Concurrency in Python

We can speed up the execution of code by taking advantage of the multiple cores available in modern computers. We will learn how to speed up the execution of embarrassingly parallel problems. A problem is called embarrassingly parallel if it can be broken into smaller pieces that have little to no dependency between each another.

1 Multiple Python Processes

Consider a function that takes some time to execute:

```
import time


def slow_computation(x):
    print(f"Computing {x}")
    time.sleep(2)
    return x
```

If we need to execute this function several times:

```
import numpy as np

start = time.perf_counter()
results = np.zeros(10)
for i in range(10):
    results[i] = slow_computation(i)
print(f"Total time: {time.perf_counter() - start:.2f} seconds")
```

The entire process would take about 20 seconds to finish.

Notice that this problem is embarrassingly parallel. Indeed, each iteration in the loop is independent from one another. We could execute the loop in a random order and we would still obtain the same results. Also, each iteration in the loop takes a few seconds to finish. When you see a problem of this type, its solution can potentially benefit from parallelization.

We can parallelize Python code using the built-in module `concurrent.futures`. This module implements a class named `ProcessPoolExecutor`, which is an interface for submitting callables (like functions) to be executed at different processes. That is, we can use `ProcessPoolExecutor` to launch several Python processes, and then send each iteration of the loop above to a different process. The interface takes care of getting the results back on the Python that launched the processes.

On this first example, we will use the high-level method of `ProcessPoolExecutor` named `map`. The `map` method takes a function and applies it to several values stored in an iterable (like a list). However, the function is executed asynchronously and concurrently on different Python processes (usually one per core). Let's modify the example above:

```
from concurrent.futures import ProcessPoolExecutor

start = time.perf_counter()
with ProcessPoolExecutor() as executor:
    print("Starting parallel computation")
    results = executor.map(slow_computation, range(20))
print("Obtained the generator filled with values")
# We can now call next on the generator to obtain the results
# one by one, or call list on the generator to get all the
# values in a list.
print(list(results))
print(f'Total time: {time.perf_counter() - start:.2f}')
```

The variable `results` is a generator. We can call `next` on it to obtain the value of each computation, or call `list` to get all of the values back in a list. After obtaining all values, the generator is used up and there are no more values to give.

In my computer I have 2 cores, so we would expect the loop to finish in about 10 seconds, implying in a 50% performance improvement over the single-core version. Since there is some communication involved between the Python to processes to collect the results, it will take a little more than half the time to actually finish the job.

Notice that even though the order of execution is random, the order of the results is the same as the order of the iterable.

The `with` is used with `ProcessPoolExecutor`, which also provides a context manager (implements `__enter__` and `__exit__`). The `__exit__` method of the `ProcessPoolExecutor` instance, will call the method `shutdown(wait=True)`. This method will wait for all the processes to finish executing before continue executing the program.

If an exception is raised in one of the processes, it will pop up when we are calling `next` on the generator. For example:

```
def slow_computation_with_exceptions(x):
    print(f"Computing {x}")
    time.sleep(2)
    if x == 2:
        raise ValueError("x==2")
    return x


with ProcessPoolExecutor() as executor:
    results = executor.map(slow_computation_with_exceptions,
                           range(5))

type(results)
next(results)
next(results)
next(results)
# ValueError is raised
next(results)
# StopIteration is raised
next(results)
```

After an exception is raised in a generator, the generator stops producing new values and starts raising `StopIteration` if `next` is called again.

2 Futures

The name of the module is concurrent.futures, but we have not seen any object of the type Future. These objects are used by the .map method. A Future class represents a computation that is happening asynchronously, and which may or may not have yet completed (similar to a Promise in JavaScript). When we call ProcessPoolExecutor().map, several Future instances are being created. Each instance has access to the function and value to be used in a computation. These objects can then be put in a queue and distributed to the different Python processes. In other words, the Future objects serve as the interface to communicate with the main Python process. These objects also provide methods to check whether they have completed the computation they were assigned.

The ProcessPoolExecutor().map method creates the Future instances, puts them in a queue, checks when they are completed and retrieves the results. The completion check actually works a little differently. Instead of checking explicitly whether a Future instance is done, we let the instance do something when it finishes. This is known as a callback. The Future class has a add_done_callback method, which takes a function to be executed when the Future is done.

