have completed our Major Qualifying Project without you.

To our sponsor, Matthew Iannucci of Tetra Tech, thank you for your dedication in explaining the necessary software, establishing a clear project scope, and consistently touching base with the team as the work on this project progressed. For the feedback and strive to provide the team with the necessary information for project deliverables, we value your active involvement in our work.

To our sponsor, James Doyle of Tetra Tech, thank you for your expertise in project management, diligently organizing and participating in bi-weekly progress meetings, and continuous advice for the team throughout the project. Despite corporate logistics, we appreciate your hard work in ensuring the continuity and success of the student programs.

To our sponsor, Jonathan Joyce of Tetra Tech, thank you for your day by day approval of the work we were able to accomplish. Your validation of our open-source project work is held in the highest regard by our team. As a technical lead in your field, your insight is always deeply appreciated and respected.

To our sponsor, Donald Moretti of Tetra Tech, thank you for overseeing our project operations and incorporating marketing technology strategies in our project's completion. We are grateful for your involvement as a pipeline manager in Tetra Tech's sponsorship.

Contents

List of Tables

List of Figures

1 Introduction

1.1 Geospatial Data

Imagine being able to witness the dynamic shifts and large-scale processes of the world unfold in real time. This is the transformative power of geospatial data. Geospatial data is a subset of data that contains location-embedded information, allowing the data to be mapped across the Earth's surface. This type of data is used in many applications, from everyday tasks like GPS navigation to intricate analyses in fields like climatology, geology, and oceanography. The widespread use of the applications of geospatial data emphasizes its importance in modern geoscience, and a considerable number of technologies in the field would be possible if not for geospatial data analytics and visualization. Geographic Information Systems (GIS) is a prime example of a use case for geospatial visualization. They organize geospatial data based on location and transform it into a tangible representation through mapping, fostering subsequent analysis. [2]. Geospatial data is often updated regularly, especially datasets that pertain to real-time applications, such as real-time ocean forecasting models. This periodic data cycle means that geospatial data systems can quickly render newly released data, which can be very effective in identifying patterns and trends in real-time processes [3].

Geospatial datasets can be gathered from an area of the Earth's surface directly (eg. satellite imagery/LIDAR data) or the data can be generated by interpolating sensor data through computer models. The typical use of these datasets involves requesting various information to be mapped (via specific longitudinal and latitudinal labels) in order to analyze a particular geographical area of interest. Visualization of such geospatial data can combine numerous categories of data sources (such as salinity, satellite data, etc) onto a single image. Many of the geospatial datasets that were benchmarked and optimized during this project were complex in nature; as they were intended for equally complex visualization and analysis applications.

1.2 Challenges of Data Processing

The challenges posed by the high computational demands of geospatial data analysis emphasize the need for efficient data processing systems and practices, particularly in the context of cloud computing environments where computation time is a valuable resource. An immense volume of geospatial data is generated every day from various sources, and in some cases new data is released on a one-to-three-hour

basis for certain models, further increasing the amount of available data [4]. According to the estimation by the United Nations Initiative on Global Geospatial Information Management (UN-GGIM), 2.5 quintillion bytes of data are being generated every day, and a large portion of the data is location-aware [5]. To tap into this real-time data, a geospatial data processing system must regularly access newly released datasets and process them to fit the needs of that system's use case. For a system that makes many frequently updated datasets available to the end user, this can place huge computational loads on the system as it constantly processes newly available data in real time.

Additionally, the complexity of some geospatial datasets means that long and computationally intense techniques to process the data into a usable visualization are not uncommon. Some datasets organize the geospatial data in ways that are not straightforward to convert into a map-type visualization directly, adding a complex "projection" step where the dataset is transformed before it can even be rendered. Another complex aspect of working with geospatial datasets can be their sheer size alone. When analyzing and processing geospatial data, some data sources can produce high-resolution grids that may consist of millions of individual data points. The National Oceanic and Atmospheric Administration (NOAA), for example, hosts datasets with timestamps that range from 1895 to the present and also uses specific ocean data that can reach a grid size of 4500x3298 [4]. These NOAA datasets are an essential tool in modern oceanography due to their large assortment of data with an oceanographic focus, which often update at regular intervals allowing for the most recent data to be accessed and analyzed. A geospatial visualization analyst can request a small sample size of the available data to view, which queries data from the entire dataset and renders an image for the end user.

Like many other performance problems, these challenges may be directly addressed by implementing optimization methods to increase the performance and efficiency of geospatial data processing systems. However, without first understanding the specific challenges involved in geospatial data processing, the team would not have the knowledge required to implement these optimizations in the first place.

1.3 Project Objectives

The goal of this project is to investigate Web Map Service (WMS) requests to optimize map rendering and reduce the latency of data delivery. To achieve the project's goal, the team developed the following objectives:

1. Benchmark an existing WMS server (ncWMS) to identify areas of slowdown in the existing mapping processes.
2. Compare the performance of the prototype WMS (Xpublish-wms) with the existing system and investigate areas for improvement.
3. Implement new optimization methods to increase server performance and reduce the number of operations done on the server via caching strategies.

2 Background

2.1 Tetra Tech

The team's sponsor for this project, Tetra Tech (formerly RPS Group), is a leading global provider of consulting and engineering services [6]. Tetra Tech operates across twelve diverse service clusters. Collaborating with their Ocean Science Division Team, this project's involvement extends to their product, OCEANSMAP (Appendix A). This tool serves multiple functions, encompassing data management, maritime planning, water quality assessment, and operational response planning. Derived primarily from NOAA models, these layers are meticulously organized by parameter, with point observation layers depicted as pins on the map. OCEANSMAP enables users to harness the potential of inputted data, allowing them to access, visualize, and interpret a broad range of environmental data from around the globe, including both real-time observations and model forecasting [6]. Featuring web-based met-ocean (meteorological-oceanographic) data and an emergency response simulation system, OCEANSMAP includes a user-friendly map interface, a database for spatial data, and web services from the OCEANSMAP server. The web interface facilitates users and groups in managing, visualizing, and analyzing various met-ocean data, incorporating operational models and on-site measurements. This functionality can be used in applications such as rapid contingency planning, spill response forecasting, and resource management. Notably, major industry players like Shell and Avangrid are among the distinguished clients who utilize this product for their diverse needs.

2.2 GIS

Geographic Information System (GIS) software is a tool used to analyze and visualize spatial data, allowing users to utilize maps that are easily translatable to the real world. GIS is an essential resource in various fields, from urban planning and environmental management to emergency response and business intelligence. By integrating data from multiple sources, such as maps, satellite imagery, and databases, GIS software helps users identify patterns, trends, and relationships within the data, ultimately improving planning and decision-making processes [7]. OCEANSMAP, just like many other web map services, is a simplified GIS that includes the features needed to make data-driven decisions without requiring to learn a complex interface, which other traditional GIS systems can often require. Additionally, web-based GIS allows data normally restricted to desktops to be viewed from a greater number of devices and users, reducing the client's computational load.

2.3 Geospatial Data Formats

In the realm of geospatial data delivery, some major data formats that prevail over other less common approaches are NetCDF and GRIB. GRIB is the World Meteorological Association (WMO) standardized format for gridded data. Because these data formats are used so interchangeably, any versatile WMS needs to be able to efficiently handle both data formats. In this project, the team regularly put both GRIB and NetCDF data formats to use depending on the chosen data source, so understanding the nuances and implementation details for both major data formats became an obvious objective early on in the project.

2.3.1 NetCDF

NetCDF, or network Common Data Form, is a set of libraries for multidimensional data types or variables for scientific data. NetCDF is a type of file format that displays these variables by creating a view sheet from the netCDF file and projecting them through a dimension such as time. The Unidata program has hosted the data format since its start in 1988 at the University Corporation for Atmospheric Research, and they are the managing power of netCDF's development, documentation, updates, and functionality. The file format that led to the creation of netCDF was inspired by NASA's Common Data Format and has evolved to support multiple binary formats and no longer supports compatibility with NASA's conceptual model [8]. The software's wide range of capabilities is applicable because geospatial data is written in netCDF format, and is then read back through a GIS to overlay on a map.

2.3.2 GRIB

GRIB, or General Regularly distributed Information in Binary, is a binary format designed for storing and distributing gridded meteorological data sponsored by the WMO (World Meteorological Organization). It is widely used in meteorological and oceanographic applications due to its efficiency in storing large volumes of gridded data and is a standard format for archiving and exchanging gridded data [9].

GRIB files are self-contained records of two-dimensional data, where individual records can stand alone as meaningful [9]. This means collections of GRIB records can be appended to each other or separated, and, like NetCDF, is self-describing and portable.

The project required the updated GRIB2 format as opposed to GRIB1, where in GRIB2, several variables are defined with more precision; such as latitudes and longitudes are defined in micro-degrees, as well as the detail that longitude values must be within the range of 0 and 360 degrees. GRIB2 also has the

possibility of compression, which makes it a high priority for transmission formats.

The GRIB2 data format is essential for the team because Tetra Tech's NOAA models store their data in both NetCDF and GRIB formats as a requirement and retrieve data from external sources. Furthermore, GRIB2 is a required and essential tool in this project due to the efficient storage, transmission, and manipulation of gridded data. The widespread use including the ability to store precise and high-resolution makes GRIB2 a highly relevant and valuable resource. This project uses GRIB2 for its enhanced application and compression and is exceptionally useful due to its ability to converge with netCDF; additionally, the web mapping services used throughout this project require the GRIB data format as standard.

2.4 Projections

Map projections are used to represent the three-dimensional surface of the Earth on a flat, two-dimensional surface. Since a perfect representation of the Earth's surface on a flat map is impossible, different map projections are employed to balance and prioritize various properties, such as direction, distance, area, and shape [10].

OCEANSMAP uses WMS (A server standard that's followed for tile generation) to project the dataset to the Web Mercator projection, utilizing EPSG codes to interpret the requested bounding box. Each EPSG code records the reference datum and coordinate system used to create the image [11]. This process of projecting the spherical Earth onto a 2D surface involves mathematical transformations that inevitably introduce distortions [10]. A map projection is chosen based on the specific requirements of the intended use, considering factors like navigational accuracy, accurate representation of distances, preservation of land area, or shape accuracy. No single projection can perform well in all these aspects at once.

Figure 1: A Mercator projection of the world. A type of Mercator projection was used extensively in the project.

Figure 2: Greenland's true size in comparison to the USA. Different projection types can drastically vary the scale of objects on a map.

There are four primary categories of projections: cylindrical, conic, azimuthal, and pseudocylindrical. One classic example of a cylindrical map projection is the Mercator projection, commonly used for navigation purposes as it preserves local angles and shapes. Navigation charts classically use a Mercator projection because any straight line on this type of map is a line of constant true bearing, which allows a navigator to plot a straight-line course easily. However, this projection distorts the size of landmasses near the poles, leading to misconceptions about the relative sizes of continents and countries [10]. For example, Greenland appears much larger on a Mercator map than it truly is. Figure 2, shows that Greenland is actually about a third of the US in land area.

