nabu: A distributed, parallel, data processing platform

Antonio T. Lorenzo*, William F. Holmgren+
College of Optical Sciences*, Department of Hydrology & Atmospheric Sciences+, University of Arizona

THE UNIVERSITY OF ARIZONA

Introduction

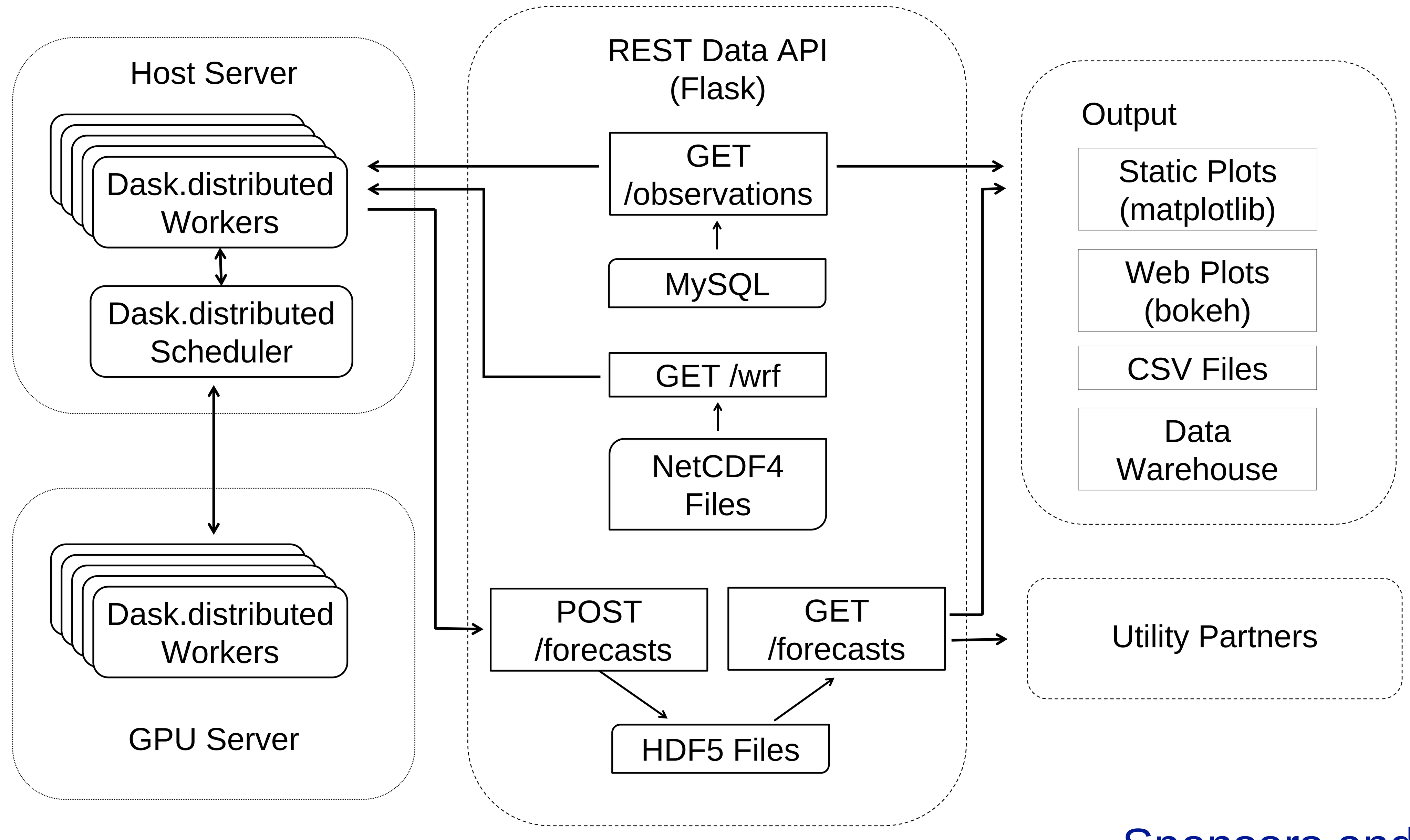
nabu is a tool designed to process data in parallel on distributed machines using Python. It will be used to generate operational solar and wind power forecast for electric utility companies in the Southwest US. These forecasts cover five minutes to seven days in the future requiring a blending of short-term, statistical forecasts with long-term, NWP forecasts, and forecasts will be regenerated every five minutes. The current forecast system relies on a large, monolithic python program that consumes 10 GB of memory and may take five minutes to produce forecasts (and plots). nabu will consume a fraction of the memory and wall time while also being easier to develop and debug.