We will modify our example to create the Future instances explicitly. Then, we will use the as_completed method to get the results as the futures finish executing. This gives us more flexibility, since we can schedule different functions and handle errors better than with map.

```python
from concurrent import futures

# max_workers is how many cores we want to use at maximum
# in my laptop there are only 2
with futures.ProcessPoolExecutor(max_workers=2) as executor:
    # Schedule all jobs
    print("Scheduling jobs")
    # Next, create several Future instances:
    to_do = []
    for x in range(10):
        # Schedule slow_computation to be executed with the
        # input x
        future = executor.submit(slow_computation, x)
        # future is a Future instance, and it represents
        # the asynchronous execution of slow_computation(x)
        # Store all futures in a list:
        to_do.append(future)
        print(f'Scheduled {x} on {future}')

    print("Retrieve results as they are completed")
    results = []
    iterator = futures.as_completed(to_do)
    # futures.as_completed returns an iterator
    # This iterator yields a Future instance when it completes.
    # That is, calling next(iterator) will yield the first
    # Future that completed.
    # Calling next(iterator) will yield the second Future that
    # completed, and so on.
    for future in iterator:
        # If we are inside this block, it is because one of
        # the futures has completed.
        # We can get the results:
        res = future.result()
        print(f'Result of {future}: {res}')
        results.append(res)

print(results)
```

Notice that as soon as we `executor.submit` a job, it starts executing. In fact, even before we are able to print the "Scheduled" message, we get a message saying the function started executing. When we finish scheduling, we call `futures.as_completed` and get an iterator. We then iterate over the results of this iterator. We get back the futures as they are completed. We get back the first futures quickly, but then we need to wait a few seconds for the next future to return. Observe that the futures are scheduled and returned in a random order, so the variable `results` has the results in a random order.

3 Ordering Results

The `map` method waits for all futures to finish running, then sorts their order and then creates the generator, which is returned to us. We can also do that when submitting the futures with `executor.submit`, we just need to keep track of the jobs ourselves.

We will modify the variable `to_do` from list to a dictionary. We will use each `Future` instance as a key in the dictionary. And each value of `x` as the value associated with the key. This is because when we call `futures.as_completed`, we get back a `Future`. We can use this future to find out what was the input to the function, and then order the results using the input.

```
from concurrent import futures

with futures.ProcessPoolExecutor(max_workers=2) as executor:
    print("Scheduling jobs")
    to_do = {}
    total_jobs = 10
    for x in range(total_jobs):
        future = executor.submit(slow_computation, x)
        to_do[future] = x
        print(f'Scheduled {x} on {future}')

    results = np.zeros(total_jobs)
    for future in futures.as_completed(to_do.keys()):
        # We do not know the order of execution, so we do not
        # know which future we are getting back.
        # But, now that we have the to_do dictionary, we can
        # find out.
        x = to_do[future]
        res = future.result()
        print(f'Result of {future}: {res}')
        results[x] = res
```

```
print(results)
```

4 Progress Display

We can use the tqdm package to create a progress bar during the execution of our script. First, install it with pip:

```
pip install tqdm
```

There are two main interfaces. If you are using Python on a terminal, you can import the function `tqdm` from the package: `from tqdm import tqdm`. If you are using Python on a Jupyter notebook, then you import the function `tqdm_notebook`: `from tqdm import tqdm_notebook`. Both functions take an iterable as an argument, and return another iterable. The iterable it returns will print on the screen a progress bar. Each time the iterable yields, the progress bar is updated.

The package uses `print` to create the progress bar. So, if we are printing things inside the job, the progress bar will be printed on many lines.

```
from tqdm import tqdm_notebook

for i in tqdm_notebook(range(10), total=10):
    time.sleep(1)
```

Let's modify our script to display a progress bar. First, redefine the `slow_computation` function so that it does not print on the screen:

```
def slow_computation_silent(x):
    time.sleep(2)
    return np.exp(x) + 3
```

Now, use `tqdm_notebook` to create a progress bar:

```
total_jobs = 10
with futures.ProcessPoolExecutor(max_workers=2) as executor:
    print("Scheduling jobs")
    to_do = {}
    for x in range(total_jobs):
        future = executor.submit(slow_computation_silent, x)
        to_do[future] = x
        print(f'Scheduled {x} on {future}')

    results = np.zeros(total_jobs)
    iterator = tqdm_notebook(futures.as_completed(to_do.keys())
       ,
                             total=total_jobs)
    for future in iterator:
        x = to_do[future]
        results[x] = future.result()
```

```
for x, res in enumerate(results):
    print(f'{x}: {res}')
```

5 Handling Errors

With `map` we could handle errors up to a point, since when we found an exception the generator would stop giving out new values. However, by scheduling the callables ourselves we can handle errors.