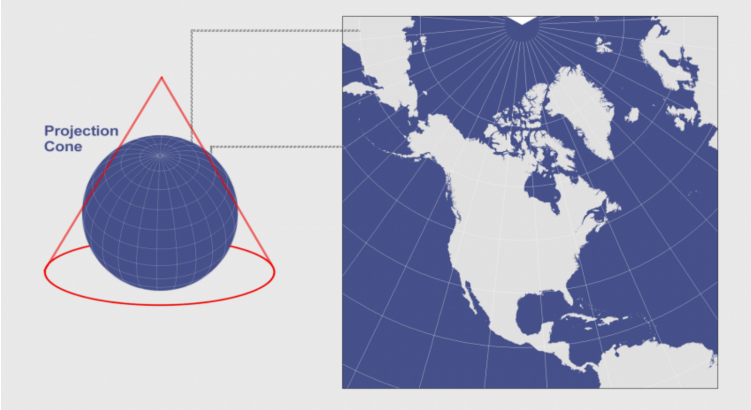


Figure 3: The Lambert Conformal Conic projection of North America [1]. This minimizes distortion within the chosen area but gets more distorted outside it.

Another type of map projection is the Lambert Conformal Conic projection, which is a conic map projection that maintains accurate shapes and angles over small areas [10]. This projection is mainly used for aeronautical navigation charts due to its angle preservation [10]. In aviation, accurate representation of angles is helpful for pilots to determine flight paths, headings, and orientation relative to landmarks, navigation aids, and other aircraft. On the other hand, with this map projection, distances are accurate only along the standard parallels. Scale, area, and distances are increasingly distorted away from the standard parallels [1]. In Figure 3, the Lambert Conformal Conic projection is centered on North America, showcasing its utility in mapping large regions with east-west extent while minimizing distortion within the chosen area but getting more distorted outside it.

2.5 Web Mercator

Web Mercator is a modified version of the Mercator projection designed specifically for web mapping applications. With this projection, the map of the world is fitted to a square shape. This adjustment helps minimize distortion when users zoom in or out on the map [12]. It also can divide up the entire world into tiles that are perfect squares and still have an equal number of latitude and longitude tiles at any zoom level. By maintaining relatively low distortion levels and consistent tiles, Web Mercator ensures that geographical features remain accurately represented regardless of the scale. This consistency smoothly scales large maps, making it particularly useful for mapping platforms like Google Maps, which extensively employ the Web Mercator projection to deliver seamless and accurate mapping experiences to users worldwide.

WMS servers use the Web Mercator projection to generate the map tiles that are passed to the tiling client to be displayed. Since geospatial datasets often have their own unique projections chosen for

optimal appearance or utility, a WMS server needs to efficiently convert between the dataset's projection and Web Mercator to provide accurate tiles to the client.

2.6 Spherical Mercator Tiles

Spherical Mercator is a type of cartographic projection used to represent the Earth's surface on 2D maps, preserving accurate shapes and angles. This projection is a variant of the Mercator projection, which transforms the Earth's spherical surface into a flat plane by stretching areas near the poles which distorts certain perspectives [10].

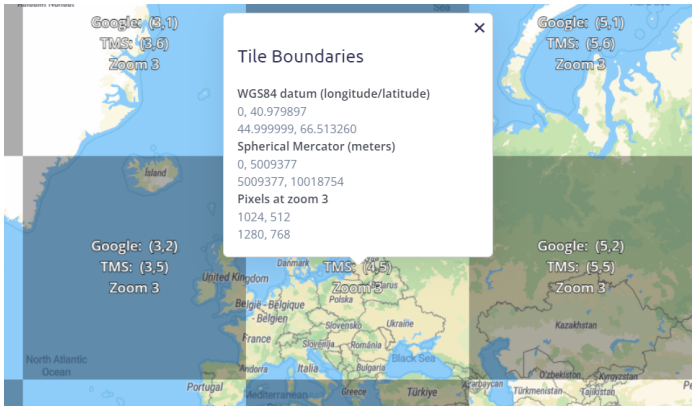


Figure 4: Spherical Mercator tiles are broken down into a grid. Each tile in the grid is represented by coordinates representing locations on a two-dimensional surface.

Spherical Mercator tiles break down the projected map into a grid of square tiles, each represented by a unique set of coordinates as shown in Figure 4. Popularized by web mapping services like Google Maps, OpenStreetMap, and others, this tiling approach enables efficient rendering of maps at various zoom levels [13]. It simplifies the process of displaying maps in web applications, allowing developers to fetch and display pre-rendered map sections, or tiles, based on user interactions such as panning and zooming. This is possible since the displays correspond to the same latitude and longitude every time [13]. Spherical Mercator tiles have become a standard in web mapping due to their compatibility with widely-used mapping APIs and their ability to provide a seamless and responsive user experience when navigating interactive online maps. OCEANSMAP requests map tiles using the Spherical Mercator tiling system, preserving the advantages outlined earlier. Additionally, Spherical Mercator tiles were requested randomly by the team's testing scripts to better mimic the types of tiles that a tiling client will usually request.

2.7 WMS - Web Mapping Service

Web Mapping Service (WMS) was developed by the Open Geospatial Consortium (OGC), WMS allows the integration of dynamic, interactive maps into web applications, providing a globally standardized way for users to access and display spatial data [14]. This allows clients to request map images from a server, which dynamically generates and delivers them in a standardized format. Key features of WMS include the ability to overlay multiple layers of spatial data, query specific geographic features, and customize the appearance of maps through parameterized requests [14]. The standard protocol has found applications in various sectors, including urban planning, environmental monitoring, disaster management, and navigation. OCEANSMAP employs WMS which projects the datasets into Web Mercator.

2.7.1 ncWMS

The Open Geospatial Consortium defines a standard for geospatial data to ensure everyone can run and understand the data openly. One of these specifications, known as the Web Map Service (WMS) is a standard for the rendering of this data. Data formats supported by this standard and utilized by Tetra Tech include Network Common Data Form (NetCDF) and Gridded Binary (GRIB) [15]. It is crucial for any system following this format to efficiently parse and render both data formats for a seamless user experience and to perform data analysis to generate needed maps.

The original performance goal of ncWMS was to generate one 256x256 tile in less than one second, regardless of the data set. The creators of ncWMS have also identified common slowdowns of the current software, being either bound by the server's processor or storage. In the study conducted to measure the efficiency of the server, they attempted to utilize different ways to read the geospatial data, as seen in Figure 5 [15].

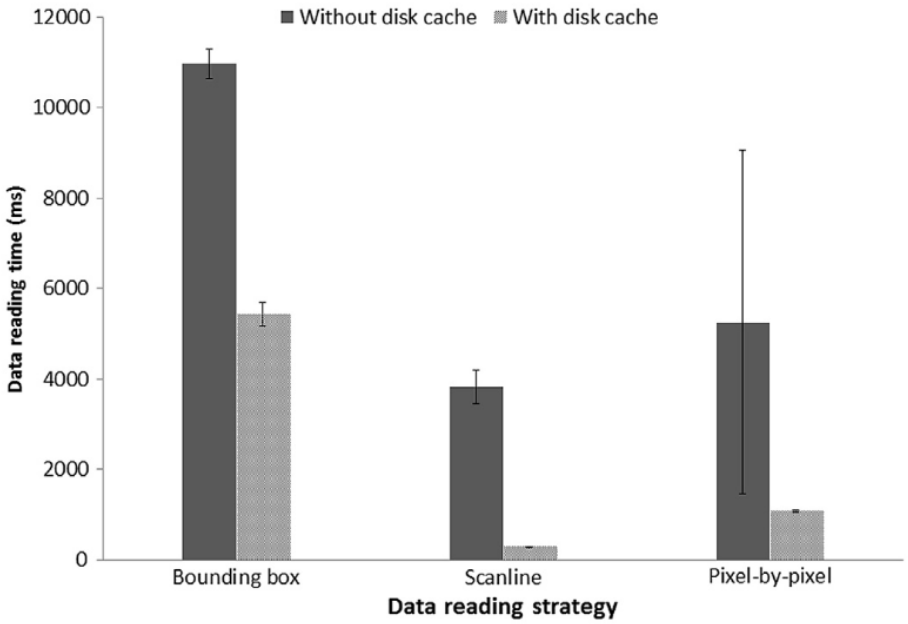


Figure 5: Different measures for reading geospatial data. With each reading strategy, the time (ms) to read the data varies, with and without disk caching.

While the method for reading the geospatial data is evidently important from an efficiency standpoint, this graph also shows how useful caching is when reading the dataset. In ncWMS, the caching implementation only stores the raw geospatial data, not the rendered tiles. By default, the map in the Godiva3 interface requires nine tiles arranged in a 3x3 arrangement to create a complete map view. For today's standards, and the increased amount of data in modern datasets, a more efficient method to generate these tiles is required, as this solution has become too slow for a comfortable and reliable user experience.

2.7.2 Xpublish-wms

Xpublish is a tool that allows for the sharing and publication of Xarray datasets through a web application. Tetra Tech developed a version of this called Xpushlish-wms. Xpublish-wms, as defined in its name, is a WMS (Web Map Service) router for Xpublish. It is designed to work with the OGC (Open Geospatial Consortium) WMS API. This plugin is developed and maintained by the Xpublish community via GitHub. Xpublish, as mentioned, is a tool that facilitates the sharing and publication of Xarray Datasets through a user-friendly web application and intuitive user commands and/or interfaces. Xarray is a Python package that provides an in-memory data model and analytical tools for multi-dimensional arrays. This makes Xpublish particularly useful for dealing with multi-dimensional arrays found in the GRIB and NetCDF file formats.

Once Xpublish-wms is installed, the plugin registers itself with Xpublish, and WMS endpoints are included for each dataset on the server. This makes it a valuable tool for managing, viewing, analyzing, reading, and distributing gridded data. Xpublish-wms can efficiently manage and distribute large volumes of gridded data such as meteorological and oceanographic forecast agencies, where oceanographic datasets are the focus of this project. Xpublish-wms functions with a subset of Xarray datasets out of the box. To be compatible, a dataset must have CF (Climate and Forecast) compliant latitude and longitude coordinates. It supports either latitude and longitude dimensions that correspond to the CF-compliant coordinates or CF-compliant SGRID (Structured Grid) metadata, strictly in the form of properly spaced latitude/longitude and structured grids.

One of the key features of Xpublish is its server-side integration with Dask. Dask is a flexible library for parallel computing in Python that is designed to integrate with the broader Python ecosystem. It enables efficient, on-demand delivery of large datasets such as those used in server requests throughout this project. This means that data can be loaded and processed as needed, rather than all at once, which can significantly improve performance when working with large datasets.

In summary, Xpublish-wms is a powerful tool for managing and distributing gridded data. Its compatibility with the OGC WMS API and integration with Xpublish make it a valuable resource for anyone working with gridded data. Its support for CF-compliant coordinates and SGRID metadata further enhances its utility and relevance with WMS and is relevant to the computation of large data arrays used.

2.7.3 Other WMS Solutions

Aside from the WMS solutions described above in more depth, there are multiple other useable and supportive WMS solutions available. One such application is ArcGIS Server, which allows users to publish a WMS service by enabling the WMS capability when publishing a map or image service (Or input gridded file). This makes maps available online in an open, recognized way across different platforms and clients. Another method involves using ArcPy, a Python site package that makes it possible to perform geographic data analysis, data conversion, data management, and map automation with Python. This package can be used to publish and overwrite a service that has the WMS capability enabled. These applications provide robust solutions for handling common data formats in a WMS context in addition to ncWMS, Publish-wms, and Xarray. GeoServer is another powerful open-source server that allows users to share, process, and edit geospatial data. Designed for interoperability, it publishes data from any major spatial data source using open standards. These tools, along with ArcGIS Server and ArcPy, offer a wide range of options for working with WMS applications, excluding XPublish-wms, ncWMS, and Xarray.

