Scalable and Computationally Reproducible Approaches to Arctic Research

Matt Jones, Bryce Mecum, Jeanette Clark, Sam Csik

September 19, 2022

Table of contents

Preface

About

This 5-day in-person workshop will provide researchers with an introduction to advanced topics in computationally reproducible research in python and R, including software and techniques for working with very large datasets. This includes working in cloud computing environments, docker containers, and parallel processing using tools like parsl and dask. The workshop will also cover concrete methods for documenting and uploading data to the Arctic Data Center, advanced approaches to tracking data provenance, responsible research and data management practices including data sovereignty and the CARE principles, and ethical concerns with data-intensive modeling and analysis.

Schedule

Code of Conduct

Please note that by participating in this activity you agree to abide by the NCEAS Code of Conduct.

	Monday	Tuesday	Wednesday	Thursday	Friday
08:00-08:30	Coffee (optional)	Coffee (optional)	Coffee (optional)	Coffee (optional)	Coffee (optional)
08:30-09:00 09:00-09:30	**1. Welcome and Course Overview** (Jeanette)	**6. Data structures and formats for large data** (Bryce)	**10. Spatial and Image Data using GeoPandas** (Jeanette)	**15. Google Earth Engine** (Ingmar, Sam)	**19. What is cloud computing anyways?** (Matt)
09:30-10:00 10:00-10:30	**2. Remote computing** (Sam)		**11. Data futures: Parquet and Arrow** (Jeanette)		
10:30-11:00	BREAK	BREAK	BREAK	BREAK	BREAK
11:00-11:30	**3. Python programming on clusters** (Jeanette)	**7. Parallelization with Dask** (Bryce)	**12. Software Design II** (Bryce)	**16. Billions of Ice Wedge Polygons** (Chandi)	**20. Reproducibility redux via containers** (Bryce)
11:30-12:00					**Survey Feedback** **Q & A**
12:00-12:30 12:30-13:00	Lunch	Lunch	Lunch	Lunch	Adjourn
13:00-13:30 13:30-14:00 14:00-14:30 14:30-15:00	**4. Pleasingly Parallel Programming** (Matt)	**8. Group project I** **Data staging and pre-processing** (Jeanette)	**13. Group project II** **Parallel data processing** (Jeanette)	**17. Group project III** **Visualizing big geospatial data** (Jeanette)	
15:00-15:30	Break	Break	Break	Break	
15:30-16:00	**5. Documenting and Publishing Data** (Daphne)	**9. Software design I** (Bryce)	**14. Data Ethics** (Matt)	**18. Workflows for data staging and publishing** (Jeanette)	
16:00-16:30			**Breather** Catch-up		
16:30-17:00	Q&A	Q&A	Q&A	Q&A	

Setting Up

In this course, we will be using Python (> 3.0) as our primary language, and VS Code as our IDE. Below are instructions on how to get VS Code set up to work for the course. If you are already a regular Python user, you may already have another IDE set up. We strongly encourage you to set up VS Code with us, because we will use your local VS Code instance to write and execute code on one of the NCEAS servers.

Download VS Code and Extensions

First, download VS Code if you do not already have it installed.

Check to make sure you have Python installed if you aren't sure you do. To do this, from the terminal run:

```
python3 --version
```

If you get an error, it means you need to install Python. Here are instructions for getting installed, depending on your operating system. Note: There are many ways to install and manage your Python installations, and advantages and drawbacks to each. If you are unsure about how to proceed, feel free to reach out to the instructor team for guidance.

- Windows: Download and run an installer from Python.org.
- Mac: Install using homebrew. If you don't have homebrew installed, follow the instructions from their webpage.
  - `brew install python3`

After you run your install, make sure you check that the install is on your system PATH by running `python3 --version` again.

Set up VS Code

This section summarizes the official VS Code tutorial. For more detailed instructions and screenshots, see the source material

First, install the Python extension for VS Code.

Open a terminal window in VS Code from the Terminal drop down in the main window. Run the following commands to initialize a project workspace in a directory called `training`. This example will show you how to do this locally. Later, we will show you how to set it up on the remote server with only one additional step.

```
mkdir training
cd training
code .
```

Next, we will select the Python interpreter for the project. Open the **Command Palette** using Command + Shift + P (Control + Shift + P for windows). The Command Palette is a handy tool in VS Code that allows you to quickly find commands to VS Code, like editor commands, file edit and open commands, settings, etc. In the Command Palette, type "Python: Select Interpreter." Push return to select the command, and then select the interpreter you want to use (your Python 3.X installation).

Finally, download the Jupyter extension. You can create a test Jupyter Notebook document from the command pallete by typing "Create: New Jupyter Notebook" and selecting the command. This will open up a code editor pane with a notebook that you can test.

Test your local setup (Optional)

To make sure you can write and execute code in your project, create a Hello World test file.

- From the File Explorer toolbar, or using the terminal, create a file called `hello.py`

- Add some test code to the file, and save

```
msg = "Hello World"
print(msg)
```

- Execute the script using either the Play button in the upper-right hand side of your window, or by running `python3 hello.py` in the terminal.
  - For more ways to run code in VS Code, see the tutorial

About this book

These written materials reflect the continuous development of learning materials at the Arctic Data Center and NCEAS to support individuals to understand, adopt, and apply ethical open science practices. In bringing these materials together we recognize that many individuals have contributed to their development. The primary authors are listed alphabetically in the citation below, with additional contributors recognized for their role in developing previous iterations of these or similar materials.

Citation: Matthew B. Jones, Bryce Mecum, S. Jeanette Clark, Samantha Csik. 2022. Scalable and Computationally Reproducible Approaches to Arctic Research.

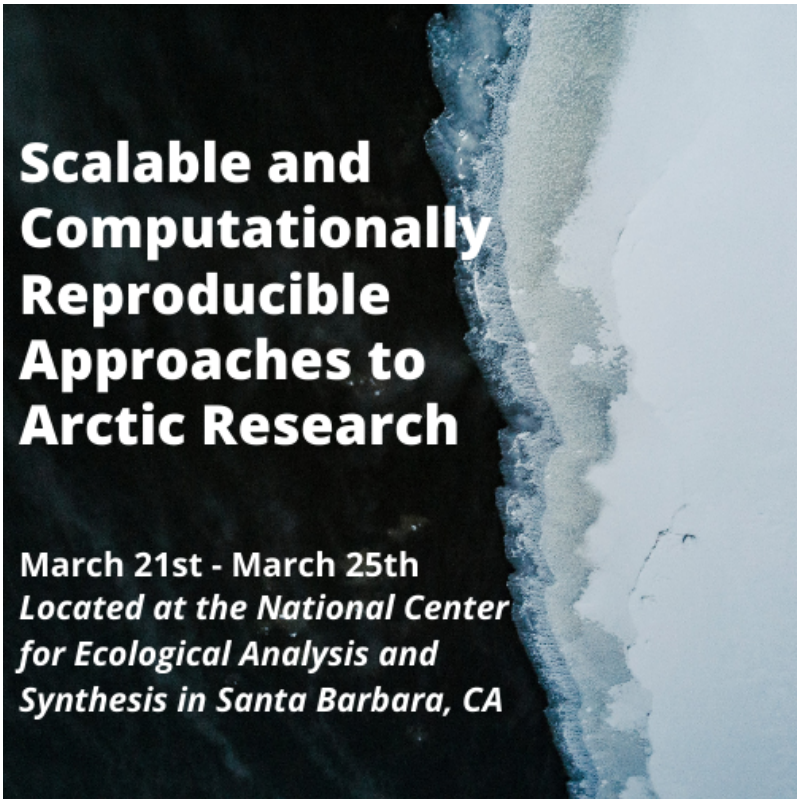
Additional contributors: Amber E. Budden, Natasha Haycock-Chavez, Noor Johnson, Stephanie Hampton, Jim Regetz, Bryce Mecum, Julien Brun, Julie Lowndes, Erin McLean, Andrew Barrett, David LeBauer, Jessica Guo.

