

1

We can see that our computation gets faster as we have more workers who contribute to the task, while the actual predicted amount stays the same across trials.

Number of Cores	Runtime (ms)
1	2035.855200
2	1840.903200
4	1787.134400

6.S080 Lab 6

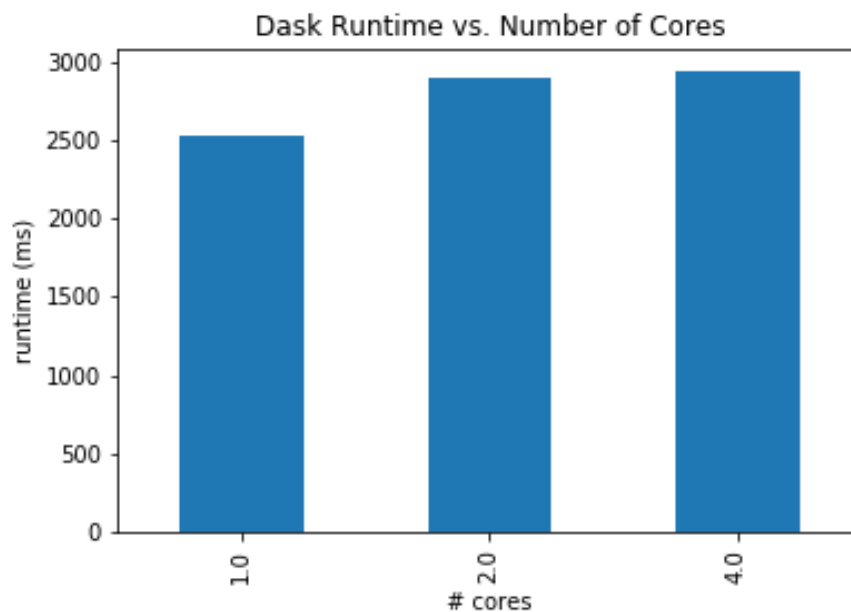
2

The rolling 5m ‘y’ average for the timeseries data looks like:

```
timestamp
2018-01-01 00:00:00    -0.263514
2018-01-01 00:00:01     0.215313
2018-01-01 00:00:02     0.120395
2018-01-01 00:00:03     0.289408
2018-01-01 00:00:04     0.314059
...
2018-01-31 23:59:55    -0.027162
2018-01-31 23:59:56    -0.028964
2018-01-31 23:59:57    -0.026947
2018-01-31 23:59:58    -0.020739
2018-01-31 23:59:59    -0.017693
Freq: S, Name: y, Length: 2678400, dtype: float64
```

The entire series can be found in `part2.ipynb`.

The pandas runtime is faster than the fastest dask runtime by about 300ms. This suggests that the dataset isn’t big enough for multiprocessing to make a difference. Additionally, the overhead of multiprocessing seems to outweigh the benefit because 2 & 4 cores take longer than 1 to run, and using dask even with 1 core takes more time than just using pandas. The runtime of using pandas including converting the table from dask is about the same as just using dask.



Operator	Runtime (ms)
dask (1 core)	2525.1510000016424
dask (2 cores)	2887.740099999064
dask (4 cores)	2937.811199997668
pandas (with dask conversion)	2776.6767000030086
pandas (computation only)	641.5956000018923

dask code:

```

n_workers = 1
runtime_df = pd.DataFrame(columns=['n_cores','runtime (ms)'])

while n_workers <= n_cores:
    # rescale cluster
    print('Resizing cluster to %s worker(s)...' % n_workers)
    cluster.scale(n_workers)
    time.sleep(1)

    # run calculation
    t_start = perf_counter()
    dd_df.y.rolling('5min').mean().loc['2018-01-01':'2018-01-31'].compute()
    t_stop = perf_counter()

    # save runtime to dataframe
    runtime = (t_stop - t_start)*1000
    runtime_df = runtime_df.append({'n_cores': n_workers, 'runtime (ms)': runtime},
                                   ignore_index=True)
    print('dask time (ms): ', runtime)

    # double desired # workers
    n_workers *= 2;

# plot results
runtime_df.plot(x='n_cores',y='runtime (ms)',kind='bar',legend=False)
plt.title('Dask Runtime vs. Number of Cores')
plt.xlabel('# cores')
plt.ylabel('runtime (ms)')
plt.savefig('images/q2.png')