Consider a function that will raise an exception on negative inputs:

```
import math


def comp(x):
    time.sleep(2)
    return math.log(x)
```

The call to `math.log` can fail depending on the input. When that happens we will take note of the times it failed, but continue to get the results from the rest of the submissions:

```
inputs = 2*np.random.random(10) - 1
total_jobs = len(inputs)

with futures.ProcessPoolExecutor(max_workers=2) as executor:
    # scheduling jobs
    print("Scheduling jobs")
    to_do = {}
    for i, val in enumerate(inputs):
        future = executor.submit(comp, val)
        to_do[future] = i
        print(f'Job {i} scheduled on {future}: {comp.__name__
           }({val})')

    results = {}
    fails = {}
    for future in futures.as_completed(to_do.keys()):
        i = to_do[future]
        try:
            value = future.result()
        except ValueError as err:
            fails[i] = err.args
        else:
            results[i] = value

print('Successfully Completed:')
for i, val in results.items():
    print(f'Job {i} computed {comp.__name__}({inputs[i]}) = {
       val}')
print('Failed Jobs:')
for i, val in fails.items():
    print(f'Job {i} failed at computing {comp.__name__}({val})
       ')
```

6 Example: Time Series Panel

We will now consider a complete example of parallelization with real data.

6.1 Loading the Panel

Let's start by loading a time series panel. First, obtain the data from this repository. Extract the files to a folder you can quickly find. In my case, I extracted the files to the folder stored in the `path` variable below. We will start by loading this data into Python, cleaning it and organizing it in a single data frame.

Obtain the names of the files we extracted:

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

path = ('/Users/guilhermesalome/'
        'Teaching/Duke/Econ890 Python - 2019/supporting/data/
           StocksHF/')
files = [f for f in os.listdir(path) if '.csv' in f]
files
```

Now, load one of the files to inspect the data:

```
d = pd.read_csv(path + files[0], skiprows=0, header=None)
d
d = d.iloc[:, 2]
d.name = files[0].strip('.csv')
d
```

Do the same for a second file:

```
e = pd.read_csv(path + files[1], skiprows=0, header=None)
e = e.iloc[:, 2]
e.name = files[1].strip('.csv')
e
```

Now, let's concatenate two series:

```
pd.concat(objs=[d, e], axis=1)
```

We get a `DataFrame`, where each column represents the price for a given stock. Notice that the columns have appropriate names, because we added names to our `Series` objects before the concatenation.

Since we want to load multiple files, let's create a function that will do the job of loading and cleaning the data, just like we did above:

```
def extract(folder, filename):
    df = pd.read_csv(folder + filename,
                     skiprows=0,
                     header=None,
                     usecols=[2],
                     names=[filename.strip('.csv')])
    return df

# Test it:
```

```
extract(path, files[0])
```

Now, load the common indices for the panel:

```
date_times = pd.read_csv(path + files[0], skiprows=0, header=
    None, usecols=[0, 1])
dt = pd.to_datetime(date_times[0]*10**4+date_times[1], format
    ='%Y%m%d%H%M')
dt.name = 'Time'
dt
```

Concatenate all of the files and add the time indices:

```
panel = pd.concat([extract(path, f) for f in files], axis=1)
panel.index = dt
panel
```

Compute geometric returns, but only intraday:

```
np.log(panel).diff()
```

The above is not what we want, since includes overnight returns. We need to group the prices by day, and then compute the returns, this way we avoid computing returns across two different days.