This is a Quarto book. To learn more about Quarto books visit https://quarto.org/docs/books.

1 Welcome and Introductions

This course is one of three that we are currently offering, covering fundamentals of open data sharing, reproducible research, ethical data use and reuse, and scalable computing for reusing large data sets.


Reproducible Practices for Arctic Research Using R
February 14th - February 18th, 2022
This course will be taught virtually


Scalable and Computationally Reproducible Approaches to Arctic Research
March 21st - March 25th
Located at the National Center for Ecological Analysis and Synthesis in Santa Barbara, CA

2 Remote Computing

Notes from Google Sheet (DELETE LATER)
- Servers & Networking
- IP addressing
- Bash shell programming
- SSH
- Remote session in VS Code

- Understand the basic architecture of computer networks
- Become familiarized with Bash Shell programming to navigate your computer's file system **(??)**
- Learn how to connect to a remote computer via a shell

2.1 Introduction

- Scientific synthesis and our ability to effectively and efficiently work with big data depends on the use of computers & the internet
- VS Code + remote development on a cluster is easy and way faster than your local machine

2.2 Servers & Networking

- Host computers connect via networking equipment and can send messages to each other over communication protocols (aka internet protocols)
  - **Client:** the host *initiating* the request
  - **Server:** the host *responding* to a request

2.3 IP addressing

- Hosts are assigned a **unique numerical address** used for all communication and routing called an Internet Protocol Address (IP Address). They look something like this: 128.111.220.7
- Each IP Address can be used to communicate over various "ports", which allows multiple applications to communicate with a host without mixing up traffic
- IP addresses can be difficult to remember, so they are also assigned **hostnames**
  - Hostnames are handled through the global Domain Name System (DNS)
  - Clients first look up a hostname in DNS to find the IP address, then they open a connection the the IP address
    * aurora.nceas.ucsb.edu == 128.111.220.46 **(UPDATE THIS WITH SERVER USED FOR COURSE?)**

2.4 Bash Shell Programming

- *What is a shell?* From Wikipedia

  "a computer program which exposes an operating system's services to a human user or other programs. In general, operating system shells use either a command-line interface (CLI) or graphical user interface (GUI), depending on a computer's role and particular operation."

- *What is Bash Shell?* A command line tool (language) commonly used to manipulate files and directories
  - **Mac:** bash via the Terminal **(QUESTION: Mac users may have to switch from zsh to bash? `exec bash`? or `exec zsh` to switch back)**

  - **Windows:** bash via Git Bash

2.5 Some group exercise:

- Navigate file system (show that this is equivalent to using Finder/Windows version), create a file, edit file, etc.
  - `pwd`
  - `cd`
  - `ls`
  - `touch`
  - `mkdir`
  - (**Queston:** Do we want/need to show all of these? Missing any important ones?)

2.6 Connecting to a remote computer via a shell

- You can use a shell to gain accesss to and remotely control (manage/transfer files/etc) other computers. To do so, you'll need the following:
  - remote computer (e.g. server) turned on
  - IP address or name of remote computer
  - necessary permissions to access the remote computer
- Secure Shell, or SSH, is often used for securely connecting to and running shell commands on a remote host
  - Tremendously simplifies remote computing
  - Supported out-of-the-box on Linux and Macs

2.7 Exercise:

1. Launch your Terimal program:
    - **MacOS:** navigate to `Applications | Utilities` and open `Terminal`
    - **Windows:** Navigate to `Windows Start | Git` and open `Git Bash`
    - **ALTERNATIVELY, from VS Code:** Two options to open a terminal program
        a) Click on `Terminal | New Terminal` in top menu bar
        b) Click on the `+ (dropdown menu) | bash` in the bottom right corner **(QUESTION: Not sure that is always open/available depending on user configurations??)**
2. Connect to a remote server **(UPDATE THIS SECTION)**

```
jones@powder:~$ ssh jones@aurora.nceas.ucsb.edu
jones@aurora.nceas.ucsb.edu's password:
jones@aurora:~$
```

3. Change your password **(UPDATE THIS SECTION)**

```
jones@aurora:~$ passwd
Changing password for jones.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
```

4. **create python script on server | write/execute some code | etc**

3 Python Programming on Clusters

- Basic Python review
- Using virtual environments
- Writing in Jupyter notebooks
- Writing functions in Python

3.1 Introduction

- VS Code + remote development on a cluster is easy and way faster than your local machine
- Jupyter is a great way to do literate analysis
- Functions provide ways to reuse your code across notebooks/projects

3.2 Starting a project

- Connect to the server
- Start a `training` workspace

3.3 Virtual Environments

Why virtual environments? We'll answer this.

First we will create `.bash_profile` file to create variables that point to the install locations of python and `virtualenvwrapper`. `.bash_profile` is just a text file that contains bash commands that are run every time you start up a new terminal. Although setting up this file is not required to

use `virtualenvwrapper`, it is convenient because it allows you to set up some reasonable defaults to the commands (meaning less typing, overall), and it makes sure that the package is available every time you start a new terminal.

To set up the `.bash_profile`. In VS Code, select 'File > New Text File' then paste this into the file:

```
export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python3
export VIRTUALENVWRAPPER_VIRTUAWORKON_HOME=$HOME/.virtualenvs
source /usr/share/virtualenvwrapper/virtualenvwrapper.sh
```

The first line points `virtualenvwrapper` to the default python installation to use. In this case, we point it to the system wide install of python on the server. The next line sets the directory where your virtual environments will be stored. We point it to a hidden directory (`.virtualenvs`) in your home directory. Finally, the last line sources a bash script that ships with `virtualenvwrapper`, which makes all of `virtualenvwrapper` commands available in your terminal session.

Save the file in the top of your home directory as `.bash_profile`.