2.8 Tiling Clients

To further utilize WMS server capabilities, users can connect the WMS endpoints to tiling clients, a tool that will request necessary tiles and stitch them together to create a map. Two tiling clients used during this project include Godiva3 and Xreds Viewer. Both of these clients connect to WMS endpoints, allowing the user to choose layers from datasets that are added to the server. When a user selects one of these data layers, the client will then request the tiles, placing them on the tiling client's canvas [13]. These tiling clients may have extra features such as adding a base map below the tiles. These base maps can vary depending on the client's usage, including satellite imagery or a map of country borders, etc. The flexibility of WMS allows the clients to be easily customized depending on the use case (ex. OCEANSMAP).

2.9 XArray

In essence, XArray is simply a Python library that makes working with multidimensional arrays much easier. Multidimensional arrays are an essential data type that is utilized in countless fields, including the geospatial sciences. Although the standard NumPy library does provide support for multidimensional arrays, it does not support labels on array rows and columns, making working with the data much more tedious and time-consuming, particularly when referencing specific elements or subsets of the array [16]. By employing XArray, row and column labels are fully supported and can be used to simplify working with multidimensional arrays. Some of the label abilities introduced by XArray include selecting rows/columns by label instead of number, mathematical operations across multiple dimensions using their labels, grouping by functionality, database-like alignment/joins, and metadata/attributes attached to labels.

Even more useful for the project is the fact that XArray is built to function directly with the NetCDF data format (by using the same data model), which is likely the most widespread data format used in geosciences and a data format that is essential to the work of this project. This added functionality means that working with NetCDF data only requires XArray and an additional library to support reading/writing to NetCDF files, making XArray the natural choice for working with geospatial data in Python [16].

2.10 Docker

Docker is an open platform designed for developing, shipping, and running applications efficiently. Docker was used in the project to run the ncWMS premade Docker image. Docker utilizes containers, which are loosely isolated environments, to package and run applications. These containers ensure consistency

across different environments, enabling developers to work in standardized settings. Docker coordinates the entire lifecycle of containers, from development and testing to deployment in various environments, whether local, cloud-based, or hybrid. It promotes fast and consistent application delivery through continuous integration and continuous delivery (CI/CD) workflows [17]. The Docker Engine serves as the core, managing the creation and operation of containers through a command-line interface and Docker Desktop. Docker images, read-only templates containing application code and dependencies, are used to instantiate containers. Docker's flexibility allows for responsive deployment and scaling, making it easy to manage workloads dynamically.

2.11 Linux

Linux is an open-source operating system kernel that serves as the foundation for various Linux-based operating systems. Known for its stability, security, and flexibility, Linux is widely used in server environments, embedded systems, and as an alternative desktop operating system [18]. Because many of the tools used during the project were designed for use on Linux, using Linux for both the team's testing server (which used Ubuntu for its operating system) and some of the development work became an important part of the project. Originally, Docker was designed to be used with Linux, so the easiest way to set up the ncWMS Docker image was on a Linux system. Additionally, rendering GRIB2 files requires ecCodes, which are only officially supported on Linux and macOS. Therefore, opting for Linux was the logical choice to ensure compatibility and efficiency across all project tasks.

2.12 Web Requests

A web request is a request made by a client, such as a web browser, to a server to retrieve a web page or other resource. Web requests are sent using the Hypertext Transfer Protocol (HTTP), which is a standard protocol for transmitting data on the World Wide Web [19]. This protocol defines the rules for communication between the user's browser and the web server, ensuring the smooth exchange of information. In a client-server architecture, the client requests content and the server is the provider of requested resources or information. This separation of responsibilities streamlines data transfer and processing. Web browser requests come in various types, such as GET requests, which are used for retrieving data [19]. In essence, web browser requests are important in retrieving and displaying web content, forming the backbone of the interconnected online experience.

2.12.1 AIOHTTP vs Python requests

AIOHTTP and Python requests are libraries that serve different purposes and operate on distinct principles. "Requests" is a synchronous HTTP library widely used for making HTTP requests in a straightforward and blocking manner. It sends messages to the server and waits for the server to respond before continuing [20]. When a request is sent using the requests library, code execution on the client is paused until the server returns a response. Requests are simple to use and suitable for many applications, however, it may not be optimal for handling a large number of concurrent requests efficiently because the requests are synchronous.

On the other hand, AIOHTTP is an asynchronous HTTP client/server framework specifically designed for asynchronous programming using Python's "asyncio" module. Asynchronous programming continually delivers updated application data to users, freeing up the executive thread to perform more functions. [20]. This asynchronous nature makes AIOHTTP well-suited for scenarios where responsiveness and scalability are prioritized, such as handling numerous concurrent connections. Within this project, the team tested sending tile requests synchronously (for small numbers of concurrent requests) and asynchronously (for large numbers of requests sent all at once) to see a difference in the request time as the number of requests increases.

2.13 Caching Solutions

Caching solutions are essential for enhancing the performance of applications by storing a subset of data in a high-speed storage layer or command, reducing the need to access underlying slower storage layers. Amazon ElastiCache is a notable example, providing an in-memory data store and cache in the cloud, supporting cloud caching solutions such as Redis. It improves the performance of web applications by allowing the retrieval of information from fast, managed, in-memory data stores. These solutions provide efficient and scalable caching mechanisms for file requests, removing the necessity for multiple redundant rerendering of web tiles. In this project, Redis, LRU Cache, and the Memoization Python library were leveraged as effective strategies and solutions for caching.

2.13.1 Least Recently Used (LRU) Cache

The LRU caching algorithm is a key algorithm extensively applied in this project as a highly efficient caching strategy. LRU operates by removing the least recently used item in a specifically full cache when a

new data request is issued [21]. The caching frame is then updated, and the new tile is detached from where it could be (like memory) and is placed in the newly emptied frame in the cache [21]. By keeping the most recently used data in the cache, the LRU algorithm significantly reduces the time taken to access frequently used data, which increases the data delivery overall. Additionally, the LRU algorithm is straightforward to implement and manage, making it a popular choice for caching in various applications.

In this project, LRU cache played a crucial role in efficiently managing memory resources with a cost reduction benefit for the aforementioned redundancy in tile rerendering and reprojecting. The LRU cache, combined with the time-to-live support and flexibility offered by the Memoization library, provided a robust caching solution for the team's optimizations [22].

2.13.2 Memoization Library

The Memoization Python library is a robust caching solution that the team utilized in this project. The practice of "memoizing" is an effective strategy the team applied to more efficiently compute and cache projected data arrays while handling large amounts of data. Based on the preexisting 'Functools' library, this library offers a variety of features including configurable cache size, thread safety, time-to-live support, and cache statistics [23]. Regarding its functionality, a memoized function caches the results of a set of input requests. It virtually cancels the computation cost (memory and speed) of a function's running time [23]. Memoization supports multiple caching algorithms such as Least Recently Used (LRU), Least Frequently Used (LFU), and First In First Out (FIFO), all of which can be used to avoid redundancies in computing [24]. Implementing memoization was advantageous because the library assisted with content that the standalone LRU cache algorithms could not support [23]. Furthermore, memoization has a preexisting built-in implementation of LRU cache, supports unhashable arguments, and allows for custom and order-independent cache keys. Memoization offers flexibility in caching different types of data, which is a strategy also utilized in the application of Redis.

2.13.3 Redis

Redis is another powerful caching solution utilized in this project. Redis provides performance enhancements that properly mesh with this project's use cases as an in-memory data store. It is primarily designed for high-performance, low-latency data access [25]. Redis is often used as a cache for web applications, where it can store frequently accessed data such as user sessions, website content, and API responses. The implementation of Redis is relevant due to the web mapping service's computationally expensive API

calls. Its in-memory data storage, flexibility, and support for fast operations make it an ideal and integral caching solution used in the teams' strategies.

Applying Redis server-side in context is utilizing memory to efficiently store tiles as they render. Redis capitalizes using unallocated memory as a tile storage system, where frequently and recently used tiles are placed in the cache. The stored rendered tiles are then positioned for rendering cost elimination for repeated tile requests. In summary, Redis plays a crucial role in the teams' speed improvements in tile retrieval operations, saving tiles it has already rendered for easier user access.

3 Testing Methods

3.1 Creating the Testing Environment

Before the team began testing and analyzing the speed of the WMS systems, the team had to figure out a way to consistently and reliably host the systems while they were being benchmarked. A key obstacle that the team had to overcome early in the testing phase of the project was that each team member was using a different system for project work, and some of these systems even varied on a day-to-day basis. Without a constant testing environment, the benchmarking results cannot be directly compared, and it would not be possible to draw long-term effective conclusions. To remove the variability introduced by differing testing environments, a shared testing server was created and configured for the sole purpose of running the WMS servers during the project. Having a single dedicated server for testing ensured that all tests were run on the same hardware and operating system, and eliminated many other unwanted variables during testing. However, there were some drawbacks to using a custom server that needed to be accounted for during the server's setup process. One important consideration was that the server would be in a private space instead of a shared location, meaning only a single team member could physically access the server. This limitation on direct server access meant that the server must be reliable for the entire team even when that team member cannot physically access the machine, therefore contingencies like power outages and server crash recovery needed to be considered. The final server specifications are as follows:

Processor	i7-3770k
RAM	16 GB RAM DDR3
Storage	1x HDD 7200 RPM
Operating System	Ubuntu 22.04 LTS

Table 1: Server specifications

Since the team started "from scratch" with the testing server, the first step for server configuration was installing an operating system and configuring it for reliability. Because Linux is generally the best choice for working with geospatial data formats (particularly GRIB), the operating system of choice for the testing server was Ubuntu 22.04 LTS. Additionally, the system was configured to automatically restart in the BIOS options to ensure it would recover from any potential loss of power. Once Ubuntu was installed and configured for a server environment (automatic login on startup), Docker was installed and configured. An important note is that the Docker service was configured to run automatically on startup to ensure that

the WMS system could fully restart without any user intervention.

For easy ncWMS setup on the Ubuntu server, the team used the premade Docker image of ncWMS. The containerization provided by Docker allowed the team to quickly add datasets, control resource usage, and keep the software virtualized. Once this Docker image was configured, the server was ready to test with ncWMS.

After configuring ncWMS, the next step was to configure Xpublish-wms, the other WMS server that the team would be testing. Although the setup itself was much more straightforward than ncWMS, Xpublish-wms requires the "ecCodes" library to read GRIB files, which was a difficulty to overcome in the project. The ecCodes package provided for Ubuntu 22.04 only goes up to version 2.24.2, whereas the earliest version of ecCodes recommended for Xpublish-wms is version 2.31.0. This meant that it was necessary to compile ecCodes from source on the testing server. After this installation, finalizing the Xpublish-wms setup on the testing server consisted of configuring a Python virtual environment, installing the needed dependencies, and opening/serving the dataset(s) to be tested. In the end, the final server used these relevant software versions:

ncWMS	v2.5.2
Xpublish-wms	V0.4.0 (and prototype optimized versions)
Xpublish	v0.3.3
ecCodes	v2.32.1

Table 2: Software versions of the project's tech stack.

3.2 Testing WMS Tiling Endpoints

Before optimizing the WMS plugin for Xpublish, the team needed a way to benchmark and measure the current performance of the system. To achieve this, the team wrote a Python script to request tiles and measure the response time. The results of the tests were output into comma-separated values (CSV) to be easily reviewed and imported into Microsoft Excel sheets for further analysis. The script has many modifiable settings to test the server in different scenarios.