```

pandas code:

```

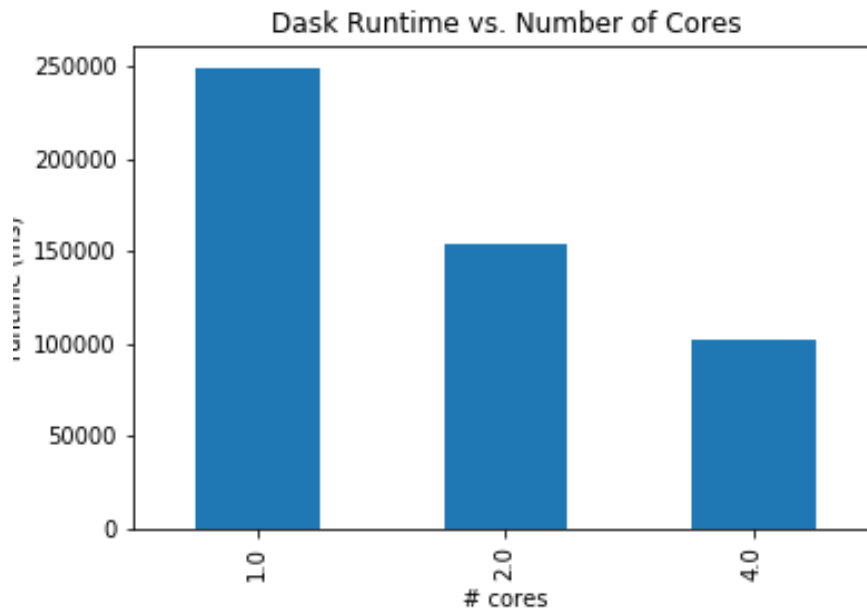
t_1 = perf_counter()
df = dd_df.compute()
t_2 = perf_counter()
df.y.rolling('5min').mean().loc['2018-01-01':'2018-01-31']
t_3 = perf_counter()
print('Pandas time for computation only(ms): %s' % ((t_3 - t_2)*1000))
print('Pandas time with table conversion (ms): %s' % ((t_3 - t_1)*1000))

```

6.S080 Lab 6

3

The top 2 notebook providers in August 2019 were **Github (374145 runs)** and **Gist (5894 runs)**. In this case, the dataset is large enough that multiprocessing does make a difference. Using 1 core takes significantly longer than 2 cores, which takes significantly more time than 4 cores.



Operator	Runtime (ms)
dask (1 core)	249008.9795999993
dask (2 cores)	153375.6652000011
dask (4 cores)	102789.95629999918

dask bag code:

```
# get data
urls = (db.read_text('https://archive.analytics.mybinder.org/index.jsonl')
        .map(json.loads)
        .pluck('name')
        .compute())
urls = ['https://archive.analytics.mybinder.org/' + u for u in urls]
notebook_runs = db.read_text(urls).map(json.loads)

runtime_df_q3 = pd.DataFrame(columns=['n_cores', 'runtime (ms)'])
n_workers = 1

while n_workers <= n_cores:
    # resize cluster
```

```

print('Resizing cluster to %s worker(s)...' % n_workers)
cluster.scale(n_workers)
time.sleep(1)

# run calculation
t_start = perf_counter()
(notebook_runs.filter(lambda record: record['timestamp'].startswith('2019-08'))
    .pluck('provider').frequencies(sort=True).take(2))
t_stop = perf_counter()

# save runtime to dataframe
runtime = (t_stop - t_start)*1000
runtime_df_q3 = (runtime_df_q3.append({'n_cores': n_workers, 'runtime (ms)':
    runtime}, ignore_index=True))
print('dask time (ms): ', runtime)

# double desired # workers
n_workers *= 2

```

4

Our code can be found in `filter.py` and filters out urls based on whether their `status` has the value `success`, which ends up being **all urls**. Using different amounts of workers outputs the same result, but allows for speedup.

Workers	Time	Status
1	4274	success = 144000
2	4046	success = 144000
4	2385	success = 144000

Erica Zhou (*ezhou*) & Katharina Gschwind (*gschwind*)
Repo: <https://github.com/ericazhou7/6.s080-lab6>
Commit Hash: *61e0274550b2fbb943c38a1fcae0b8afb64f231b*

13 November 2019

6.S080 Lab 6

5

N/A

6

We use the dataset of crimes on London streets found at: <https://data.police.uk/api/crimes-street-dates>. In `crime.py`, we calculate how often streets are mentioned in these monthly aggregates of crime at this url, and what the worst street by that metric is. We use multiprocessing to answer this question. We identify that (one of) the worst street(s, since many streets end up tying) is **Bedfordshire** with 36 many stop-and-searches (where stop-and-searches are the type of crimes reported within this specific data set).