Restart your terminal, then check to make sure it was installed and configured correctly

```
mkvirtualenv --version
```

Now we can create the virtual environment we will use for the course

```
mkvirtualenv scomp
```

By default, this will point to our Python 3.9 installation on the server, because of the settings in `.bash_profile`. If you want to point to a different version of python, or be more explicit about the version, you can use the `-p` flag and pass a path to the python install, like so:

```
mkvirtualenv -p /usr/bin/python3 test
```

After making a virtual environment, it will automatically be activated. You'll see the name of the env you are working in on the left side of your terminal prompt in parentheses. To deactivate your environment (like if you want to work on a different project), just run `deactivate`. To activate it again, run

```
workon scomp
```

You can get a list of all available environments by just running:

```
workon
```

Now let's install the dependencies for this course into that environment. (Note: need to figure out how to get them this file)

```
python3 -m pip install -r requirements.txt
```

3.3.0.1 Installing locally (optional)

`virtualenvwrapper` was already installed on the server we are working on. To install on your local computer, run:

```
pip3 install virtualenvwrapper
```

And then follow the instructions as described above, making sure that you have the correct paths set when you edit your `.bash_profile`.

3.4 Brief overview of python syntax

Assign values to variables using `=`

```
x = 4
print(x)
```

```
4
```

There are 5 standard data types in python

- Number (int, long, float, complex)
- String
- List
- Tuple
- Dictionary

We already saw a number type, here is a string:

```
str = 'Hello World!'
print(str)
```

```
Hello World!
```

Lists in python are very versatile, and are created using square brackets `[]`. Items in a list can be of different data types.

```
list = [100, 50, -20, 'text']
print(list)
```

```
[100, 50, -20, 'text']
```

You can access items in a list by index using the square brackets. Note indexing starts with 0 in python. The slice operator enables you to easily access a portion of the list without needing to specify every index.

```
list[0] # print first element
list[1:3] # print 2nd until 4th elements
list[:2] # print first until the 3rd
list[2:] # print last elements from 3rd
```

```
100
```

```
[50, -20]
```

```
[100, 50]
```

```
[-20, 'text']
```

The + and * operators work on lists by creating a new list using either concatenation (+) or repetition (*).

```
list2 = ['more', 'things']

list + list2
list * 3
```

```
[100, 50, -20, 'text', 'more', 'things']
```

```
[100, 50, -20, 'text', 100, 50, -20, 'text', 100, 50, -20, 'text']
```

Tuples are similar to lists, except the values cannot be changed in place. They are constructred with parentheses.

```
tuple = ('a', 'b', 'c', 'd')
tuple[0]
tuple * 3
tuple + tuple
```

```
'a'
```

```
('a', 'b', 'c', 'd', 'a', 'b', 'c', 'd', 'a', 'b', 'c', 'd')
```

```
('a', 'b', 'c', 'd', 'a', 'b', 'c', 'd')
```

Observe the difference when we try to change the first value. It works for a list:

```
list[0] = 'new value'
list
```

```
['new value', 50, -20, 'text']
```

...and errors for a tuple.

```
tuple[0] = 'new value'
```

```
TypeError: 'tuple' object does not support item assignment
```

Dictionaries consist of key-value pairs, and are created using the syntax `{key: value}`. Keys are usually numbers or strings, and values can be any data type.

```
dict = {'name': ['Jeanette', 'Matt'],
        'location': ['Tucson', 'Juneau']}

dict['name']
dict.keys()
```

```
['Jeanette', 'Matt']
```

```
dict_keys(['name', 'location'])
```

To determine the type of an object, you can use the `type()` method.

```
type(list)
type(tuple)
type(dict)
```

```
list
```

```
tuple
```

```
dict
```

3.5 Jupyter notebooks

To create a new notebook, from the file menu select File > New File > Jupyter Notebook

At the top of your notebook, add a first level header using a single hash. Practice some markdown text by creating:

- a list
- **bold** text
- a link

Use the Markdown cheat sheet if needed.

To open a chunk of code, type three backticks (‘), curly braces, and then the word python. Close the code chunk using three more backticks.

3.5.1 Load libraries

In your first code chunk, lets load in some modules. We’ll use `pandas`, `numpy`, `matplotlib.pyplot`, `requests`, `skimpy`, and `exists` from `os.path`.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import requests
import skimpy
from os.path import exists
```

A note on style: There are a few ways to construct import statements. The above code uses three of the most common:

```
import module
import module as m
from module import function
```

The first way of importing will make the module a function comes from more explicitly clear, and is the simplest. However for very long module names, or ones that are used very frequently (like `pandas`, `numpy`, and `matplotlib.plot`), the code in the notebook will be more cluttered with constant calls to longer module names. So `module.function()` instead is written as `m.function()`

The second way of importing a module is a good style to use in cases where modules are used frequently, or have extremely long names. If you import every single module with a short name, however, you might have a hard time remembering which modules are named what, and it might be more confusing for others trying to read your code. Many of the most commonly used libraries for python data science have community driven styling for how they are abbreviated in import statements, and these community norms are generally best followed.

Finally, the last way to import a single object from a module can be helpful if you only need that one piece from a larger module, but again, like the first case, results in less explicit code and therefore runs the risk of your or someone else misremembering the usage and source.

3.5.2 Read in a csv

Create a new code chunk that will download the csv that we are going to use for this tutorial.

- Navigate to Rohi Muthyala, Åsa Rennermalm, Sasha Leidman, Matthew Cooper, Sarah Cooley, et al. 2022. 62 days of Supraglacial streamflow from June-August, 2016 over southwest Greenland. Arctic Data Center. doi:10.18739/A2XW47X5F.
- Right click the download button for 'Discharge_timeseries.csv'
- Click 'copy link address'

Create a variable called URL and assign it the link copied to your clipboard. Then use `requests.get` to download the file, and `open` to write it to disk, to a directory called `data/`

```
if not exists('data/discharge_timeseries.csv'):

        url = 'https://arcticdata.io/metacat/d1/mn/v2/object/urn%3Auuid%3Ae248467d-e1f9-4a32

        data = requests.get(url)
        a = open('data/discharge_timeseries.csv', 'wb').write(data.content)
```

Now we can read in the data from the file.

```
df = pd.read_csv('data/discharge_timeseries.csv')
df.head()
```

```
/home/runner/.local/lib/python3.8/site-packages/IPython/core/formatters.py:343: FutureWarning:
  return method()
```

	Date	Total Pressure [m]	Air Pressure [m]	Stage [m]	Discharge [m3/s]	temperature [degree
0	6/13/2016 0:00	9.816	9.609775	0.206225	0.083531	
1	6/13/2016 0:05	9.810	9.609715	0.200285	0.077785	
2	6/13/2016 0:10	9.804	9.609656	0.194344	0.072278	
3	6/13/2016 0:15	9.800	9.609596	0.190404	0.068756	
4	6/13/2016 0:20	9.793	9.609537	0.183463	0.062804	

The column names are a bit messy so we can use `clean_columns` from `skimpy` to make them more accessible very quickly. We can also use the `skim` function to get a quick summary of the data.