The script was created with the ability to be run in three different modes, which each manipulate how the requests are sent to the server. The first mode is single-threaded, which will only send one tile request at a time. The script will wait until the first request is complete before sending the next request. The second mode is multi-threaded, which sends the request using a set number of threads by the user. Each

thread will send the same amount of requests concurrently and will send the next request once the previous request in its respective thread is complete, ignoring the status of other threads. The third mode sends the requests asynchronously. The script will send out the requests, ignoring the status of all other requests, and then record the time it took for each request to be fulfilled. This is the most important mode of all three as modern web browsers send requests asynchronously, so this will closely mimic a user experience on a tiling client such as Godiva3.

Regardless of which mode the script is running in, the requests being sent can be modified to test different variables and their effect on performance. For example, the resolution of the tile can be adjusted in multiple runs of the script. By default, the tiles are 256x256 pixels, and the Godiva3 map utilizes nine tiles in a three-by-three arrangement to form the full map. Additionally, the file type of each tile can be modified, such as a PNG (A lossless image data format) or a JPEG (A lossy image data format).

One of the most important pieces of the query sent to the server is the bounding box. The bounding box is a rectangle that describes what location the tile should pertain to. A bounding box is created using four parameters: the westernmost point, the southernmost point, the easternmost point, and the northernmost point. To quickly generate the tiles for testing, the team used the Python package 'Mercantile,' which simplifies the bounding box to a simple three parameters by using Spherical Mercator tiles. The new parameters were an X tile index, a Y tile index coordinate, and a Z index which represented the zoom level for the tile. Spherical Mercator tiles allow for easy scaling when the zoom is manipulated, as each time the zoom level is increased by one, the range of X and Y tiles doubles for the entire world. As a result, the zoom level can be manipulated in the script easily, which can increase or decrease the amount of data required to generate a tile. It should be noted that a tile should not be infinitely zoomed in to, as different geospatial datasets will have different resolutions, such as one dataset having data for a region at a resolution for each square kilometer, and another having a resolution of a square meter.

Figure 6: Demonstrating a conversion from Spherical Mercator Conversion to latitude longitude bounding box. This was an important step for being able to quickly generate tiles for testing.

Additionally, when testing the ncWMS server, the team ran the tests twice; once with caching enabled, and once with it disabled. Xpublish-wms does not have this as a toggle-able feature. This will allow the team to determine the importance of caching in the ncWMS solution when sending a tile, and assist in determining if improving the caching in Xpublish-wms should be a primary focus for improvement.

3.3 Testing Results

To investigate the structure of the ncWMS server and the capacity it can handle, a Python script was developed to generate random tiles at runtime, allowing for adjustment of the number of threads, tile resolution, and zoom level range. As depicted in Figure 7, these global values can be easily manipulated.

```
# globals for config file values initialized to placeholder/empty values
multi_threading = False
threads_per_layer = 1
hostname = ""
filename = ""
tilewidth = 256
tileheight = 256
minzoom = 4
maxzoom = 8
num_trials = 1
layers = []
epsg = 3857
limit_x = False
limit_y = False
min_x = -1
min_y = -1
max_x = -1
max_y = -1
```

Figure 7: In the configuration file, global variables can easily be changed for different tests.

This script was used to test the total time and average time it takes for a request to take place. Three types of tests were conducted with no caching. The following equation calculated the total trials: (thread_per_layer X 4) X num_of_trials = 1200 total trials, 600 trials per dataset. In each test, the total time, average time (ms), lower quartile (ms), upper quartile (ms), lower 1% (ms), and upper 1% (ms) for both GRIB2 and nc data were recorded. For the first test with no caching, the tile size was changed to 64x64, 128x128, 256x256, and 512x512 while the zoom levels were kept constant. As shown in Figure 8, there was little to no change identified when increasing the tile size.

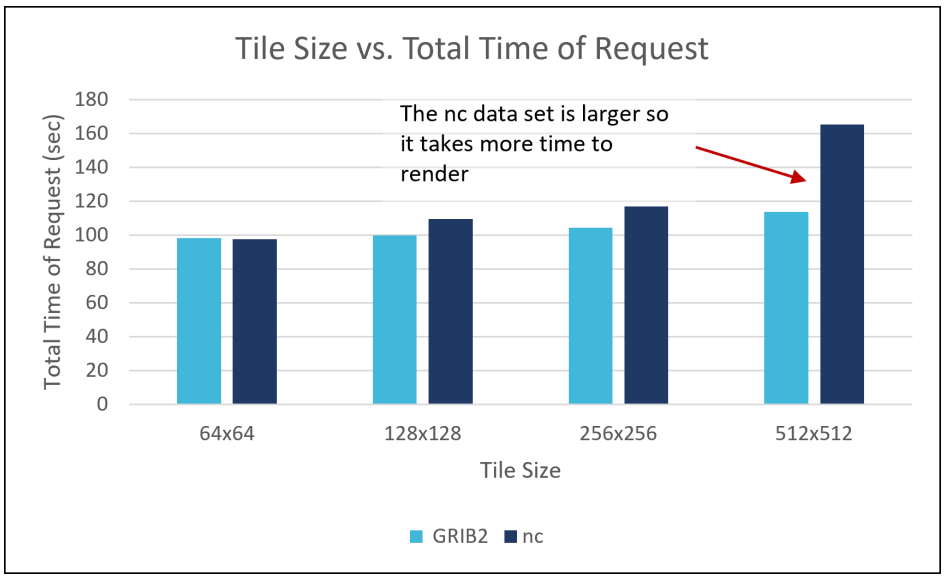


Figure 8: Almost no change in time of request when increasing the tile size and keeping zoom constant. This is not the main cause of the slowdown.

This meant that having high-resolution tiles but low-resolution data is not where the issue of the slowdown in software lies. It could also be observed that the NetCDF data slows down more than the GRIB2 data but this is because it is a larger dataset and more difficult to render. For the second test, the number of threads was changed by increments of four. In Figure 9 and Figure 10, it is observed that the concurrent execution of multi-threaded requests exhibited a reduction in the total time required to fulfill 1200 requests.

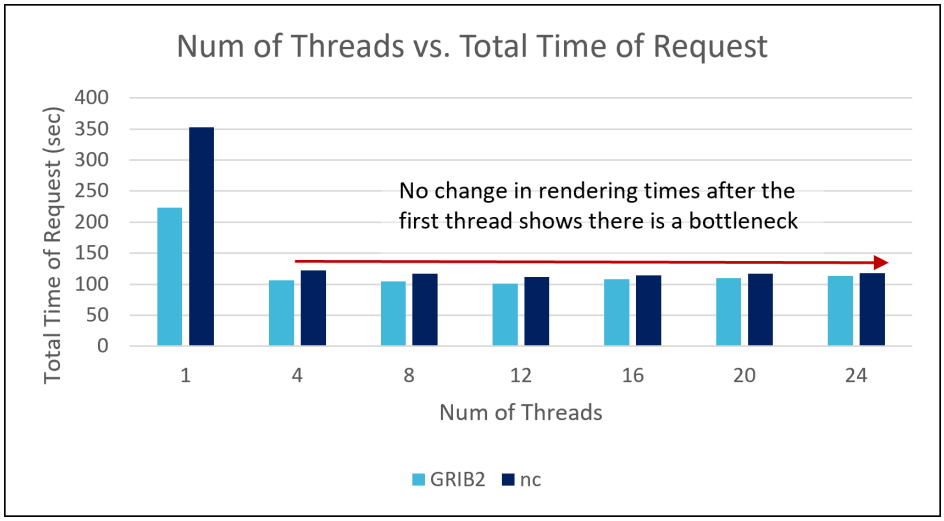


Figure 9: With concurrent requests, the render time plateaus limit opportunities to enhance efficiency through parallel processing.

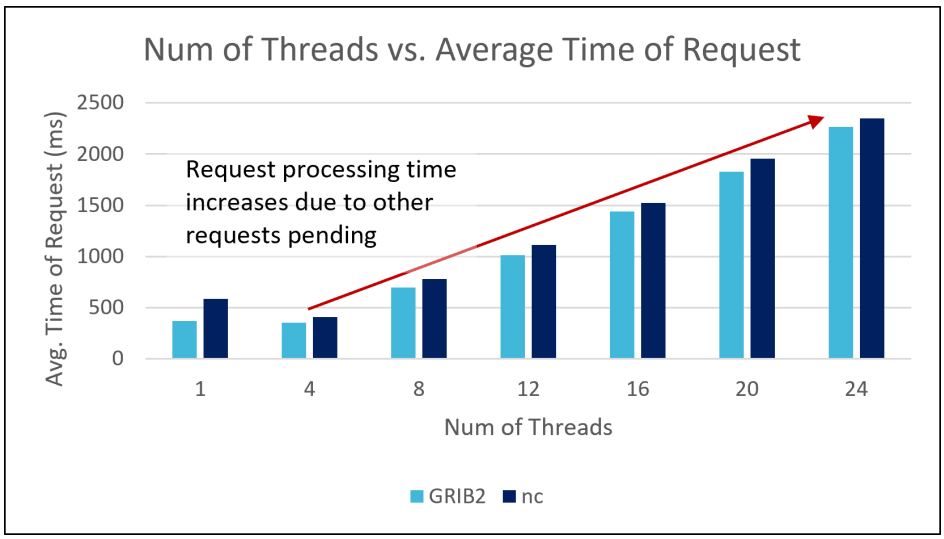


Figure 10: The request processing time increases due to a larger amount of requests pending in the queue. The gain in the number of threads tested shows a significant increase in average runtime for both datasets.

This reduction was particularly significant in the case of GRIB2 data, where the time was halved, and even more so in the case of nc data, where it was reduced to one-third of the original time. However, it is noteworthy that the total time plateaus when the number of concurrent threads exceeds four. This observation indicates the presence of a bottleneck, which hinders further parallel processing efficiency. Also, as the thread count escalates, the individual request's processing time increases significantly, due to the need to await its turn in a queue of pending requests. For the third test, the min and max of the zoom level were changed along with the tile size. Starting with a minimum zoom level of four and a maximum zoom level of eight and decreasing by one for each trial. These ratios were chosen to maintain the pixel size of the map tile image. In this testing scenario, the team would expect the server to spend more time on the higher-resolution tiles compared to the first-resolution experiment. This is because as zoom levels decrease, the resolution becomes coarser, encompassing more data within each tile. Therefore, this trial has more data in the larger tiles than in the first trial where only the tile size was changed and not the zoom.

