Parallel Processing in Python – A Practical Guide with Examples

by *Selva Prabhakaran (https://www.machinelearningplus.com/author/selva86/)* | *Posted on October 31, 2018 (https://www.machinelearningplus.com/python/parallel-processing-python/)*

Parallel processing is a mode of operation where the task is executed simultaneously in multiple processors in the same computer. It is meant to reduce the overall processing time. In this tutorial, you'll understand the procedure to parallelize any typical logic using python's multiprocessing module.

Contents

1. Introduction

Parallel processing is a mode of operation where the task is executed simultaneously in multiple processors in the same computer. It is meant to reduce the overall processing time.

However, there is usually a bit of overhead when communicating between processes which can actually increase the overall time taken for small tasks instead of decreasing it.

In python, the `multiprocessing` module is used to run independent parallel processes by using subprocesses (instead of threads). It allows you to leverage multiple processors on a machine (both Windows and Unix), which means, the processes can be run in completely separate memory locations.

By the end of this tutorial you would know:

- How to structure the code and understand the syntax to enable parallel processing using `multiprocessing` ?
- How to implement synchronous and asynchronous parallel processing?
- How to parallelize a Pandas DataFrame?
- Solve 3 different usecases with the `multiprocessing.Pool()` interface.

The maximum number of processes you can run at a time is limited by the

Feedback

2. How many maximum parallel processes can you run?

number of processors in your computer. If you don't know how many processors are present in the machine, the `cpu_count()` function in `multiprocessing` will show it.

```
import multiprocessing as mp
print("Number of processors: ", mp.cpu_count())
```

3. What is Synchronous and Asynchronous execution?

In parallel processing, there are two types of execution: Synchronous and Asynchronous.

A synchronous execution is one the processes are completed in the same order in which it was started. This is achieved by locking the main program until the respective processes are finished.

Asynchronous, on the other hand, doesn't involve locking. As a result, the order of results can get mixed up but usually gets done quicker.

There are 2 main objects in `multiprocessing` to implement parallel execution of a function: The `Pool` Class and the `Process` Class.

1. `Pool` Class
   1. Synchronous execution
      - `Pool.map()` and `Pool.starmap()`
      - `Pool.apply()`
   2. Asynchronous execution
      - `Pool.map_async()` and `Pool.starmap_async()`
      - `Pool.apply_async()`)
2. `Process` Class

Let's take up a typical problem and implement parallelization using the above techniques. In this tutorial, we stick to the `Pool` class, because it is most convenient to use and serves most common practical applications.

4. Problem Statement: Count how many numbers exist between a given range in each row

The first problem is: Given a 2D matrix (or list of lists), count how many numbers are present between a given range in each row. We will work on the list prepared below.

Tags

Feedback

```
import numpy as np
from time import time

# Prepare data
np.random.RandomState(100)
arr = np.random.randint(0, 10, size=[200000, 5])
data = arr.tolist()
data[:5]
```

Solution without parallelization

Let's see how long it takes to compute it without parallelization. For this, we iterate the function `howmany_within_range()` (written below) to check how many numbers lie within range and returns the count.

```
# Solution Without Paralleization

def howmany_within_range(row, minimum, maximum):
    """Returns how many numbers lie within `maximum` and `minimum` in a given `row`"""
    count = 0
    for n in row:
        if minimum <= n <= maximum:
            count = count + 1
    return count

results = []
for row in data:
    results.append(howmany_within_range(row, minimum=4, maximum=8))

print(results[:10])
#> [3, 1, 4, 4, 4, 2, 1, 1, 3, 3]
```

5. How to parallelize any function?

The general way to parallelize any operation is to take a particular function that should be run multiple times and make it run parallelly in different processors.

To do this, you initialize a `Pool` with n number of processors and pass the function you want to parallelize to one of `Pool`s parallization methods.

`multiprocessing.Pool()` provides the `apply()`, `map()` and `starmap()` methods to make any function run in parallel.

Nice! So what's the difference between `apply()` and `map()`?

Both `apply` and `map` take the function to be parallelized as the main argument. But the difference is, `apply()` takes an `args` argument that accepts the parameters passed to the 'function-to-be-parallelized' as an argument, whereas, `map` can take only one iterable as an argument.

So, `map()` is really more suitable for simpler iterable operations but does the job faster.

Feedback

We will get to `starmap()` once we see how to parallelize `howmany_within_range()` function with `apply()` and `map()`.