```
clean_df = skimpy.clean_columns(df)
skimpy.skim(clean_df)
```

```
6 column names have been cleaned

                    skimpy summary
          Data Summary                Data Types

   dataframe           Values     Column Type   Count
```

```
Number of rows      17856     float64       5
Number of columns   6         string        1

                                        number

column_name     NA   NA %    mean     sd      p0        p25     p75    p100   hist

total_pressur    0       0      9.9    0.12       9.6     9.8     10     10
e_m
air_pressure_    0       0      9.6    0.06       9.5     9.6    9.7    9.7
m
stage_m          0       0     0.28    0.12   0.00056    0.17   0.37   0.56
discharge_m_3    0       0     0.22    0.19   4.7e-08   0.055   0.35   0.96
_s
temperature_d    8   0.045   -0.034   0.053      -0.1    -0.1      0    0.2
egrees_

                                        string

column_name           NA      NA %       words per row           total words

date                       0          0                        2               36000

                      End
```

We can see that the `date` column is classed as a string, and not a date, so let's fix that.

```
clean_df['date'] = pd.to_datetime(clean_df['date'])
skimpy.skim(clean_df)
```

```
                    skimpy summary
        Data Summary                Data Types

dataframe           Values    Column Type   Count

Number of rows      17856     float64       5
Number of columns   6         datetime64    1
```

```
                                                number

column_name     NA   NA %    mean     sd      p0        p25     p75    p100   hist

total_pressur    0       0     9.9    0.12       9.6     9.8     10     10
e_m
air_pressure_    0       0     9.6    0.06       9.5     9.6    9.7    9.7
m
stage_m          0       0    0.28    0.12   0.00056    0.17   0.37   0.56
discharge_m_3    0       0    0.22    0.19   4.7e-08   0.055   0.35   0.96
_s
temperature_d    8   0.045  -0.034   0.053      -0.1    -0.1      0    0.2
egrees_

                                          datetime

column_name      NA    NA %    first           last                      frequency

date               0        0    2016-06-13       2016-08-13 23:55:00     5T

                      End
```

If we wanted to calculate the daily mean flow (as opposed to the flow every 5 minutes), we need to:

- create a new column with only the date
- group by that variable
- summarize over it by taking the mean of the discharge variable

First we should probably rename our existing date/time column to prevent from getting confused.

```
clean_df = clean_df.rename(columns = {'date': 'datetime'})
```

Now create the new date column

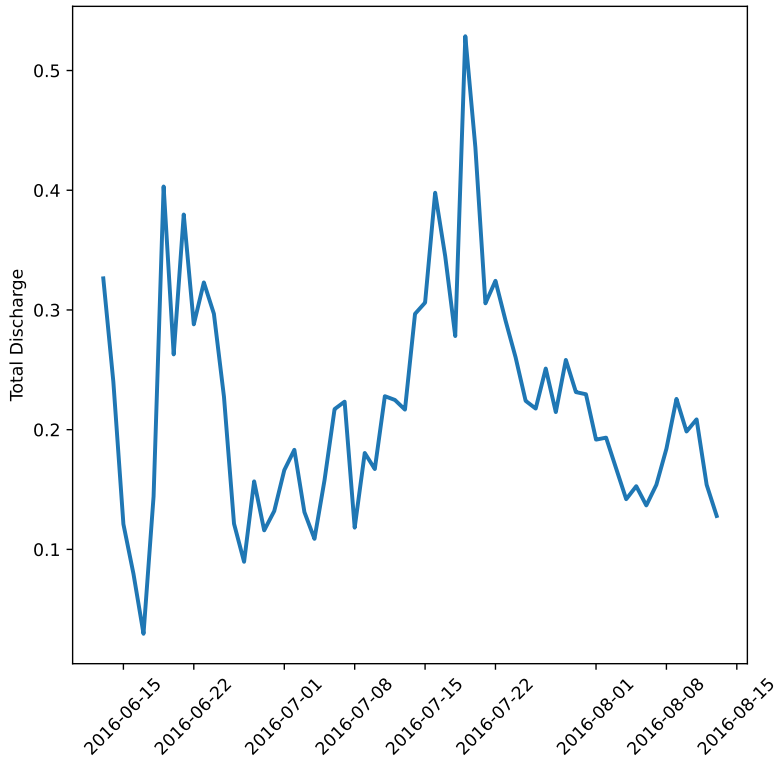
```
clean_df['date'] = clean_df['datetime'].dt.date
```

Finally, we use group by to split the data into groups according to the date, apply a function (**mean**) to each group, and then combine the results in a single data table.

```
daily_flow = clean_df.groupby('date', as_index = False).mean()
```

- create a simple plot

```
var = 'discharge_m_3_s'
var_labs = {'discharge_m_3_s': 'Total Discharge'}

fig, ax = plt.subplots(figsize=(7, 7))
plt.style.use("seaborn-talk")
plt.plot(daily_flow['date'], daily_flow[var]);
plt.xticks(rotation = 45);
ax.set_ylabel(var_labs.get('discharge_m_3_s'));
```



3.6 Functions

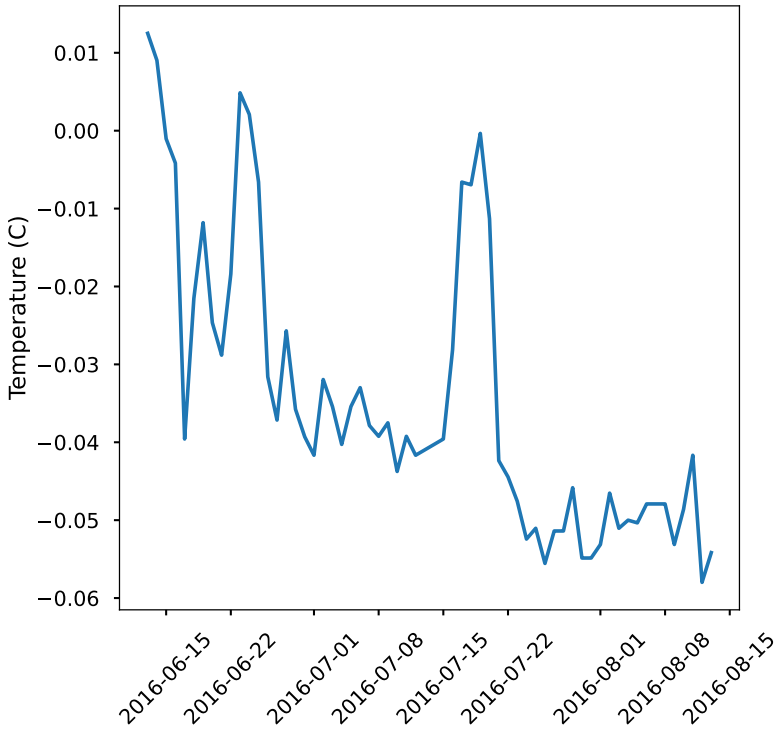
The plot we made above is great, but what if we wanted to make it for each variable? We could copy paste it and replace some things, but this violates a core tenet of programming: Don't Repeat Yourself! Instead, we'll create a function called `myplot` that accepts the data frame and variable as arguments.

- create `myplot.py`

