Parallel Computing On Pegasus: Python + Dask

David Grossman
Dr. Amelung's Lab

https://dask.org/

A Little Bit About Myself

1. Senior @ Arts & Science
2. Math & Computer Science majors
3. Physics & Economics Minor
4. Intern and future hire @ Coursera
5. 2 years @ Dr. Amelung's lab
6. Catch me surfing in the Bay Area starting next January!

davidgrossman1000@gmail.com

Wet Lab

https://github.com/2gotgrossman

The Problem

1. You write Python code, but it’s slow. You think, “it must be easy to parallelize!”
2. You write Python code, but the data doesn’t fit on 1 server. You think, “it must be easy to distribute the data across multiple servers!”
3. But after spending hours trying out different options, nothing works easily on Pegasus and it doesn’t provide significant speedup
4. Enter Dask

A Possible Solution
Why not submit multiple Pegasus jobs?

1. It is possible to break up 1 large job on Pegasus into a lot of smaller jobs.
2. However, there are downsides:
   a. What happens when a Pegasus job fails? You need to write retry code logic
   b. What happens when an exception is thrown? You need to parse job error files to find the issue
   c. You need to then write code to glue a bunch of output data back into 1 large output file

What is Dask?

1. “Dask provides advanced parallelism for analytics, enabling performance at scale for the tools you love”
2. Dask solves all the issues that the “multiple Pegasus job” solution created
   a. Dask has smart retry
   b. Code is executed from within Python and exceptions are thrown in the main Python script
   c. You need minimal code glue
3. Dask can be run on 1 server or on High Performance Computing cluster
4. Dask implements close equivalents to NumPy and Pandas DataFrames
5. Dask has built in Diagnostics (So cool!)

The Main Parts of Dask

1. Dask Collections
    a. Array, Bag, DataFrame
2. `dask.delayed`
    a. Build a sequence computations that are not executed immediately. Then execute all computation at once
3. `dask.distributed`
    a. "Submit" individual "futures" (think jobs) to a a scheduler that handles scheduling of execution
    b. This scheduler can operate on 1 machine or many
    c. What we will focus on

https://docs.dask.org/en/latest/

How This Talk Fits into the BIG Picture

1. Using Dask as a parallel framework
    a. Tim Norris gave a talk on Dask in general. Notes are here
2. Using Dask on Pegasus
    a. This talk
3. Using Dask Collections on Pegasus
    a. Future work. Ideally we can write Numpy code and then just change `import numpy` to `import dask.array` and it would ***just work***.
4. Using Dask as ***the*** way to run parallel code on Pegasus
    a. Future work. This would require CCS to support Dask for all Pegasus users.

Dask: Clusters and Clients

TODO

http://docs.dask.org/en/latest/setup/single-distributed.html

`dask.distributed: futures`

- `future = client.submit(f):`
    - returns a Future, which refers to a remote result. This result may not yet be completed. Eventually it will complete
- `res = future.result()`
    - Blocks until task completes and data arrives

```python
from dask.distributed import as_completed

futures = []
for i in range(10):
    future = client.submit(function, param1, param2, i)
    futures.append(future)

for future, result in as_completed(futures, with_results=True):
     # do something with output of each future
```

http://docs.dask.org/en/latest/futures.html

Using Dask: Let's Code!

1. Make sure you have the following packages installed:
    a. `pip install dask[complete]`
    b. `pip install numpy`
    c. `Pip install dask_jobqueue`

Sequential Code

```
import numpy as np
import math
from timeit import timeit

Y_DIM = 2000

def sequential_main():
    two_d_array = np.random.rand(10000, Y_DIM)

    vectorized_sqrt = np.vectorize(lambda x: math.sqrt(x))

    output = vectorized_sqrt(two_d_array)
    total = sum(sum(output))
    return total

print(timeit(stmt=sequential_main, number=1))
```

Parallel Code

```python
# My least favorite slide

# Sends data over the network

from dask.distributed import as_completed
from dask.distributed import LocalCluster, Client
import numpy as np
import math
from timeit import timeit

Y_DIM = 2000
NUM_JOBS = 4

def parallel_func(array):
    vectorized_sqrt = np.vectorize(lambda x: math.sqrt(x))
    output = vectorized_sqrt(array)
    total = sum(sum(output))
    return total
```

```python
# Submit parallel jobs
def distributed_main():
    two_d_array = np.random.rand(10000, Y_DIM)

    futures = []
    for i in range(NUM_JOBS):
        start = (i * Y_DIM) // NUM_JOBS
        end = ((i + 1) * Y_DIM) // NUM_JOBS
        future = client.submit(parallel_func,
                               two_d_array[:, start:end])
        futures.append(future)

    total = 0
    for future in as_completed(futures):
        total += future.result()

    print(total)
    client.close()
    return total


if __name__ == "__main__":
    cluster = LocalCluster(n_workers = 2,
                           threads_per_worker = 1)
    client = Client(cluster)
    print(timeit(stmt=distributed_main, number=1))
```