5.1. Parallelizing using Pool.apply()

Let's parallelize the `howmany_within_range()` function using `multiprocessing.Pool()`.

```python
# Parallelizing using Pool.apply()

import multiprocessing as mp

# Step 1: Init multiprocessing.Pool()
pool = mp.Pool(mp.cpu_count())

# Step 2: `pool.apply` the `howmany_within_range()`
results = [pool.apply(howmany_within_range, args=(row, 4, 8)) for row in data]

# Step 3: Don't forget to close
pool.close()

print(results[:10])
#> [3, 1, 4, 4, 4, 2, 1, 1, 3, 3]
```

5.2. Parallelizing using Pool.map()

`Pool.map()` accepts only one iterable as argument. So as a workaround, I modify the `howmany_within_range` function by setting a default to the `minimum` and `maximum` parameters to create a new `howmany_within_range_rowonly()` function so it accetps only an iterable list of rows as input. I know this is not a nice usecase of `map()`, but it clearly shows how it differs from `apply()`.

```python
# Parallelizing using Pool.map()
import multiprocessing as mp

# Redefine, with only 1 mandatory argument.
def howmany_within_range_rowonly(row, minimum=4, maximum=8):
    count = 0
    for n in row:
        if minimum <= n <= maximum:
            count = count + 1
    return count

pool = mp.Pool(mp.cpu_count())

results = pool.map(howmany_within_range_rowonly, [row for row in data])

pool.close()

print(results[:10])
#> [3, 1, 4, 4, 4, 2, 1, 1, 3, 3]
```

5.3. Parallelizing using Pool.starmap()

In previous example, we have to redefine `howmany_within_range` function to make couple of parameters to take default values. Using `starmap()`, you can avoid doing this. How you ask?

Like `Pool.map()`, `Pool.starmap()` also accepts only one iterable as argument, but in `starmap()`, each element in that iterable is also a iterable. You can to provide the arguments to the 'function-to-be-parallelized' in the same order in this inner iterable element, will in turn be unpacked during execution.

So effectively, `Pool.starmap()` is like a version of `Pool.map()` that accepts arguments.

```python
 # Parallelizing with Pool.starmap()
import multiprocessing as mp

pool = mp.Pool(mp.cpu_count())

results = pool.starmap(howmany_within_range, [(row, 4, 8) for row in data])

pool.close()

print(results[:10])
#> [3, 1, 4, 4, 4, 2, 1, 1, 3, 3]
```

6. Asynchronous Parallel Processing

The asynchronous equivalents `apply_async()`, `map_async()` and `starmap_async()` lets you do execute the processes in parallel asynchronously, that is the next process can start as soon as previous one gets over without regard for the starting order. As a result, there is no guarantee that the result will be in the same order as the input.

6.1 Parallelizing with Pool.apply_async()

`apply_async()` is very similar to `apply()` except that you need to provide a callback function that tells how the computed results should be stored.

However, a caveat with `apply_async()` is, the order of numbers in the result gets jumbled up indicating the processes did not complete in the order it was started.

A workaround for this is, we redefine a new `howmany_within_range2()` to accept and return the iteration number (`i`) as well and then sort the final results.

```python
 # Parallel processing with Pool.apply_async()

import multiprocessing as mp
pool = mp.Pool(mp.cpu_count())

results = []

# Step 1: Redefine, to accept `i`, the iteration number
def howmany_within_range2(i, row, minimum, maximum):
    """Returns how many numbers lie within `maximum` and `minimum` in a given `row`"""
    count = 0
    for n in row:
        if minimum <= n <= maximum:
            count = count + 1
    return (i, count)


# Step 2: Define callback function to collect the output in `results`
def collect_result(result):
    global results
    results.append(result)


# Step 3: Use loop to parallelize
for i, row in enumerate(data):
    pool.apply_async(howmany_within_range2, args=(i, row, 4, 8), callback=collect_result)

# Step 4: Close Pool and let all the processes complete
pool.close()
pool.join()  # postpones the execution of next line of code until all processes in the queue ar

# Step 5: Sort results [OPTIONAL]
results.sort(key=lambda x: x[0])
results_final = [r for i, r in results]

print(results_final[:10])
#> [3, 1, 4, 4, 4, 2, 1, 1, 3, 3]
```