```
returns = np.log(panel).groupby(panel.index.date).diff()
returns
```

Since we do not compute the overnight returns, for each day we should have a `NaN` at the very first row. We can verify that is indeed the case:

```
returns.iloc[0::386]
# Drop the NaNs
returns = returns.dropna()
# Now the first values are at 9:36 AM
returns.iloc[0::385]
returns
```

6.2 Computing the Statistic of Interest

We want to compute confidence intervals for a certain statistic, which is computed using intraday data. The statistic is computed for each day and for each stock.

Let's build a function that will compute the statistics given a panel of data:

```
def computeRV(returns):
    return returns.groupby(returns.index.date).apply(lambda x:
        (x**2).sum())
```

Notice that the result of the `groupby` and `apply` is a new `DataFrame`. The index of this `DataFrame` is given by `returns.index.date`.

```
start = time.perf_counter()
RV = computeRV(returns)
stop = time.perf_counter()
print(f'Elapsed Time: {stop - start:.2f} seconds')
```

Computing the statistic just once for the entire panel takes a significant time. Imagine what would happen if the panel had all the constituents of the S&P500 index, or if we have data spanning the past two decades. In my case, it took about 1.51 seconds. If we want to compute confidence intervals for this statistic via bootstrapping, it would take at least 1510 seconds (with 1000 bootstrap samples), or about 25 minutes. If we have multiple cores available, then we can speed up the execution of this task.

6.3 Generating Bootstrap Samples

To build the code, we select a smaller sample to work with. Test it, and then scale it to the full sample.

Let's start with just 1 month and 2 stocks:

```
test = returns.loc['2007-01-01':'2007-01-31', ['CSCO', 'BA']].
   copy()
```

To create a bootstrap sample from this data (random sample with replacement), we can use the `sample` method of `DataFrame`:

```
test.sample(n=test.shape[0], replace=True)
```

The only issue is that our statistic is computed using intraday data, so we need to sample for each day separately:

```
new_sample = test.groupby(test.index.date).apply(
    lambda group: group.sample(n=group.shape[0], replace=True))
new_sample
```

We can drop the first level of the `MultiIndex`:

```
new_sample.droplevel(0, axis=0)
```

Alternatively, we can use the `groupby` option `group_keys=False`:

```
new_sample = test.groupby(test.index.date, group_keys=False).
   apply(
    lambda group: group.sample(n=group.shape[0], replace=True))
new_sample
```

Let's define a function that computes generates the bootstrap sample:

```
def bootstrap_returns(returns):
    groups = returns.groupby(returns.index.date, group_keys=
       False)
    return groups.apply(lambda group: group.sample(n=group.
       shape[0], replace=True))
```

We can combine `computeRV` and `bootstrap_returns` to compute the statistic day by day:

```
computeRV(bootstrap_returns(test))
```

At this point we can work on running this statement in parallel. This is an embarrassingly parallel problem, since each bootstrap samples and statistic computation is independent from one another. However, at the end of the computations we will have several data frames. One data frame for each bootstrap sample. And we need to create a function that will aggregate all of the results and compute the confidence intervals.

6.4 Computing the Confidence Intervals

Consider you have three data frames containing the value of the statistic for two bootstrap samples:

```
df1 = computeRV(bootstrap_returns(test))
df2 = computeRV(bootstrap_returns(test))
df3 = computeRV(bootstrap_returns(test))
df1.values
```

We want to compute percentiles for each day and each stock. Thus, we need to traverse all of the data frames (unfortunately there is no way around this). Let's save everything into a multi-dimensional matrix:

```
all_stats = np.array((df1.values, df2.values, df3.values))
all_stats.shape
all_stats
```

We can use the `np.quantile` function to compute the quantile along one axis. The axis we want to use is the first one, so that it traverse all the different matrices to compute the quantile for one day and one stock. The axis input of `np.quantile` is what will be changed when finding the quantile. In this case, `all_stats.shape` is `(3, 20, 2)`, where 3 is the number of data frames (or matrices in this case). We want to compute the percentile along 0th axis, while having fixed the values for the 1st and 2nd axes.

To get the upper bound of the 99% confidence interval for the statistic:

```
print(np.quantile(all_stats, 0.995, axis=0))
```

For the lower bound:

```
print(np.quantile(all_stats, 0.005, axis=0))
```

Now we can take these values back to a dataframe that will hold the results. We need will build 2 data frames, one for the lower bound and one for the upper bound. We will create them with an appropriate `MultiIndex`. And then merge all in a single data frame:

```
cols = df1.columns.values
cols
list(zip(cols, ['lower']*len(cols)))
cols_lower = pd.MultiIndex.from_tuples(zip(cols, ['lower']*len(
   cols)))
cols_lower
lower = pd.DataFrame(data=np.quantile(all_stats, 0.005, axis=0)
   ,
                     index=df1.index, columns=cols_lower)
lower
```

We repeat the process for the upper bounds:

```
cols_upper = pd.MultiIndex.from_tuples(zip(cols, ['upper']*len(
   cols)))
upper = pd.DataFrame(data=np.quantile(all_stats, 0.995, axis=0)
   ,
                     index=df1.index, columns=cols_upper)
upper
```

We can now merge both data frames:

```
ci = pd.merge(left=lower, right=upper, left_index=True,
   right_index=True)
ci
# Do a sort on the columns first hierarchy
ci = ci.sort_index(axis=1, level=0)
ci
```

We can now define an aggregating function that generates the confidence intervals:

```
def getCI(df_iterable):
    all_stats = np.array([df.values for df in df_iterable])
    # Obtain column names
    cols = df_iterable[0].columns.values
    # Create MultiIndex for the lower bound
    cols_lower = pd.MultiIndex.from_tuples(zip(cols, ['lower']*
       len(cols)))
    lower = pd.DataFrame(data=np.quantile(all_stats, 0.005,
       axis=0),
                            index=df1.index, columns=cols_lower)
    # Create MultiIndex for the upper bound
    cols_upper = pd.MultiIndex.from_tuples(zip(cols, ['upper']*
       len(cols)))
    upper = pd.DataFrame(data=np.quantile(all_stats, 0.995,
       axis=0),
                            index=df1.index, columns=cols_upper)
    # Merge both frames
    ci = pd.merge(left=lower, right=upper, left_index=True,
       right_index=True)
    # Do a sort on the columns first hierarchy
    return ci.sort_index(axis=1, level=0)


# Test function
getCI([computeRV(bootstrap_returns(test)) for _ in range(5)])
```

6.5 Solution in a Single Core

Let's time the solution in a single core:

```
print("Solution in a Single Core")
print("|{:^15s}|{:^15s}|".format("Samples", "Time (s)"))
print("|{}|{}|".format('-'*15, '-'*15))
single_core_time = {}
for bsamples in [1, 5, 10, 50, 100, 200, 500, 1000]:
    start = time.perf_counter()
    ci = getCI([computeRV(bootstrap_returns(test)) for _ in
       range(bsamples)])
    stop = time.perf_counter()
    total_time = stop - start
    single_core_time[bsamples] = total_time
    print(f"|{bsamples:^15.0f}|{total_time:^15.2f}|")
```

6.6 Solution in Parallel

We will use Futures to schedule the computations on different cores. Let's create the function that we will submit to each core. It will take a DataFrame containing the data, generate a single bootstrap sample and output the statistic:

```
def sample_to_statistic(data):
    return computeRV(bootstrap_returns(data))
```

Let's create another function that uses sample_to_statistic several times, and then returns the confidence intervals for the statistic:

```
def computeCI(data, bsamples):
    with futures.ProcessPoolExecutor(max_workers=2) as executor
       :
        to_do = []
        # the order does not matter, so to_do does not need to
           be a dict
        for i in range(bsamples):
            future = executor.submit(sample_to_statistic, data)
            to_do.append(future)
    return getCI([f.result() for f in to_do])
        results = []
        for future in futures.as_completed(to_do):
            results.append(future.result())

    return getCI(results)
```

Let's time the results using two cores:

```
many_cores_time = {}
for bsamples in tqdm_notebook([1, 5, 10, 50, 100, 200, 500,
   1000]):
    start = time.perf_counter()
    ci = computeCI(test, bsamples)
    stop = time.perf_counter()
    many_cores_time[bsamples] = stop - start
```

6.7 Compare Results

Let's save the results in a DataFrame:

```
time_single = pd.DataFrame(data={'Single Core': list(
   single_core_time.values()),
                                 'Two Cores': list(
                                    many_cores_time.values())},
                           index=list(single_core_time.keys()))
time_single
```

We can now easily visualize the results:

```
ax = time_single.plot(kind='line',
                      marker='o',
                      color=['black', 'green'],
                      linewidth=0.5)
```

```
ax.grid(True, linestyle='--', alpha=0.8)
ax.set(xlabel='Samples', ylabel='Time (seconds)')
```

6.8 Scaling to the Full Sample

The results indicate that using two cores instead of one reduces the total execution time in a significant amount. We can further reduce the execution time by running the code in a computer with more cores. We will discuss two options in detail in the next lecture.

Based on the results in my laptop, it took about 40 seconds to finish computing the CI for 2 stocks on a single month. The full sample has 12 months and 12 stocks, so the code would take about 6*12*40 seconds, or about 48 minutes. This is using two cores.

The only issue remaining has to do with concatenating the several dataframes to compute the quantiles. When we scale to the full sample, this might lead to memory issues. If we are working on a computer with limited resources, we will need to take that into consideration, and break the problem in pieces. If we use a computer with more resources, then the code as is should be fine.

7 Compiling Python

The project Numba provides a just-in-time compiler for Python. The idea of the project is to transform Python code into optimized machine code. It also supports parallelization. However, the project is limited to a subset of the core language and the NumPy package. This means that most of the code using Pandas will not be able to be converted. Numba is useful when you have code that is mainly based on Numpy.

You can install Numba with `pip`:

```
pip install numba
```

The package provides a decorator `njit`. We use `njit` to make Numba try to compile a function into machine code. Consider a function designed to work in a single core.

```
import numpy as np

def comp_with_np(n):
    X = np.random.random(size=(n, n))
    return X.mean()
```

Let's time its execution:

```
import time

start = time.perf_counter()
mode = comp_with_np(10000)
print(f"Total Time: {time.perf_counter() - start:.2f} seconds")
```

We can now apply the `njit` decorator, which will attempt to compile the entire function into machine code.

```python
from numba import njit

comp_with_np = njit(comp_with_np)
```

We need to call the function once to have it be compiled. It is then stored in the memory, and the next time we want to use it, it will already be compiled.

```python
# compile with numba
comp_with_np(100)
```

```python
start = time.perf_counter()
ci = comp_with_np(10000)
print(f"Total Time: {time.perf_counter() - start:.2f} seconds")
```

While the project has its many merits, the functionality is very limited. To have your code compiled, you can only use the supported functions. If you are working with the common features of Numpy, then using Numba will possibly increase the speed of your scripts. If it does not work, then you already know how to do parallel processing.

8 Assignment

Problem 1 *Study the presentation: Types of Concurrency by Raymond Hettinger. Answer the questions:*

- *Understand the differences between threads, multiprocessing and asynchronous concurrency.*
- *What are the advantages of threads? What are its disadvantages?*
- *What are the rules to follow when using threads?*
- *Why use multiprocessing?*
- *Is it easy to start with asynchronous programming? Why would you use it?*

Problem 2 *(Optional)*

Consider a deterministic growth model, where an agent decides between consumption (c_t) and investment in capital (k_t), while maximizing his utility. We can write this problem as:

$$\max \sum_{t=0}^{\infty} \beta^t U(c_t)$$

$$\textit{subject to} \begin{cases} k_{t+1} = k_t^{\alpha} - c_t + (1-\delta)k_t, \forall t >= 0 \\ k_0 > 0 \end{cases}$$

Write the problem as a Bellman equation. Let $U(c;\sigma) = \frac{c^{1-\sigma}-1}{1-\sigma}$. Obtain the Euler equation for this problem in terms of the consumption c. Solve the problem by Value Function Iteration. Consider $\sigma = 2$, $\beta = 0.95$, $\delta = 0.1$ and $\alpha = 0.33$. Use the steady state value of k to create a grid for the possible values of k, say 100 points between $0.25k^*$ and $1.75k^*$. Start with a guess for V over the grid, for example $V(k) = 0$ for all k in the grid. Use a minimization function to solve for k. You may want to add the constraint that c should always be positive.

Use concurrency to speed up the solution of this problem. What can be easily run in parallel in this problem? Is this problem embarrassingly parallel? If not, are there subsets of the problem that are embarrassingly parallel?

Graph the time it takes to find the solution of the problem when the number of grid points increases. Compare the time using regular loops and using concurrency.