```
def myplot(df, var):

        var_labs = {'discharge_m_3_s': 'Total Discharge (m^3/s)',
                    'total_pressure_m': 'Total Pressure (m)',
                    'air_pressure_m': 'Air Pressure (m)',
                    'stage_m': 'Stage (m)',
                    'temperature_degrees_c': 'Temperature (C)'}

        fig, ax = plt.subplots(figsize=(7, 7))
        plt.style.use("seaborn-talk")
        plt.plot(df['date'], df[var]);
        plt.xticks(rotation = 45);
        ax.set_ylabel(var_labs.get(var));
```

- load myplot into jupyter notebook (`from myplot.py import myplot`)
- replace old plot method with new function

```
myplot(daily_flow, 'temperature_degrees_c')
```



- more to come in Bryce's section

3.7 Resources

4 Pleasingly Parallel Programming

- Understand what parallel computing is and when it may be useful
- Understand how parallelism can work
- Review sequential loops and map functions
- Build a parallel program using `concurrent.futures`
- Build a parallel program using `parsl`
- Understand Thread Pools and Process pools

4.1 Introduction

Processing large amounts of data with complex models can be time consuming. New types of sensing means the scale of data collection today is massive. And modeled outputs can be large as well. For example, here's a 2 TB (that's Terabyte) set of modeled output data from Ofir Levy et al. 2016 that models 15 environmental variables at hourly time scales for hundreds of years across a regular grid spanning a good chunk of North America:

There are over 400,000 individual netCDF files in the Levy et al. microclimate data set. Processing them would benefit massively from parallelization.

Alternatively, think of remote sensing data. Processing airborne hyperspectral data can involve processing each of hundreds of bands of data for each image in a flight path that is repeated many times over months and years.

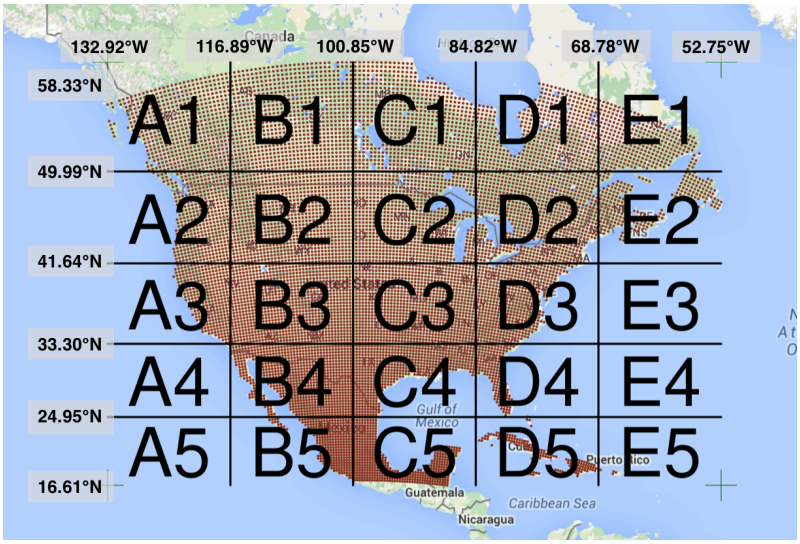


Figure 4.1: Levy et al. 2016. doi:10.5063/F1Z899CZ

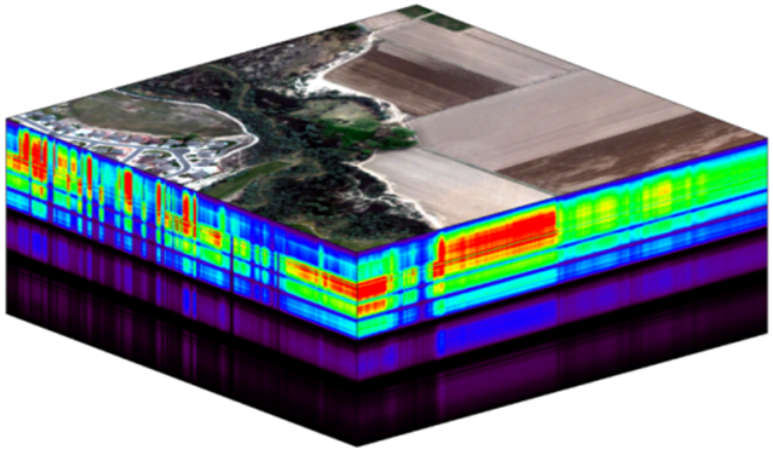


Figure 4.2: NEON Data Cube

4.2 Why parallelism?

Much R code runs fast and fine on a single processor. But at times, computations can be:

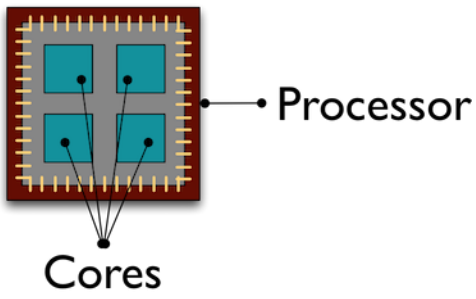
- **cpu-bound**: Take too much cpu time
- **memory-bound**: Take too much memory
- **I/O-bound**: Take too much time to read/write from disk
- **network-bound**: Take too much time to transfer

To help with **cpu-bound** computations, one can take advantage of modern processor architectures that provide multiple cores on a single processor, and thereby enable multiple computations to take place at the same time. In addition, some machines ship with multiple processors, allowing large computations to occur across the entire cluster of those computers. Plus, these machines also have large amounts of memory to avoid **memory-bound** computing jobs.

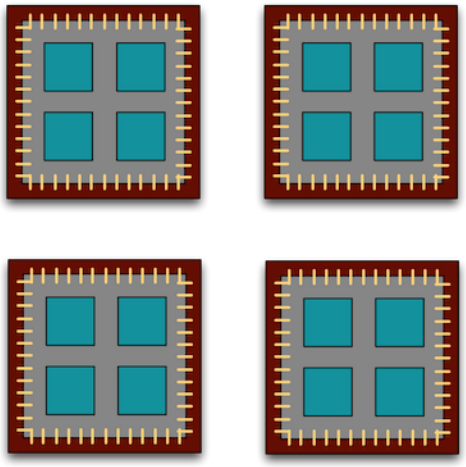
4.3 Processors (CPUs) and Cores

A modern CPU (Central Processing Unit) is at the heart of every computer. While traditional computers had a single CPU, modern computers can ship with mutliple processors, which in turn can each contain multiple cores. These processors and cores are available to perform computations.

A computer with one processor may still have 4 cores (quad-core), allowing 4 computations to be executed at the same time.



A typical modern computer has multiple cores, ranging from one or two in laptops to thousands in high performance compute clusters. Here we show four quad-core processors for a total of 16 cores in this machine.



You can think of this as allowing 16 computations to happen at the same time. Theroetically, your computation would take 1/16 of the time (but only theoretically, more on that later).

Historically, R has only utilized one processor, which makes it single-threaded. Which is a shame, because the 2017 MacBook Pro that I am writing this on is much more powerful than that:

```
{bash eval=FALSE} jones@powder:~$ sysctl hw.ncpu
hw.physicalcpu hw.ncpu: 8 hw.physicalcpu: 4
```

To interpret that output, this machine `powder` has 4 physical CPUs, each of which has two processing cores, for a total of 8 cores for computation. I'd sure like my R computations to use all of that processing power. Because its all on one machine, we can easily use *multicore* processing tools to make use of those cores. Now let's look at the computational server `aurora` at NCEAS:

```
{bash eval=FALSE} jones@included-crab:~$ lscpu
| egrep 'CPU\(s\)|per core|per socket' CPU(s):
88 On-line CPU(s) list:   0-87 Thread(s) per core:
2 Core(s) per socket:    22 NUMA node0 CPU(s):
```

```
0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46,48,50,52,54,56,58,60,62,64,6
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,43,45,47,49,5
```

Now that's more compute power! `included-crab` has 384 GB of RAM, and ample storage. All still under the control of a single operating system.

However, maybe one of these NSF-sponsored high performance computing clusters (HPC) is looking attractive about now:

- JetStream
  - 640 nodes, 15,360 cores, 80TB RAM
- Stampede2 at TACC is coming online in 2017
  - 4200 nodes, 285,600 cores
- TODO: update with modern cluster sizes

Note that these clusters have multiple nodes (hosts), and each host has multiple cores. So this is really multiple computers clustered together to act in a coordinated fashion, but each node runs its own copy of the operating system, and is in many ways independent of the other nodes in the cluster. One way to use such a cluster would be to use just one of the nodes, and use a multi-core approach to parallelization to use all of the cores on that single machine. But to truly make use of the whole cluster, one must use parallelization tools that let us spread out our computations across multiple host nodes in the cluster.

4.4 Modes of parallelization

- TODO: develop diagram(s) showing
  - Single memory image task parallelization

Serial `Launch tasks --> Task 1 --> Task 2 --> Task 3 --> Task 4 --> Task 5 --> Finish`

Parallel
```
Launch tasks -->                    Task 1 --\
                    Task 2 ---\                  Task 3 -----> Finish
                                Task 4 ---/
Task 5 --/
```

- Cluster task parallelization

```
Cluster    parallel    Show dispatch to cluster nodes and
reassembly of data    Launch tasks -->                      Marshal
--> Task 1 --> Unmarshal --\                     Marshal
--> Task 2 --> Unmarshal ---\                     Marshal
--> Task 3 --> Unmarshal -----> Finish                        Marshal
--> Task 4 --> Unmarshal ---/                     Marshal
--> Task 5 --> Unmarshal --/
```

- TODO: Should we also include figure with data or functional dependencies?

4.5 Task parallelism with `concurrent.futures`

When you have a list of repetitive tasks, you may be able to speed it up by adding more computing power. If each task is completely independent of the others, then it is a prime candidate for executing those tasks in parallel, each on its own core. For example, let's build a simple loop that downloads the data files that we need for an analysis. First, we start with the serial implementation.

```
# Use loop for serial execution of tasks

# Tasks are to download data from a dataset
```

The issue with this loop is that we execute each trial sequentially, which means that only one of our 8 processors on this machine are in use. In order to exploit parallelism, we need to be able to dispatch our tasks as functions, with one task going to each processor. To do that, we need to convert our task to a function, and then use the `map()` function to apply that function to all of the members of a set. Here's the same code rewritten to use `map()`, which applies a function to each of the members of a list (in this case the files we want to download):

```
# Use `map` for serial execution of tasks

# Tasks are to download data from a dataset
```

4.6 Approaches to parallelization

When parallelizing jobs, one can:

- Use the multiple cores on a local computer through `mclapply`
- Use multiple processors on local (and remote) machines using `makeCluster` and `clusterApply`
  - In this approach, one has to manually copy data and code to each cluster member using `clusterExport`
  - This is extra work, but sometimes gaining access to a large cluster is worth it

4.7 concurrent.futures

```
# Loop versus map for parallel execution of tasks

# Using concurrent.futures and ThreadPool

# Tasks are to download data from a dataset
```

4.8 parsl

- Overview of parsl and it's use of python decorators.

```
# Loop versus map for parallel execution of tasks

# Using parsl decorators and ThreadPool

# Tasks are to download data from a dataset
```

- Configurable Executors in parsl
  - HightThroughputExecutor for cluster jobs

```
# Loop versus map for parallel execution of tasks

# Using parsl decorators and ThreadPool

# Tasks are to download data from a dataset
```

4.9 When to parallelize

It's not as simple as it may seem. While in theory each added processor would linearly increase the throughput of a computation, there is overhead that reduces that efficiency. For example, the code and, importantly, the data need to be copied to each additional CPU, and this takes time and bandwidth. Plus, new processes and/or threads need to be created by the operating system, which also takes time. This overhead reduces the efficiency enough that realistic performance gains are much less than theoretical, and usually do not scale linearly as a function of processing power. For example, if the time that a computation takes is short, then the overhead of setting up these additional resources may actually overwhelm any advantages of the additional processing power, and the computation could potentially take longer!

In addition, not all of a task can be parallelized. Depending on the proportion, the expected speedup can be significantly reduced. Some propose that this may follow Amdahl's Law, where the speedup of the computation (y-axis) is a function of both the number of cores (x-axis) and the proportion of the computation that can be parallelized (see colored lines):

```
#| eval: false
library(ggplot2)
library(tidyr)
amdahl <- function(p, s) {
  return(1 / ( (1-p) + p/s  ))
```

```
}
doubles <- 2^(seq(0,16))
cpu_perf <- cbind(cpus = doubles, p50 = amdahl(.5, doubles))
cpu_perf <- cbind(cpu_perf, p75 = amdahl(.75, doubles))
cpu_perf <- cbind(cpu_perf, p85 = amdahl(.85, doubles))
cpu_perf <- cbind(cpu_perf, p90 = amdahl(.90, doubles))
cpu_perf <- cbind(cpu_perf, p95 = amdahl(.95, doubles))
#cpu_perf <- cbind(cpu_perf, p99 = amdahl(.99, doubles))
cpu_perf <- as.data.frame(cpu_perf)
cpu_perf <- cpu_perf %>% gather(prop, speedup, -cpus)
ggplot(cpu_perf, aes(cpus, speedup, color=prop)) +
  geom_line() +
  scale_x_continuous(trans='log2') +
  theme_bw() +
  labs(title = "Amdahl's Law")
```

So, its important to evaluate the computational efficiency of requests, and work to ensure that additional compute resources brought to bear will pay off in terms of increased work being done. With that, let's do some parallel computing...

5 Parallel Pitfalls and their solutions

- Race conditions
- Deadlocks

5.1 Summary

In this lesson, we showed examples of computing tasks that are likely limited by the number of CPU cores that can be applied, and we reviewed the architecture of computers to understand the relationship between CPU processors and cores. Next, we reviewed the way in which traditional `for` loops in R can be rewritten as functions that are applied to a list serially using `lapply`, and then how the `parallel` package `mclapply` function can be substituted in order to utilize multiple cores on the local computer to speed up computations. Finally, we installed and reviewed the use of the `foreach` package with the `%dopar` operator to accomplish a similar parallelization using multiple cores.

5.2 Further Reading

Ryan Abernathey & Joe Hamman. 2020. Closed Platforms vs. Open Architectures for Cloud-Native Earth System Analytics. Medium.

6 Documenting and Publishing Data

6.1 Introduction

7 Spatial and Image Data Using GeoPandas

- Reading raster data with rasterasterio
- Using geopandas and rasterasterio to process raster data
- Working with raster and vector data together

7.1 Introduction

- Raster vs vector data
- What is a projection
- Processing overview
  - goal is to calculate vessel distance per commercial fishing area

7.2 Pre-processing raster data

This is a test to make sure we can run some code in this notebook.