Goals

- Generate blended power forecasts for ~50 solar and wind power plants along with aggregates for forecasts horizons from 5 minutes to 7 days
- Update forecasts every 5 minutes
- Leverage multi-core CPUs and a GPU to produce forecasts quickly
- Make the forecast generation code easy to understand, extend, and debug
- Make the forecast generation chain reproducible

Design

- Written in a functional style to improve reproducibility and make the program easier to understand
- Uses Dask.distributed which generates an optimized execution graph that can be executed on workers on many different machines with minimal modification to a serial code
- Utilizes Dask graphs to store provenance information for how a given forecast was generated
- Relies on a REST data API that enables access to raw data from nearly any host (an NFS server/mount would also work)



Helpful Tips

- Make sure NetCDF4/HDF5 files are chunked in the shape that best fits your access pattern. In our case, compression and re-chunking reduced the time it takes to retrieve a point forecast from a WRF NetCDF from almost a minute to tens of milliseconds
- The msgpack format with blosc compression is a fast serialization type for many python objects

Simplified Example

A simplified example of how nabu processes data is shown to the right. We define functions that get the data from the REST API, compute a solar power forecast from irradiance and weather data, compute a persistence forecast, compute a satellite image forecast, and combine the forecasts together then save the result. Each function is called in the blue box, but the `@delayed` ensures no computation is performed until the `client.compute` command. This command generates the computation graph shown below and sends it to the Dask.distributed scheduler. The scheduler assigns the tasks (nodes on the graph) to other worker processes intelligently to reduce data transfers between workers. We also specify that a worker with access to a GPU should be assigned the task to generate satellite forecasts.

```python
from dask import delayed
from distributed import Client


import requests
import pandas as pd
from pvlib.modelchain import ModelChain
from pvlib.pvsystem import retrieve_sam
from pvlib.tracking import SingleAxisTracker
from pvlib.location import Location


@delayed
def get_observations(params):
    """Go and get the observation data from the api"""
    return requests.get('http://127.0.0.1/observations',
                        params=params)

@delayed
def get_wrf_forecasts(params):
    """Get the WRF forecasts from the api"""
    return requests.get('http://127.0.0.1/wrf',
                         params=params)

@delayed
def get_satellite_data(params):
    """Get the satellite image from the api"""
    return requests.get('http://127.0.0.1/sat', params=params)

@delayed
def compute_power_fx(index, irradiance, weather):
    """
    Compute the power output of a single axis tracking
    solar power plant using pvlib and the irradiance
    and weather data
    """
    location = Location(32.1, -110.8, 'MST', 800, 'Tucson')
    module = retrieve_sam('sandiamod')[
        'Canadian_Solar_CS5P_220M___2009_']
    inverter = retrieve_sam('cecinverter')[
        'SMA_America__SC630CP_US_315V__CEC_2012_']
    system = SingleAxisTracker(
        module_parameters=module, inverter_parameters=inverter,
        modules_per_string=15, strings_per_inverter=300)
    mc = ModelChain(system, location)
    mc.run_model(index, irradiance=irradiance, weather=weather)
    return mc.ac

@delayed
def compute_persistence(observations, fx_index):
    """Compute a persistence forecast from the observations"""
    return pd.Series(observations.iloc[-1], index=fx_index)
```