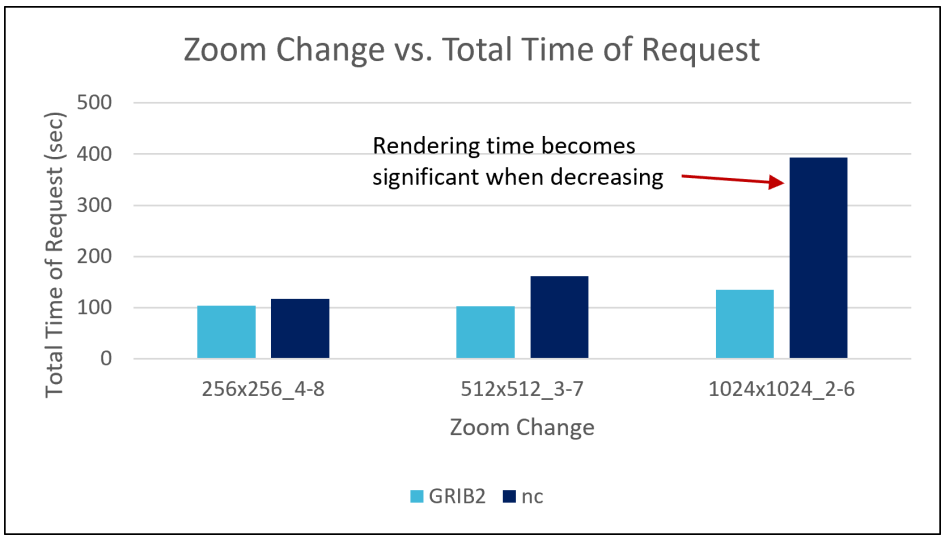


Figure 11: This graph shows how the tile rendering times are affected by larger tiles in a more realistic scenario. In a real tiling client, making individual tiles larger will increase the amount of data rendered on each tile, which increases the time spent rendering each tile in this case.

In Figure 11, when the resolution is doubled, the rendering times exhibit a delay ranging from 2 to 3 times greater. This observation underscores the efficiency advantages of employing larger tiles, exemplified by a single 1024x1024 tile outperforming the efficiency of rendering 16 individual 256x256 tiles.

Within ncWMS, there is a feature where one can easily enable or disable caching. In order to see if there are any changes in rendering times with caching (enabled through ncWMS), the team decided to start by repeating the first test.

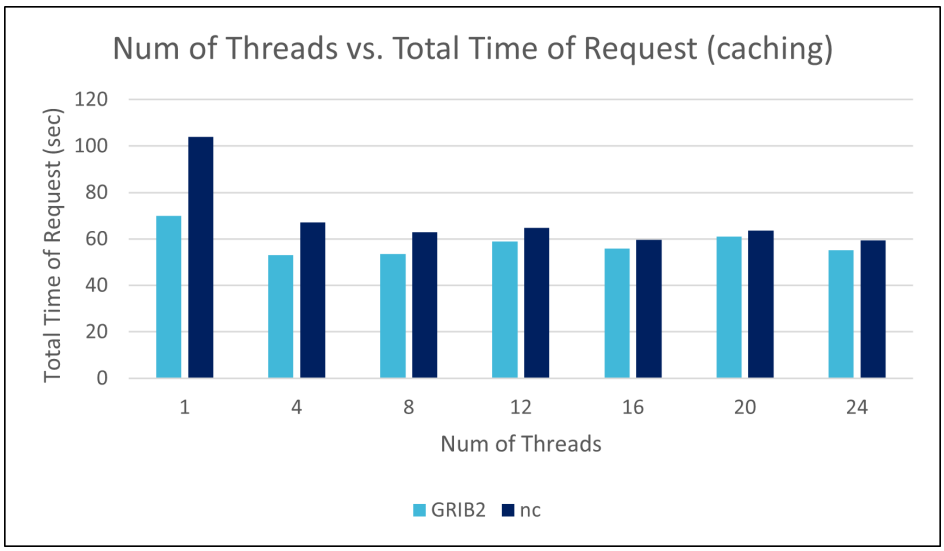


Figure 12: Total request time is reduced with caching in comparison to without caching.

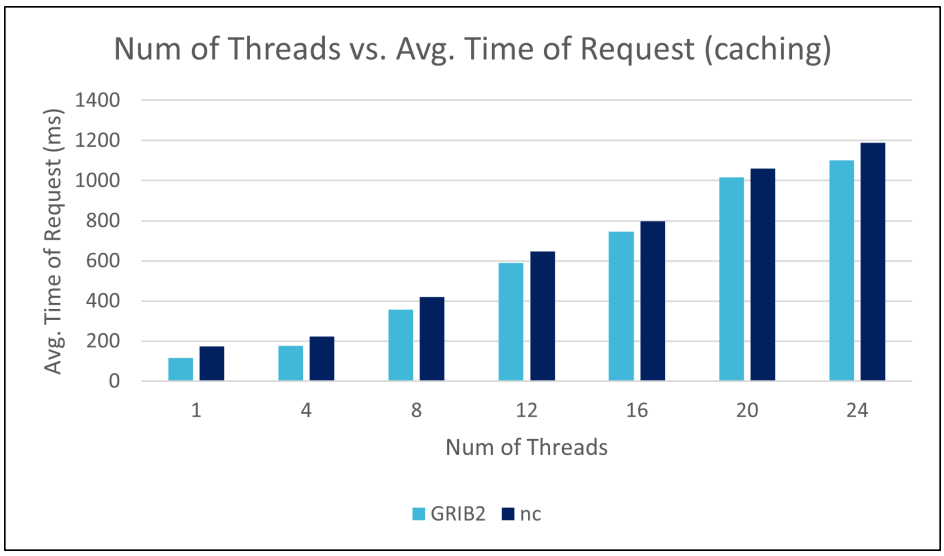


Figure 13: The average time of request is overall still faster with caching than without caching.

Figures 12 and 13, in comparison to Figures 9 and 10, showed a 30% improvement in response times with no caching. This makes sense because caching allows the server to store and retrieve previously processed requests, reducing the need for redundant computations. Due to this, the team concluded that repeating the same caching tests was unnecessary since the outcomes would have remained consistent throughout.

For further experimentation, an asynchronous request testing script was created, avoiding the use of threads. The investigation involved sending requests at the maximum feasible rate. The server exhibited resilience against an influx of thousands of concurrent requests, avoiding outright crashes. However, an observed consequence was the accumulation of requests in an extensive queue, leading to timeouts for subsequent requests. This highlights that exceeding the server's processing capacity, roughly one tile every 200 milliseconds, leads to a significant accumulation of requests, eventually causing timeouts.

In order to understand the rendering times of Xpublish, the team conducted tests on the rendering times of various models using the same number of tile requests as the previous ncWMS testing.

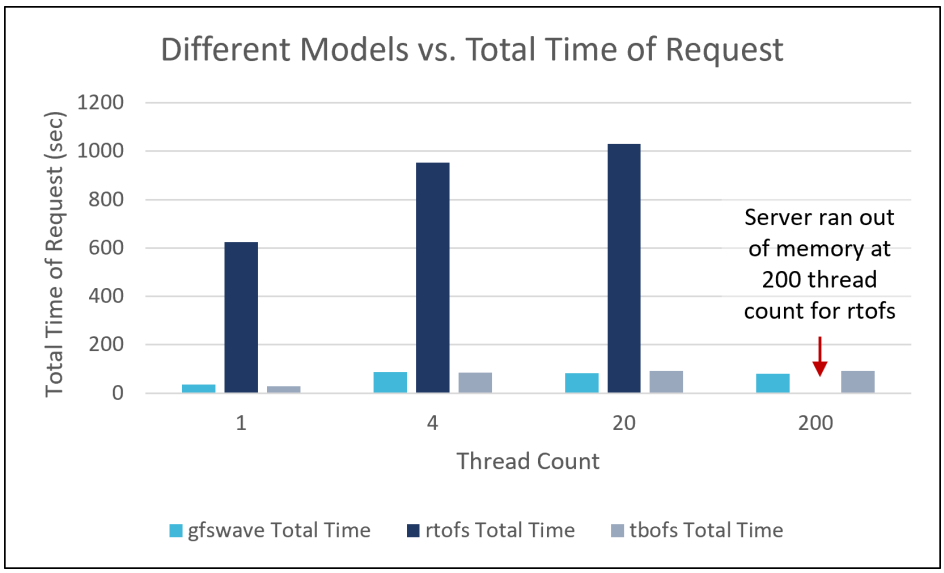


Figure 14: The models take a long time to render tiles. There is potential for optimization in this area.

As shown in Figure 14, rtofs took the most time to render, being around 1000 seconds. The gfswave (Global Forecast System - Wave) model and tbofs (Tampa Bay Operational Forecast System) model took about 100 seconds to render. These areas are where the team saw potential for improvement. The identified observations spurred the team to explore avenues for optimization, leading to a focused investigation into the slowest aspects of the rendering process.

4 Optimization Methods

4.1 Identifying Areas of Slowdown

The team began researching how to optimize Xpublish-wms by first investigating the slowest parts of the tile rendering process. During each map tile request, Xpublish-wms follows a relatively constant formula to create the map tile regardless of which dataset (and corresponding grid) is requested by the client. The basic process is as follows:

1. Select the correct data from the dataset based on the query provided by the request URL.
2. Perform any necessary fixes/modifications to the dataset based on the specific grid that it uses. This part of the process is trivial for some grids but is very significant for others.
3. Project the selected and fixed dataset into the Web Mercator projection so that the tile looks correct.
4. Render/shade the map tile based on the query parameters and the projected dataset.

The obvious place to start was to place timers on each part of this process to understand which parts of a GetMap request were computationally cheap, and which parts were expensive and took up a lot of processing time during the request. After isolating each section described above with timers and performing some basic performance tests with different grids, the team had a rough idea of where the optimization efforts needed to be focused to get the best results. In particular, the data selection portion tended to always be fast, and the rendering/shading of the tile tended to be the most costly part of the process for simple datasets. Because projection times varied greatly from dataset to dataset, ranging between negligible to extremely slow, the team decided that the best place to start would be to investigate why certain grids(notably HYCOM Grid) were so slow during steps two and three of the above process.

4.2 Early Failed Optimization Attempts

Before making any meaningful, useful optimizations, the team experimented with a few approaches that ended up failing in the end. These attempts are described here as both likely have potential in the future but are much more complex to implement than they may seem at first glance.

4.2.1 Manipulate bounding box, not data array points

In certain grids, the latitude and longitude values in the dataset itself do not directly correspond to the latitude/longitude values that are passed in for the bounding box of the WMS tile query. For example, the HYCOM Grid uses longitude values from 74-430 [26]. As a result, Xpublish-wms has to adjust these values to the -180 longitude values that the Web Mercator projection uses. Since the rtofs dataset (the major relevant dataset that uses HYCOM) is very large and complex, this operation is computationally costly, and performing this normalization on the entire dataset for each and every tile request (as Xpublish-wms was doing) is a considerable source of slowdown.

The team's first idea in sorting out this issue was to attempt to manipulate the bounding box passed in by the client to match the dataset's coordinate system, instead of conforming the entire dataset to the Web Mercator convention beforehand. After considerable effort and frustrations, the team decided that this approach is way more difficult than it sounds and is likely more effort than it is worth for a few different reasons. At first, it seemed like all that needed to be done was to normalize the input coordinates between the longitude values present in the dataset. Although the team was able to get some tiles functioning with this type of conversion, they were never in the correct place. After some time, the team realized that this issue was because the Web Mercator grid and the HYCOM grid were offset from each other, meaning that -180 (the lowest longitude value) in the Web Mercator grid was not in the same place in the world as 74 (the lowest longitude value) in the HYCOM grid. After compensating for this offset by either subtracting or adding based on the specific tile being rendered, the team had a bit more success but was never able to get a single tile of the whole world map to return correctly. At this point, the real issue with this approach revealed itself.

To gain a clear understanding, let's first define what a bounding box represents. The bounding box defines the left edge, bottom, right edge, and top of the area that the tile will encompass. Since longitude increases from left to right, the bounding box's left coordinate should always be less than the right coordinate for the tile to render correctly (and not be stretched/mirrored/distorted). Now imagine that, given a regular Web Mercator projected dataset (-180 to 180 longitude values), the client requested a tile with a left edge at 170 and a right edge at -170. To a person, this request makes sense, as the tile should go right and "wrap around" the right edge of the map to the left to finish. However, the bounding box will be interpreted as going left on the map from 170 to -170, meaning that 340 degrees of the world will be mirrored and packed into the single tile instead of 20 degrees going right as the user wanted.