```
import geopandas as gpd
import rasterio
import rasterio.mask
import rasterio.warp
import rasterio.plot
from rasterio import features
from shapely.geometry import box
from shapely.geometry import Polygon
import requests
import matplotlib.pyplot as plt
from matplotlib import style
```

```
import pandas as pd
import numpy as np
```

Download the ship traffic raster from Kapsar et al.. We grab a one month slice from December, 2020 of a coastal subset of data with 1km resolution.

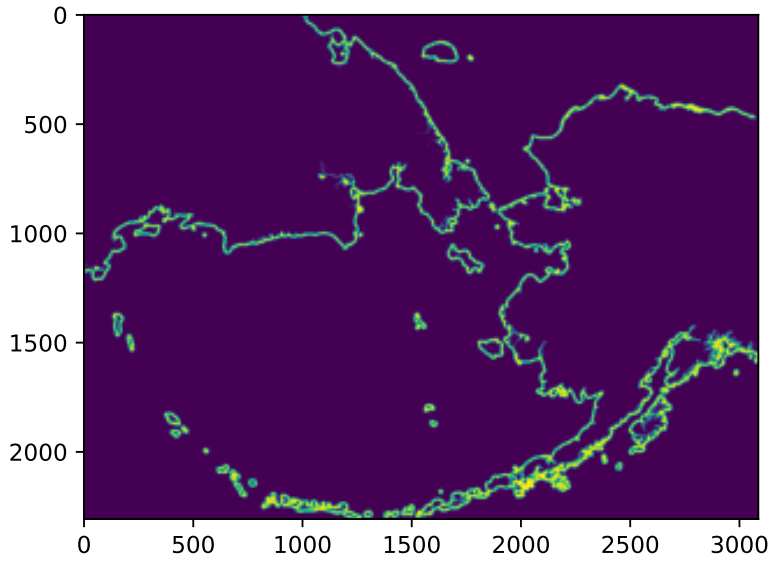
```
url_sf = 'https://cn.dataone.org/cn/v2/resolve/urn:uuid:dd61089d-f50e-4d87-9b75-6b4e2bd24776

response_sf = requests.get(url_sf)
open("Coastal_2020_12.tif", "wb").write(response_sf.content)
```

```
1132748
```

Open the raster file, plot it, and look at the metadata.

```
with rasterio.open("Coastal_2020_12.tif") as dem_src:
    ships = dem_src.read(1)
    ships_meta = dem_src.profile

plt.imshow(ships)
print(ships_meta)
```

```
{'driver': 'GTiff', 'dtype': 'float32', 'nodata': -3.3999999521443642e+38, 'width': 3087, 'heig
       0.0, -999.9687691991521, 2711703.104608573), 'tiled': False, 'compress': 'lzw', 'interle
```



Now download a vector shapefile of commercial fishing districts in Alaska.

```
url = 'https://knb.ecoinformatics.org/knb/d1/mn/v2/object/urn%3Auuid%3A7c942c45-1539-4d47-b4

response = requests.get(url)
open("Alaska_Commercial_Salmon_Boundaries.gpkg", "wb").write(response.content)
```

```
36544512
```

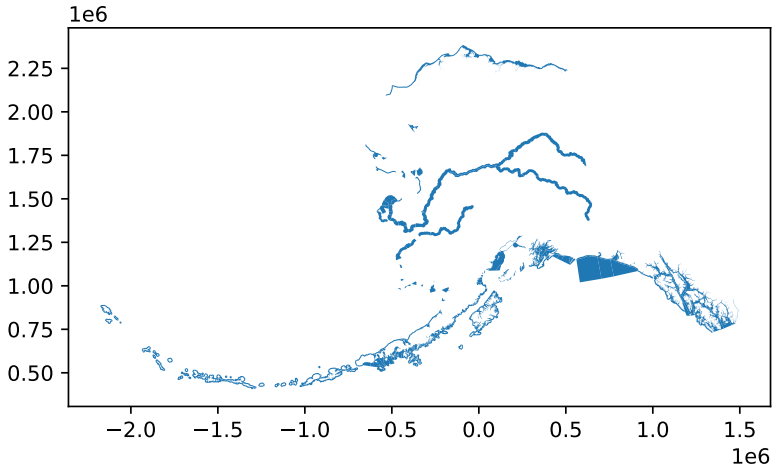
Read in the data

```
comm = gpd.read_file("Alaska_Commercial_Salmon_Boundaries.gpkg")
```

The raster data is in 3338, so we need to reproject this.

```
comm.crs
comm_3338 = comm.to_crs("EPSG:3338")

comm_3338.plot()
```

```
<AxesSubplot:>
```



We can extract the bounding box for the area of interest, and use that to clip the original raster data to just the extent we need. We use the `box` function from `shapely` to create the bounding box, then create a `geoDataFrame` from them and convert the WGS84 coordinates to the Alaska Albers projection.

todo: explain the warp transform thing here

```
coords = rasterio.warp.transform_bounds('EPSG:4326',
                                        'EPSG:3338',
                                         -159.5,
                                         55,
                                         -144.5,
                                         62)
coord_list = list(coords)

coord_box = box(coord_list[0],coord_list[1], coord_list[2], coord_list[3])

bbox_crop = gpd.GeoDataFrame(
    crs = 'EPSG:3338',
    geometry = [coord_box])
```

Read in raster again cropped to bounding box.