Parallel Code 2

```python
# My 2nd least favorite slide

# Doesn’t send data over the network

from dask.distributed import as_completed
from dask.distributed import LocalCluster, Client
import numpy as np
import math
from timeit import timeit

Y_DIM = 2000
NUM_JOBS = 4

def parallel_func2(dim):
    two_d_array = np.random.rand(10000, dim)
    vectorized_sqrt = np.vectorize(lambda x: math.sqrt(x))
    output = vectorized_sqrt(two_d_array)
    total = sum(sum(output))
    return total
```

```python
# Submit parallel jobs
def distributed_main2():
    ~~two_d_array = np.random.rand(10000, Y_DIM)~~

    futures = []
    for i in range(NUM_JOBS):
        ~~start = (i * Y_DIM) // NUM_JOBS~~
        ~~end = ((i + 1) * Y_DIM) // NUM_JOBS~~
        future = client.submit(parallel_func,
                               Y_DIM // NUM_JOBS)
        futures.append(future)

    total = 0
    for future in as_completed(futures):
        total += future.result()

    print(total)
    client.close()
    return total


if __name__ == "__main__":
    cluster = LocalCluster(n_workers = 2,
                                         threads_per_worker = 1)
    client = Client(cluster)
    print(timeit(stmt=distributed_main, number=1))
```

Dask on Pegasus: `dask_jobqueue`

1. `dask_jobqueue` allows you to execute Dask clusters on High Performance Computing supercomputers
   a. LSF, PBS, and other clusters are supported
2. `dask_jobqueue` allows you to define a worker (which is realized as a job that is submitted to the HPC queue)
   a. You can then replicate that worker many times

https://dask-jobqueue.readthedocs.io/

Dask on Pegasus: `dask_jobqueue`

The code barely changes:

```
cluster = LocalCluster()
```

becomes

```
from dask_jobqueue import LSFCluster
cluster = LSFCluster()
```

...well almost. The big idea is the same though. Only the cluster changes, but the client interface is the same

https://dask-jobqueue.readthedocs.io/en/latest/generated/dask_jobqueue.LSFCluster.html

Dask on Pegasus: `dask_jobqueue`

```python
from dask_jobqueue import LSFCluster
import sys

cluster = LSFCluster(queue='general',          # the queue on Pegasus
                     project='insarlab',        # your project name
                     cores=2,
                     mem='2GB',                 # unused by Pegasus but a required param
                     walltime='00:30:00',       # how long the worker will run for
                     interface = 'ib0',         # which network to use. NECESSARY PARAM
                     job_extra=
                        ['-R "rusage[mem=2500]"', # how to actually define memory usage
                         "-o WORKER-%J.out"],   # where to write worker output files
                     python_executable_location = sys.executable,
                     config_name='lsf')         # define your own config in a .yaml file

cluster.scale(20) # Create 20 workers with the parameters above
```

But we can hide this configuration in a `.yaml` config file

http://jobqueue.dask.org/en/latest/configuration-setup.html

Dask on Pegasus: Diagnostics

1. dask_jobqueue has a great way to visualize job execution. Here's how to do it:
    a. `ssh -L 8787:localhost:8787 dwg11@pegasus.ccs.miami.edu`
    b. Run your dask_jobqueue program
    c. Locally, open up http://localhost:8787/status in your browser
2. Check out the link at the bottom of the slide for a great tutorial

http://jobqueue.dask.org/en/latest/interactive.html#viewing-the-dask-dashboard

`dask_jobqueue`: Configuration

1. You can configure your `LSFCluster` paramaters in a `.yaml` file. These parameters can be used multiple times by all programs using Dask
    a. [Example]
2. Currently, `dask_jobqueue` can only be configured once. For example, if instantiating an LSF cluster, the "lsf" configuration in your `.yaml` file will be used
3. After the next release of `dask_jobqueue`, you will be able to define configurations custom to your specific script, not just your specific cluster
    a. [The idea] and [Updates on when the code will be released]

http://jobqueue.dask.org/en/latest/configuration-setup.html

Dask on Pegasus: Knowing the hardware

1. Each `general` queue node has the following properties
   a. 32 GB of memory
   b. 16 cores
   c. Try 1-2 GB per core
      i. Optimal parameters that I found are 2 cores per "worker" and about 2-4 GB of memory per worker
2. Pegasus supports the `ib0` network. Use it!!!
   a. This is the network over which data is transferred. The default is `eth0` which is significantly slower (10x or more)
3. I found that I can only scale to about 40 workers before some of those worker's jobs stay in a pending state

Dask on Pegasus: Let’s Try it Out!

TODO

Dask on Pegasus is in Use: PySAR

1. Dask is being used by PySAR today on Pegasus
    a. We have made a 5-15x speedup by running with Dask on Pegasus
2. The code and the configuration
3. @YunjunZhang is defending his thesis on PySAR today at 2pm. Go check it out!

http://jobqueue.dask.org/en/latest/interactive.html#viewing-the-dask-dashboard