The result here is that tiles that "wrap around" are impossible without morphing the dataset to

accommodate those requests, and this is exactly why converting just the request bounding box does not universally work with the HYCOM dataset. The right edge of the Web Mercator system and the HYCOM grid system are in different places, and the client will unknowingly try to request tiles that "wrap-around" in the HYCOM grid, and as a result, these tiles do not work correctly. This is exactly the reason why some tiles were working perfectly while others were getting destroyed through this method of manipulating the bounding box. The only obvious way to remedy this is what Xpublish-wms currently does, normalizing the data itself to the coordinates that are expected for a Web Mercator projection. The team did consider that it is probably possible to split these requests that wrap around into two separate tiles, render them separately, and then combine them to send back as a single tile, but this approach would almost certainly lead to worse performance results than the current method of normalizing the dataset.

4.2.2 Convert bounding box to Lat/Long instead of data array points

Another idea the team explored but failed to implement was converting a bounding box that was specified in web Mercator coordinates into latitude/longitude coordinates. Currently, Xpublish-wms supports bounding box requests in both EPSG:3857 (Spherical Mercator), as well as EPSG:4326 (WMS84), coordinates. If the coordinates are provided in EPSG:3857 in the request, for most grids the dataset must be converted into this coordinate system after projecting it in EPSG:4326 first. Instead of going through this conversion process, the team attempted to convert the request bounding box to EPSG:4326 in all cases to avoid this extra computation on the whole dataset. This never ended up working for unknown reasons, and the rendered tiles resulted in extreme distortion close to the north and south poles. The team suspects that this is still a viable approach, and should provide a performance benefit on all grids if implemented successfully.

4.3 Optimizing HYCOM Grid

HYCOM Grid refers to the Hybrid Coordinate Ocean Model, a sophisticated numerical ocean model utilized by scientists and researchers to simulate oceanic processes and phenomena. This grid system employs a hybrid blend of vertical coordinate systems, allowing for a more accurate representation of complex ocean dynamics across various depths [27].

HYCOM Grid has a few dataset quirks that need to be worked out to successfully render a map tile, which end up significantly contributing to the overall time that a request takes if these issues are dealt with on a per-tile basis (as Xpublish-wms was doing before any optimizations took place). The first quirk

is that there are “garbage” longitude values (greater than 500) in the dataset that are unused by the model and should be ignored during tile generation. The second quirk is that the grid is rectilinear up to 47 degrees north and curvilinear above this latitude, making projecting and rendering the grid much more complex. When this complex grid is combined with the very large size of the global rtofs dataset, tile rendering times become very, very slow, as seen in the testing results. Finally, as explained earlier, because longitude ranges from 74 to 433 in the dataset and does not align with EPSG:4326, the dataset needs to be normalized around -180 to 180 longitude values. All of these quirks together make for a very complex dataset to work with that takes quite a long time to render even in an optimized state.

4.3.1 Removing mask

Xpublish-wms originally tried to search through the data to find the longitude values greater than 500 by using a mask on the whole dataset. In lines 552-558, a mask is calculated based on longitude values greater than 500, to handle data from the Real-Time Ocean Forecast System (rtofs-Global). The function then applies this mask to the input data array, effectively masking out values where the longitude exceeds 500 to remove “junk values”. Although this approach works, it is extremely slow on a dataset as large as rtofs, taking nearly a full second just to compute this mask alone on the team’s testing server.

```
537     ) -> Union[xr.DataArray, xr.Dataset]:
538         # mask values where the longitude is >500 bc of RTOFS-Global https://oceanpython.org/2012/11/29/global-rtofs-real-time-ocean-forecast-system/
539         lng = da.cf["longitude"]
540
541         # add missing dimensions to lng mask (ie. elevation & time)
542         if len(da.dims) > len(lng.dims):
543             missing_dims = list(set(da.dims).symmetric_difference(lng.dims))
544             missing_dims.reverse()
545
546             for dim in missing_dims:
547                 lng = lng.expand_dims({dim: da[dim]})
548
549         mask = lng > 500
550
551         np_mask = np.ma.masked_where(mask == 1, da.values)[:]
552         np_mask[:-1, :] = np.ma.masked_where(mask[1:, :] == 1, np_mask[:-1, :])[:]
553         np_mask[:, :-1] = np.ma.masked_where(mask[:, 1:] == 1, np_mask[:, :-1])[:]
554         np_mask[1:, :] = np.ma.masked_where(mask[:-1, :] == 1, np_mask[1:, :])[:]
555         np_mask[:, 1:] = np.ma.masked_where(mask[:, :-1] == 1, np_mask[:, 1:])[:]
556
557         return da.where(np_mask.mask == 0)
```

Figure 15: Original code using masks to get rid of junk values. It is computationally expensive.

Through the NOAA presentation “Reading Scientific Datasets in Python,” the team found a much better way to filter out these junk values. According to NOAA, there is “a quirk in the Global rtofs datasets – the bottom row of the longitude field is unused by the model and is filled with junk data” [26]. So instead of calling the mask function, the last row of the longitude array was dropped entirely, since this row contains only the junk longitude greater than 500 values:

```python
def project(self, da: xr.DataArray, crs: str) -> Any:
    #da = self.mask(da)
    da = da[0:,:-1,] # Drop lng>500
```

Figure 16: The team's code simplifies the process by dropping "junk values" in just one line.

4.3.2 Compute normalizing mask only once

Xpublish-wms also originally created two masks in which each searched through the data to find where the longitude values were less than and equal to 180 and greater than 180.

```python
        # create 2 separate DataArrays where points lng>180 are put at the beginning of the array
        mask_0 = xr.where(da.cf["longitude"] <= 180, x: 1, y: 0)
        temp_da_0 = da.where(mask_0.compute() == 1, drop=True)
        da_0 = xr.DataArray(
            data=temp_da_0,
            dims=temp_da_0.dims,
            name=temp_da_0.name,
            coords=temp_da_0.coords,
            attrs=temp_da_0.attrs,
        )

        mask_1 = xr.where(da.cf["longitude"] > 180, x: 1, y: 0)
        temp_da_1 = da.where(mask_1.compute() == 1, drop=True)
        temp_da_1.cf["longitude"][:] = temp_da_1.cf["longitude"][:] - 360
        da_1 = xr.DataArray(
            data=temp_da_1,
            dims=temp_da_1.dims,
            name=temp_da_1.name,
            coords=temp_da_1.coords,
            attrs=temp_da_1.attrs,
        )

        # put the 2 DataArrays back together in the proper order
        da = xr.concat( objs: [da_1, da_0], dim="X")
```

Figure 17: Original code using two masks to find certain longitude values.

This was simplified by creating just one mask that checked if the longitude values were less than or equal to 180. Any value that does not fall into that bracket will be put into the other data array instead of using another mask.

```python
def project(self, da: xr.DataArray, crs: str) -> Any:
    # da = self.mask(da)

    # create 2 separate DataArrays where points lng>180 are put at the beginning of the array
    mask_0 = xr.where(da.cf["longitude"] <= 180, 1, 0)
    computedMask = mask_0.compute()
    da_0 = da.where(computedMask == 1, drop=True)

    da_1 = da.where(computedMask != 1, drop=True)
    da_1.cf["longitude"][:] = da_1.cf["longitude"][:] - 360

    # put the 2 DataArrays back together in the proper order
    da = xr.concat([da_1, da_0], dim="X")
```

Figure 18: The team's code uses only one mask to find the longitude values.

These changes reduced the number of operations performed on the dataset significantly.

4.4 Caching Solutions

In addition to the programming optimizations made above, the team investigated if caching of data could be utilized to decrease request response time. To figure out where caching could be effectively implemented, the team looked for functions that cost a lot of computation time, are frequently used, and frequently contain the same output. To do this, the team placed timers within the GetMap function and wrote the results of the timers to a CSV. After running the GetMap request a few times, the team identified that the projection and render functions took the longest time. The team concluded that caching the result of GetMap would be the most optimal solution, as it would circumvent these two expensive functions. In addition, the team investigated adding caching to the projection function to get more cache hits when it was not possible to have a cache of the requested GetMap call. An example of this occurring would be a request to render a tile of the same data layer but with a differing bounding box.

4.4.1 Redis Tile Caching

Ultimately, the final result of each GetMap query to the WMS server is the tile, so the team investigated methods to store the generated image. As Tetra Tech will be using Xpublish-wms in a cloud environment, where the number of workers scales up and down depending on the number of users, the team wanted to find a way to make sure the cache could be shared between all workers. The team decided Redis would be the optimal solution as all workers could connect to the same Redis server and share the cache. As Redis required a key-value pair to cache items, the team used the GetMap query parameters as the key

value, and the tile image buffer as the value. This was inserted into Redis when the key was not found in Redis. When a query that matches a tile stored in Redis is requested by a user, it will get the image buffer from Redis and immediately send it to the requesting client to fulfill the web request without recomputing the data. As a result, once rendered, an already requested tile can be received in less than 100 ms on average. This caching method is successful at decreasing the latency from requesting the tile and receiving it because the tiling clients rely on the Web Mercator standard. As a result, the bounding boxes of requested tiles will always be the same when in different parts of the map at a given zoom level. Had this not been the case, and the bounding box started at the leftmost part of the visible map, there would be almost no cache hits as matching the bounding box without this standard is incredibly improbable. While this strategy was proven successful at reducing latency, it did not come without any shortcomings. For example, simple changes to the query, such as changing the color mapping, would force Xpublish-wms to rerender the tile, causing a cache miss.

To combat this, the team began to investigate potential changes that could reduce the number of cache misses. Because the team did not have access to the front end of OCEANSMAP, all efforts to reduce cache misses needed to be done on Xpublish-wms. Since tiles with different color mapping would result in a cache miss, even though they contain the same data (just visualized with a different scale), the team decided that it would be beneficial to attempt to cache the image data before the shading process occurs. If successful, a previously requested tile would always get a cache hit, regardless of the requested color mapping. The only computation required before delivering the tile would be the colorization of the tile, which is not computationally expensive with image manipulation libraries such as Pillow. Currently, the shading process during rendering utilizes Python's "Matplotlib" color mapping functions. Depending on the data at the current pixel in the tile, it changes the RGBA value at this pixel to reflect the chosen colormap. The team thought of a method to do the initial shading process in grayscale, cache the grayscale tile image buffer, and then recolor the grayscale image to the requested colormap in the query to the WMS server. Utilizing the Pillow Python library, it was possible to iterate on a per-pixel basis, take the pixel color, run it through the color mapping function, and replace the pixel color with the new color value as defined by the color map. Although a cache hit request took 30 ms longer with this change compared to just storing the pre-colored tile, the server was getting many more cache hits. Additionally, when a user decides to modify the color mapping of a layer while interacting with a tiling client, they can expect a near-instant color replacement resulting from getting 100% cache hits on all requested tiles as they would have already been rendered previously.