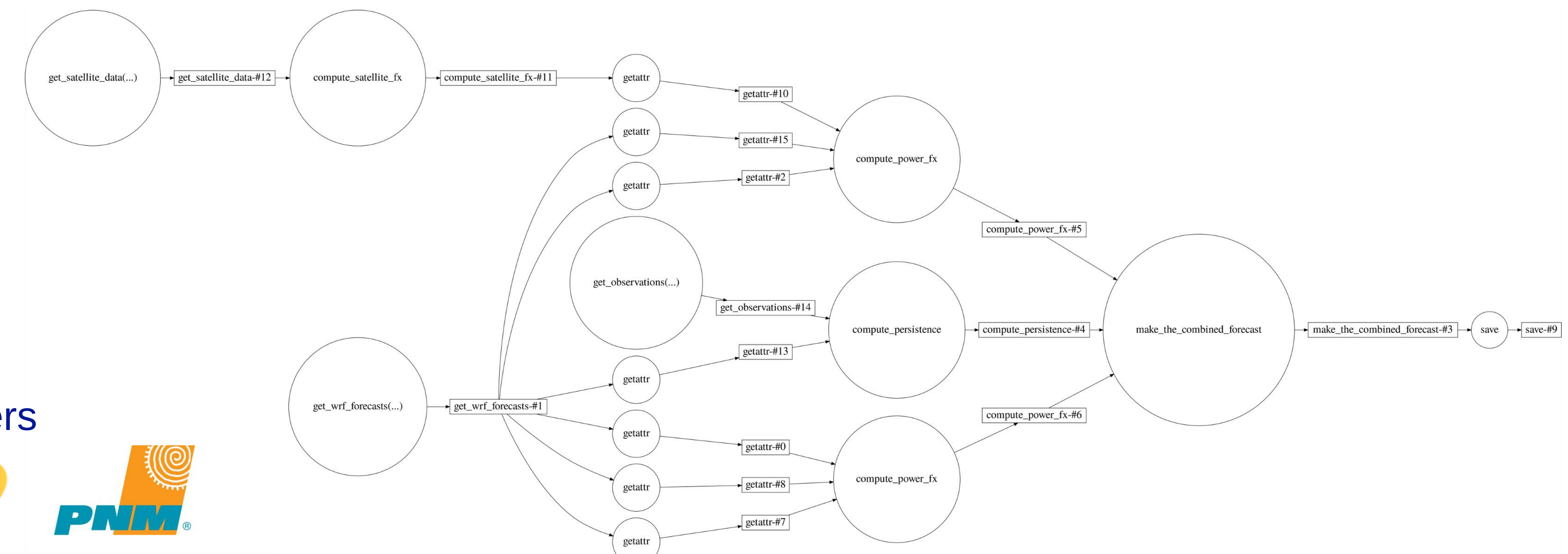
```python
@delayed
def compute_satellite_fx(sat_obs):
    """
    Compute a forecast given the latest satellite image.
    Parts of this rely heavily on linear algebra that
    can be done in 1/10 the time on a GPU if available.
    """
    return lots_of_matrix_manipulation()

@delayed
def make_the_combined_forecast(wrf_fx, sat_fx, persistence_fx):
    """
    Combine the forecasts that have different optimal forecast
    horizons in a smart way.
    """
    return some_clever_combination()

@delayed
def save(combined_fx):
    """
    Post the optimally combined forecast to the central API
    for storage and dissemination
    """
    requests.post('http://127.0.0.1/forecast',
                  data={'forecast': combined_fx})
```

```python
# Put it everything together, no processing or data collection
# occurs yet
obs = get_observations({'id': 100})
wrf_fx = get_wrf_forecasts({'model': 'UAGFS'})
sat_obs = get_satellite_data({'satellite': 'GOES-W'})
persistence_fx = compute_persistence(obs, wrf_fx.index)
wrf_power_fx = compute_power_fx(wrf_fx.index,
                                wrf_fx.irradiance,
                                wrf_fx.weather)
sat_fx = compute_satellite_fx(sat_obs)
sat_power_fx = compute_power_fx(wrf_fx.index,
                                sat_fx.irradiance,
                                wrf_fx.weather)
combined_fx = make_the_combined_forecast(wrf_power_fx,
                                         sat_power_fx,
                                         persistence_fx)
final = save(combined_fx)

# Now go ahead and run everything on our already setup
# dask.distributed workers and scheduler
# First connect to the scheduler that's already running
dask_client = Client('127.0.0.1:8686')
# Now go ahead and compute while making sure that the
# satellite forecast is computed by a worker with
# access to a GPU
dask_client.compute(final, workers={(sat_fx): 'GPU Worker'})
```



Sponsors and Partners

aps TEP PNM

RESEARCH, DISCOVERY & INNOVATION
Institute for Energy Solutions

https://forecasting.energy.arizona.edu