```
with rasterio.open("Coastal_2020_12.tif") as src:
    out_image, out_transform = rasterio.mask.mask(src, bbox_crop["geometry"], crop=True)
    out_meta = src.meta

out_meta.update({"driver": "GTiff",
                 "height": out_image.shape[1],
                 "width": out_image.shape[2],
                 "transform": out_transform,
                 "compress": "lzw"})

with rasterio.open("Coastal_2020_12_masked.tif", "w", **out_meta) as dest:
    dest.write(out_image)
```

We can also clip the shapefile data to the same bounding box

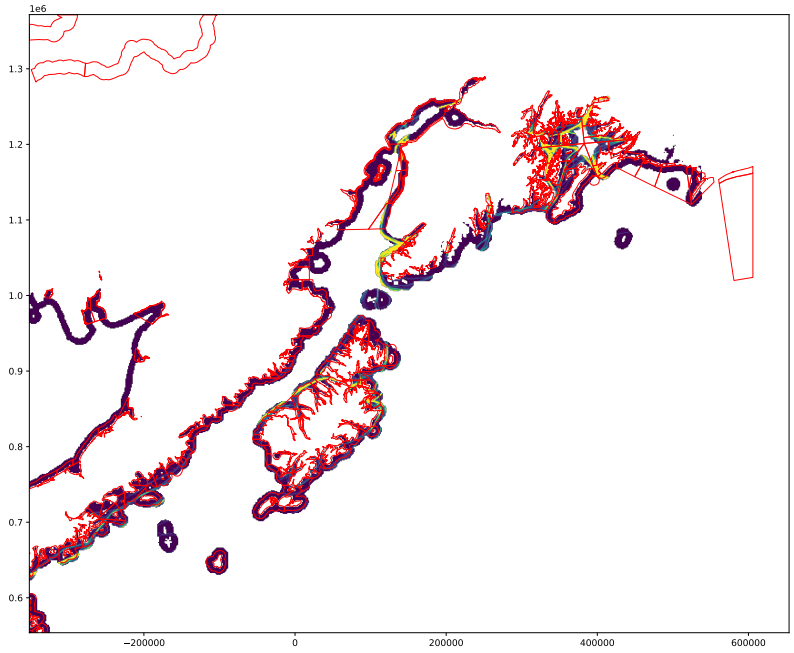
```
comm_clip = comm_3338.clip(bbox_crop['geometry'])
```

7.2.1 Check extents

Quick plot to ensure they are in the same extent, and look as expected.

```
with rasterio.open('Coastal_2020_12_masked.tif') as src:
    r = src.read(1)

r[r == src.nodata] = np.nan

fig, ax = plt.subplots(figsize=(15, 15))

rasterio.plot.show(r,
                   ax=ax,
                   vmin = 0,
                   vmax = 6000,
                   transform = src.transform)

comm_clip.plot(ax=ax, facecolor='none', edgecolor='red')
```

```
<AxesSubplot:>
```



7.3 Calculate total distance per fishing area

Rasterize each polygon in the shapefile that falls within the bounds of the raster data we are calculating statistics for.

We return a dictionary of indexed arrays, where each item corresponds to one polygon (fishing area). The array contains the indices of the original raster that fall within that fishing area.

```
with rasterio.open('Coastal_2020_12_masked.tif') as src:
    shape = src.shape
    transform = src.transform
    # read in the cropped raster
    r_array = src.read(1)
    # turn no data values into actual NaNs
    r_array[r_array == src.nodata] = np.nan
```

```
comm_3338['id'] = range(0,len(comm_3338))
```

```
crosswalk_dict = {}
for geom, idx in zip(comm_3338.geometry, comm_3338['id']):
    rasterized = features.rasterize(geom,
                                    out_shape=shape,
                                    transform=transform,
                                    all_touched=True,
                                    fill=0,
                                    dtype='uint8')
    # only save polygons that have a non-zero value
    if any(np.unique(rasterized)) == 1:
        crosswalk_dict[idx] = np.where(rasterized == 1)
```

```
/home/runner/.local/lib/python3.8/site-packages/rasterio/features.py:284: ShapelyDeprecationWa
  for index, item in enumerate(shapes):
```

Now we use the dictionary to calculate the sum of all of the pixels in the original raster that fall within each fishing area.

```
mean_dict = {}
# for each item in the dictionary
for key, value in crosswalk_dict.items():
    # save the sum of the indices of the raster to a new dictionary
    mean_dict[key] = np.nansum(r_array[value])
# create a data frame from the result
df = pd.DataFrame.from_dict(mean_dict,
    orient='index',
    columns=['distance'])
# extract the index of the data frame as a column to use in a join
df['id'] = df.index
```

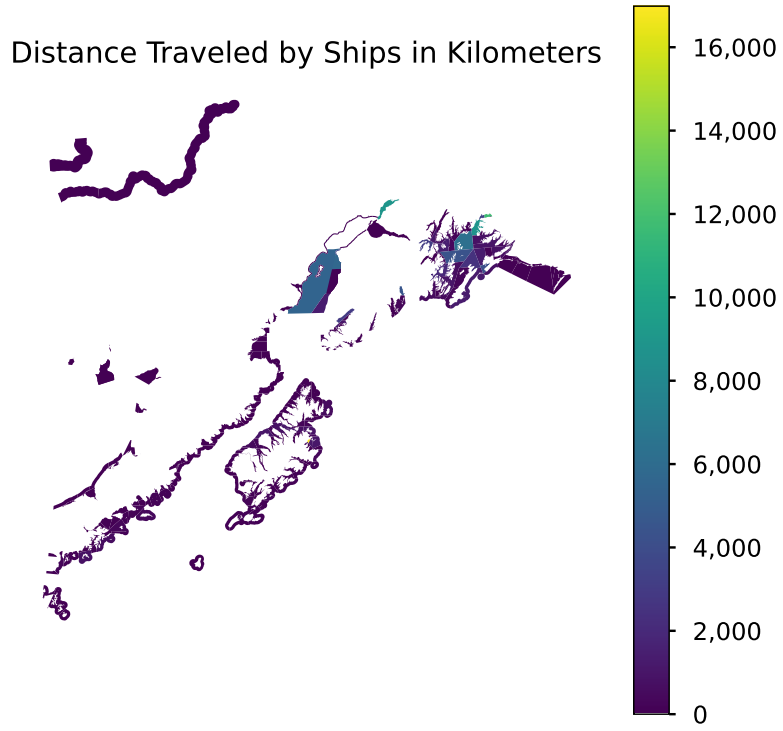
Now we join the result to the original geodataframe.

```
# join the sums to the original data frame
res_full = comm_3338.merge(df, on = "id", how = 'inner')
```

todo: Group by/summarize across another variable

```
fig, ax = plt.subplots(figsize=(7, 7))
plt.style.use("seaborn-talk")
ax = res_full.plot(column = "distance", legend = True, ax = ax)
fig = ax.figure
cb_ax = fig.axes[1]
cb_ax.set_yticklabels(["0", "2,000", "4,000", "6,000", "8,000", "10,000", "12,000", "14,000"
ax.set_axis_off()
ax.set_title("Distance Traveled by Ships in Kilometers")
plt.show()
```

```
/tmp/ipykernel_51354/3303410610.py:6: UserWarning: FixedFormatter should only be used together
  cb_ax.set_yticklabels(["0", "2,000", "4,000", "6,000", "8,000", "10,000", "12,000", "14,000"
```



8 Data Futures: Parquet and Arrow

- The difference between column major and row major data
- Speed advantages to columnnar data storage
- How `arrow` enables faster processing

8.1 Introduction

- open, seek, read, write, close - ways to access data
- difference between parquet and arrow
  - how paging and memory management works, blocks are organized by pages
  - on disk and in memory representation are the same
- column (parquet) vs row (csv) data example
- why column can give faster read speeds
- how arrow interacts with columnar data formats (like parquet)

8.2 Example

- show a read write example and benchmark maybe

References