4.4.2 Memoization Projection Caching

In addition to caching already rendered tiles, the team investigated if caching could decrease the computation time on previously unrequested tiles. As a result, the team looked into adding caching to another computationally heavy function within GetMap, the "project" function. Originally, the goal was to implement this with the same caching system as the images for tiles, Redis, but encountered difficulty. After searching for other methods, the team came across the Memoization Python library. This library was easily able to store the returned projected DataArray by developing a simple hashing method, using unique attributes located within the DataArray, ensuring there would not be any incorrect cache hits. This caching solution was able to speed up the tile rendering when a Redis cache miss occurred, and the approach was more effective on heavier datasets such as rtofs.

4.5 Results

After optimizing the HYCOM Grid and implementing the caching solutions the team observed the following results:

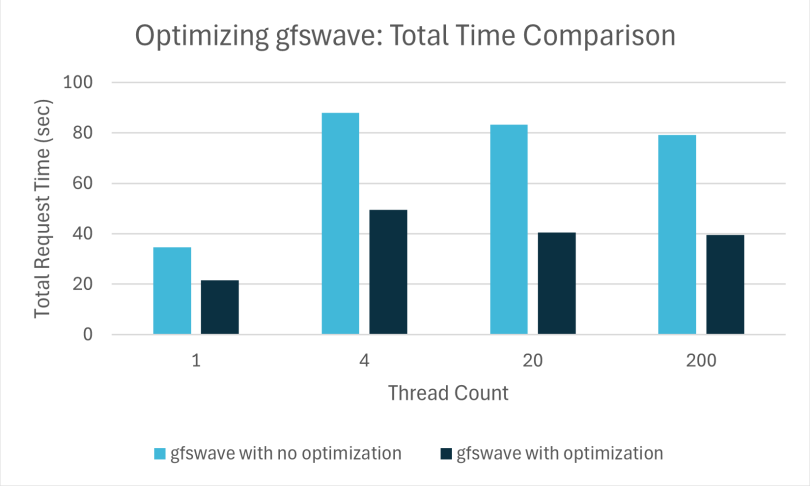


Figure 19: The gfswave model takes about half the time to render tiles with the optimizations.

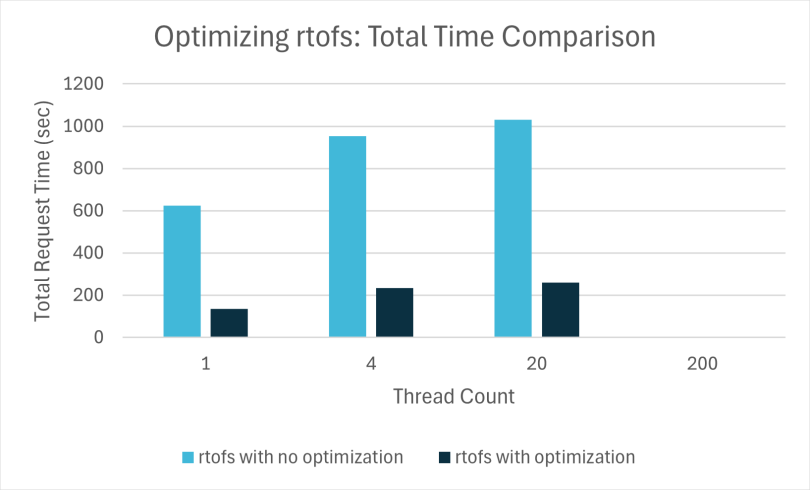


Figure 20: The rtofs model is about 4 times faster with the optimizations.

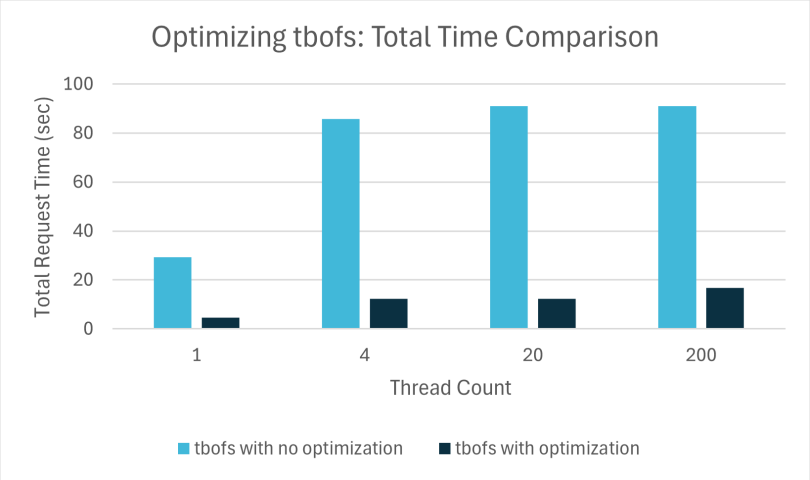


Figure 21: The tbofs model is about 6.5 times faster with the optimizations.

The main branch (which had no optimizations) took about 80 seconds to run gfswave, about 1000 seconds to run for rtofs, and about to run 90 seconds to run tbofs. However, with the team's optimizations, it took about 40 seconds for gfswave, 200 seconds for rtofs, and 11 seconds for tbofs.

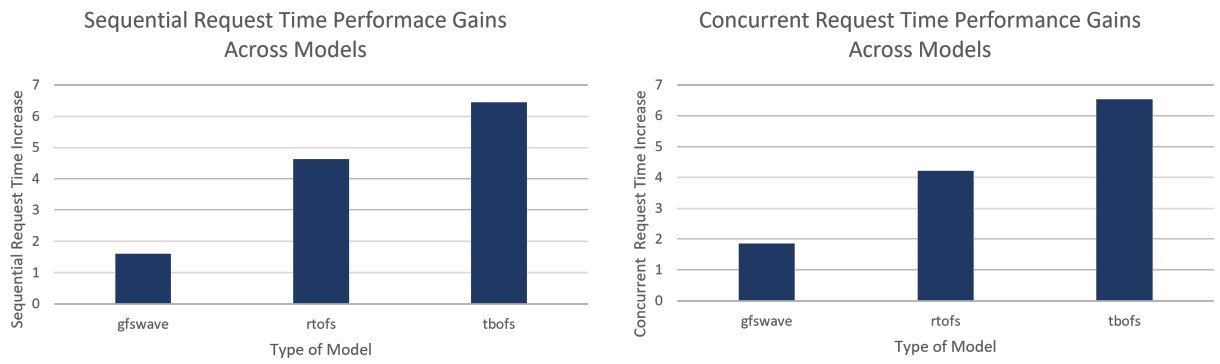


Figure 22: There were performance gains (up to 6.5 times faster) for each model after the optimizations. This was present in both sequential requests and concurrent requests.

These results demonstrate that performance improvements were achieved for each tested dataset type. For gfswave, the rendering time is almost twice as fast, for rtofs it is four times faster, and for tbofs it is about six and a half times faster. These differences in improvement make complete sense when considering the extent of optimizations each dataset benefits from. Since the gfswave dataset has data over a very large area of the globe (meaning that tile cache hits were very rare) and no optimizations to the code itself were performed, this dataset benefited solely from the projection caching that the team implemented. This had a surprisingly significant effect because the projection took the majority of the compute time for this dataset. The rtofs dataset saw a very significant increase after the optimizations were implemented because it was using both the code optimizations the team implemented as well as a caching of the projection, which cut out a substantial amount of time for each request. Finally, the tbofs dataset has a very small region of data (Tampa Bay in Florida), so this dataset was able to score tile cache hits over and over which made the request times nearly instant and increased the average speed of requests considerably. Through a meticulous combination of programming optimizations and strategic caching implementations, the team successfully achieved substantial performance gains across various models, highlighting the effectiveness of this strategy in improving overall system efficiency.

5 Discussion

5.1 Benefits at Tetra Tech

The optimizations implemented have shown large performance gains in tile generation, regardless of grid type, in Xpublish-wms. With Tetra Tech's scalable deployment of Xpublish-wms for their WMS server, Xpublish-wms may now be a suitable replacement for ncWMS for their OCEANSMAP product. Additionally, the caching solutions the team implemented within Xpublish-wms will reduce computational load on the server while generating tiles, reducing the number of workers needed for a given workload, and thus reducing the operational costs of the server.

5.2 Brief Background

Throughout this project, the team acquired a diverse set of skills and knowledge to tackle the challenges presented by Tetra Tech's OCEANSMAP project. The team began by becoming familiar with all the libraries and components of Xpublish-wms in order to understand how it functions. Understanding the geospatial data formats NetCDF and GRIB2 was essential for comprehending how data is managed within the project, especially when considering the rendering times. For example, ncWMS parsed and rendered geospatial data in formats such as NetCDF and GRIB2, but it wasn't a scalable WMS solution in comparison to Xpublish-wms. Gaining insight into Spherical Mercator tiles and projections helped the team know how the map tiles were being presented and could be manipulated. Xarray and Xpublish were crucial building blocks to parse the geospatial data and open it up to the network for the client to read.

The team's journey culminated in the implementation of caching solutions utilizing tools such as Redis, enhancing performance and scalability in tile retrieval operations and rendering times. Through the team's iterative process of experimentation and refinement, the team not only tackled the technical complexities of the OCEANSMAP project but also gained invaluable insights into problem-solving and adaptability in a real-world engineering environment.

5.3 Many Bug Fixes

One of the unexpected side effects of the project was the amount of testing and subsequent bug fixing of Xpublish-wms that the team assisted with throughout most of the second half of the project. The first time that the team got Xpublish-wms up and running, the only dataset that could return a tile

successfully was rtofs. The gfswave GRIB2 dataset returned the following error, which was discussed at a meeting with the sponsors and subsequently fixed through a GitHub issue on the project.

```
ValueError: Dimensions {'time'} do not exist. Expected one or more of ('latitude', 'longitude')
```

Unfortunately, fixing this bug (which did get gfswave working) caused a regression which then meant that the rtofs dataset returned the following error:

```
ValueError: DataArray ssh is backed by a Dask array,  but coordinate x is not backed by a Dask array with
identical  dimension order and chunks
```

This separate error required some time and another GitHub commit to solve, and in the meantime, the team had to test each dataset with a different version of Xpublish-wms so that data could be collected for both datasets.

The final major bug the team encountered was with the NOAA Operational Forecast System datasets (eg. tbofs, cbofs). These datasets would throw an error on any GetMap request when the team first started testing Xpublish (TypeError: Unhashable type: "slice"). After the first patch for this issue, the datasets would work correctly on the first GetMap request, when the server had just been restarted. However, subsequent requests would lose more and more data over time, leading to the dataset iteratively destroying itself if it was requested enough times.

Figure 23: An example of the progressive data loss seen in OFS datasets after requesting tiles consecutively through a GetMap request.

This was due to a masking issue in the grid used by the OFS datasets, and it was likely the hardest bug to diagnose that the team dealt with in the project.

Even though the team addressed many of the critical bugs encountered, there were many other bugs that the team noticed throughout the project and simply ignored because they were not serious enough to hamper progress on the project. If the project had looser time constraints, then fixing these additional

bugs would have been a high priority for the team as a project-side objective of sorts.

5.4 Future Work - Potential Client Side Optimizations

In this project, all of the team's optimizations took place within the Xpublish-wms Python plugin. While the team was able to achieve significant improvements in the rendering process speeds, improvements in the client can further improve latency for the user. For example, the tile colorization may be moved to the client. The team was able to make a proof-of-concept of this by inserting a tile into an HTML Canvas and applying a colormap on a pixel basis. When the user changed this color mapping, it was updated nearly instantly as there was no additional network request associated with getting a new colored tile. A cached tile would also return more quickly, as the WMS server would only need to return the grayscale tile, and not be concerned with recoloring; saving approximately 30ms.

In addition to coloring on the client, some additional minor adjustments can be made to improve the response time. The team identified that all tiles rendered using the same dataset resulted in a similar time, regardless of the resolution of the tile or its bounding box. The team theorizes that it could be beneficial to increase the size of tiles, especially for datasets that cover the entire planet's surface, resulting in fewer network requests required to populate the entire map. Additionally, it should be investigated if rendering the entire map as a singular tile could be the best solution.

5.4.1 Surpassing ncWMS Rendering Times

Xpublish-wms heavily relies on external libraries throughout the rendering process, some of which are slow to complete function calls and other various computations. Some examples of this include Xarray when loading the GRIB2 or NetCDF data into the code to read and render into mapping tiles. It may be more efficient to have custom-built abstractions to improve reading and data manipulation functions, preferably built in a fast and compiled language such as C or Rust. These functions written in the faster language can then be implemented into the already existing Python code using the C Foreign Function Interface (CFFI) library.

6 Conclusion

WMS as a method of retrieving geospatial data is a widely used technique in many fields. ncWMS is a commonly used, but old standard with many notable drawbacks for modern computing. Transitioning this old approach to a more modern cloud-based approach would help address the reliability and cost (under light load) issues that the existing approach suffers from. When operating in the cloud, every performance improvement that can be achieved translates directly into dollars saved. This project's findings demonstrate significant optimization methods and caching solutions that succeed in requiring less processing power to visualize geospatial data in a cloud-based environment.

The team was given the task of testing and modifying a prototype WMS server with the optimizations discussed throughout this project. Achieving the project goal took three main objectives: benchmarking an existing WMS server, comparing a prototype WMS server's performance to the existing WMS server, and implementing new optimized methodologies to improve the performance of the prototype WMS server.

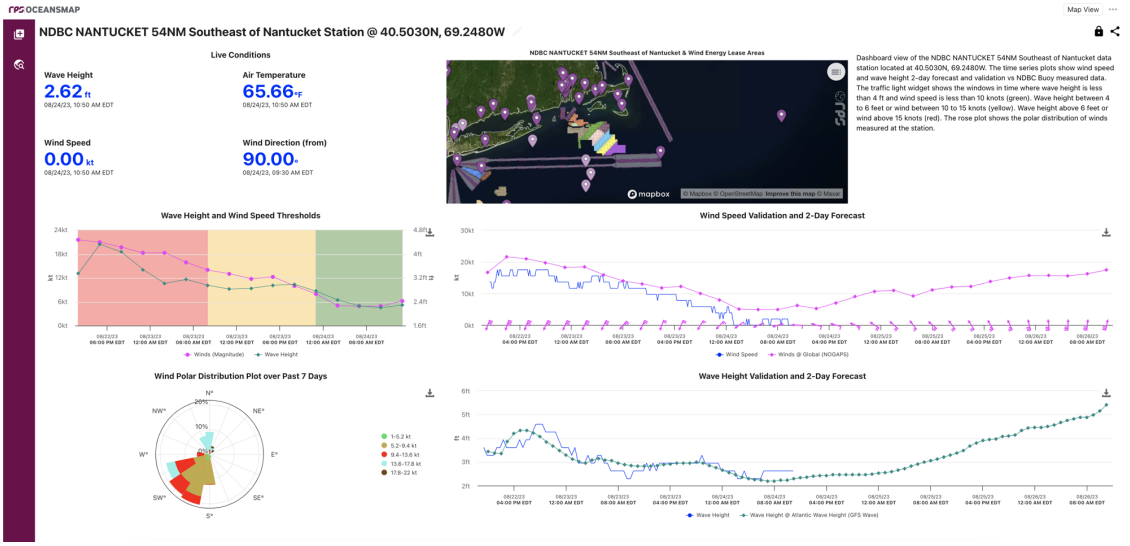
These objectives worked towards the project's goal to identify areas of slowdown in the existing WMS server and improve those areas in the prototype WMS server. Data rendering speeds were evaluated following the development of appropriate testing environments and scripts. The outcomes of these tests provide insights into the advancement of optimizations and areas requiring further refinement. The team communicated consistently with the project's sponsor to identify where to best focus efforts during the project as well as to report back any findings as the project progressed.

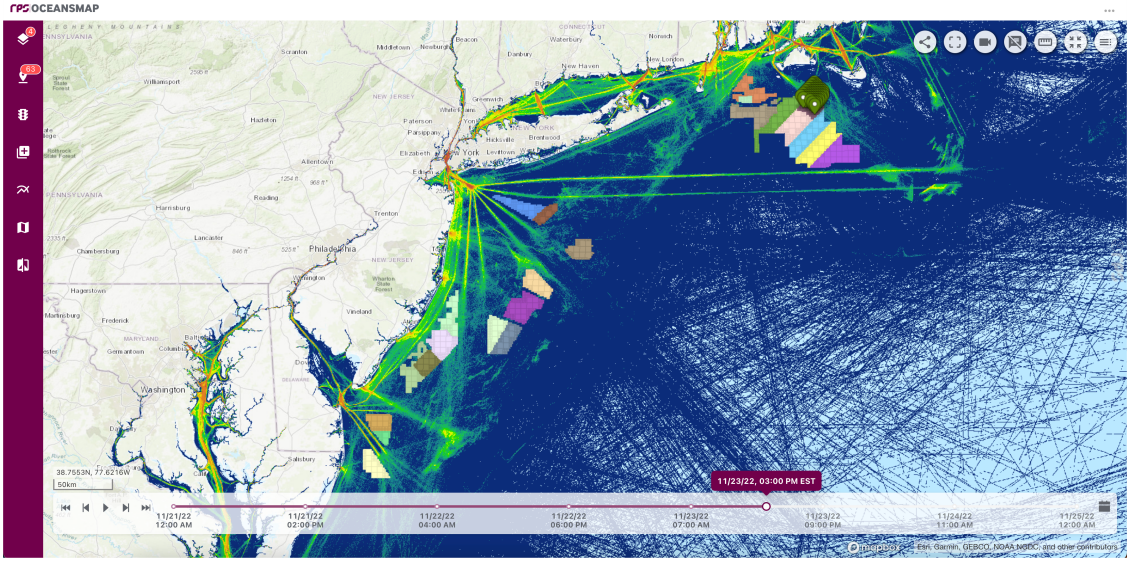
"The Efficient Delivery of Geospatial Data for Web Visualization" Major Qualifying Project is proof of several enhancements that can be made to the current state-of-the-art data delivery pipeline in geospatial analytics. These improvements are validated by increased server performance and a reduction in the number of server operations done via caching strategies.

Appendices

A OCEANSMAP

https://www.rpsgroup.com/services/oceans-and-coastal/modelling/oceansmap/





B Relevant GitHub Links

ncWMS

Xpublish-wms

WMS Testing Scripts (By MQP Team)

References

[1] "Lambert conformal conic—ArcGIS Pro | Documentation." [Online]. Available: https://pro.arcgis.com/en/pro-app/3.1/help/mapping/properties/lambert-conformal-conic.htm

[2] "What is Geospatial Data? | AWS." [Online]. Available: https://aws.amazon.com/what-is/geospatial-data/

[3] "What is Geospatial Data? | IBM." [Online]. Available: https://www.ibm.com/topics/geospatial-data

[4] "About Global RTOFS," tex.ids= globalb. [Online]. Available: https://polar.ncep.noaa.gov/global/about/

[5] J.-G. Lee and M. Kang, "Geospatial Big Data: Challenges and Opportunities," *Big Data Research*, vol. 2, no. 2, pp. 74–81, Jun. 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214579615000040

[6] "RPS services. Deep expertise in things that matter." [Online]. Available: https://www.rpsgroup.com/services/

[7] "What is GIS? | Geographic Information System Mapping Technology." [Online]. Available: https://www.esri.com/en-us/what-is-gis/overview

[8] "Network Common Data Form (NetCDF)." [Online]. Available: https://www.unidata.ucar.edu/software/netcdf/

[9] "What are GRIB files and how can I read them - Copernicus Knowledge Base - ECMWF Confluence Wiki." [Online]. Available: https://confluence.ecmwf.int/display/CKB/What+are+GRIB+files+and+how+can+I+read+them

[10] E. Borneman, "Types of Map Projections," Mar. 2023. [Online]. Available: https://www.geographyrealm.com/types-map-projections/

[11] "WMS imagery," Mar. 2021, tex.ids= wmsa, wmsc. [Online]. Available: https://onlinehelp.ihs.com/Energy/Petra/2020/Content/map_wms_imagery.htm

[12] K. Field, "Mercator, it's not hip to be square," Oct. 2019. [Online]. Available: https://www.esri.com/arcgis-blog/products/arcgis-pro/mapping/mercator-its-not-hip-to-be-square/

[13] "Tiles à la Google Maps: Coordinates, Tile Bounds and Projection | MapTiler," 2023. [Online]. Available: https://www.maptiler.com/google-maps-coordinates-tile-bounds-projection/

[14] "WMS services—ArcGIS Server | Documentation for ArcGIS Enterprise," 2023, tex.ids= wmsa. [Online]. Available: https://enterprise.arcgis.com/en/server/latest/publish-services/windows/wms-services.htm

[15] J. D. Blower, A. L. Gemmell, G. H. Griffiths, K. Haines, A. Santokhee, and X. Yang, "A Web Map Service implementation for the visualization of multidimensional gridded environmental data," *Environmental Modelling & Software*, vol. 47, pp. 218–224, Sep. 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1364815213000947

[16] "Overview: Why xarray?" [Online]. Available: https://docs.xarray.dev/en/latest/getting-started-guide/why-xarray.html

[17] "Docker overview," 2024. [Online]. Available: https://docs.docker.com/get-started/overview/

[18] P. Loshin and S. J. Bigelow, "What is the Linux operating system?" Oct. 2021. [Online]. Available: https://www.techtarget.com/searchdatacenter/definition/Linux-operating-system

[19] "What is a Web Request? - Source Defense," 2024. [Online]. Available: https://sourcedefense.com/glossary/web-request/

[20] S. Robot, "HTTP Request Tool Guide: AIOHTTP Vs. Requests Vs. HTTPX," Sep. 2023, section: All categories. [Online]. Available: https://scrapingrobot.com/blog/httpx-vs-requests/

[21] "Python - LRU Cache," May 2020, section: Python. [Online]. Available: https://www.geeksforgeeks.org/python-lru-cache/

[22] "functools — Higher-order functions and operations on callable objects." [Online]. Available: https://docs.python.org/3/library/functools.html

[23] lonelyenvoy, "memoization: A powerful caching library for Python, with TTL support and multiple algorithm options. (https://github.com/lonelyenvoy/python-memoization)," tex.ids= lonelyenvoymemoization, lonelyenvoymemoizationa. [Online]. Available: https://github.com/lonelyenvoy/python-memoization

[24] "What is memoization? A Complete tutorial," Jun. 2022, section: DSA. [Online]. Available: https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/

[25] "Redis as Cache: How It Works and Why You Should Use It," Sep. 2023. [Online]. Available: https://www.liquidweb.com/kb/redis-as-cache/

[26] T. Spindler, "Reading Scientific Datasets in Python," Sep. 2018. [Online]. Available: https://polar.ncep.noaa.gov/ngmmf_python/Python_tutorial_netcdf.slides.html#/

[27] "HYCOM Overview." [Online]. Available: https://www.hycom.org/hycom/overview