It is possible to use `apply_async()` without providing a `callback` function. Only that, if you don't provide a callback, then you get a list of `pool.ApplyResult` objects which contains the computed output values from each process. From this, you need to use the `pool.ApplyResult.get()` method to retrieve the desired final result.

```python
# Parallel processing with Pool.apply_async() without callback function

import multiprocessing as mp
pool = mp.Pool(mp.cpu_count())

results = []

# call apply_async() without callback
result_objects = [pool.apply_async(howmany_within_range2, args=(i, row, 4, 8)) for i, row in en

# result_objects is a list of pool.ApplyResult objects
results = [r.get()[1] for r in result_objects]

pool.close()
pool.join()
print(results[:10])
#> [3, 1, 4, 4, 4, 2, 1, 1, 3, 3]
```

6.2 Parallelizing with Pool.starmap_async()

You saw how apply_async() works. Can you imagine and write up an equivalent version for starmap_async and map_async? The implementation is below anyways.

```python
# Parallelizing with Pool.starmap_async()

import multiprocessing as mp
pool = mp.Pool(mp.cpu_count())

results = []

results = pool.starmap_async(howmany_within_range2, [(i, row, 4, 8) for i, row in enumerate(dat

# With map, use `howmany_within_range_rowonly` instead
# results = pool.map_async(howmany_within_range_rowonly, [row for row in data]).get()

pool.close()
print(results[:10])
#> [3, 1, 4, 4, 4, 2, 1, 1, 3, 3]
```

7. How to Parallelize a Pandas DataFrame?

So far you've seen how to parallelize a function by making it work on lists.

But when working in data analysis or machine learning projects, you might want to parallelize Pandas Dataframes, which are the most commonly used objects (besides numpy arrays) to store tabular data.

When it comes to parallelizing a DataFrame, you can make the function-to-be-parallelized to take as an input parameter:

- one row of the dataframe

- one column of the dataframe
- the entire dataframe itself

The first 2 can be done using multiprocessing module itself. But for the last one, that is parallelizing on an entire dataframe, we will use the pathos package that uses dill for serialization internally.

First, lets create a sample dataframe and see how to do row-wise and column-wise paralleization. Something like using pd.apply() on a user defined function but in parallel.

```
import numpy as np
import pandas as pd
import multiprocessing as mp

df = pd.DataFrame(np.random.randint(3, 10, size=[5, 2]))
print(df.head())
#>    0  1
#> 0  8  5
#> 1  5  3
#> 2  3  4
#> 3  4  4
#> 4  7  9
```

We have a dataframe. Let's apply the hypotenuse function on each row, but running 4 processes at a time.

To do this, we exploit the df.itertuples(name=False) . By setting name=False , you are passing each row of the dataframe as a simple tuple to the hypotenuse function.

```
# Row wise Operation
def hypotenuse(row):
    return round(row[1]**2 + row[2]**2, 2)**0.5

with mp.Pool(4) as pool:
    result = pool.imap(hypotenuse, df.itertuples(name=False), chunksize=10)
    output = [round(x, 2) for x in result]

print(output)
#> [9.43, 5.83, 5.0, 5.66, 11.4]
```

That was an example of row-wise parallelization. Let's also do a column-wise parallelization. For this, I use df.iteritems() to pass an entire column as a series to the sum_of_squares function.

```
 # Column wise Operation
def sum_of_squares(column):

    return sum([i**2 for i in column[1]])


with mp.Pool(2) as pool:

    result = pool.imap(sum_of_squares, df.iteritems(), chunksize=10)

    output = [x for x in result]


print(output)
#> [163, 147]
```

Now comes the third part – Parallelizing a function that accepts a Pandas
Dataframe, NumPy Array, etc. Pathos follows the multiprocessing style of: Pool >
Map > Close > Join > Clear. Check out the pathos docs
(https://github.com/uqfoundation/pathos) for more info.

```
 import numpy as np
import pandas as pd
import multiprocessing as mp
from pathos.multiprocessing import ProcessingPool as Pool


df = pd.DataFrame(np.random.randint(3, 10, size=[500, 2]))


def func(df):

    return df.shape


cores=mp.cpu_count()


df_split = np.array_split(df, cores, axis=0)


# create the multiprocessing pool
pool = Pool(cores)


# process the DataFrame by mapping function to each df across the pool
df_out = np.vstack(pool.map(func, df_split))


# close down the pool and join
pool.close()
pool.join()
pool.clear()
```

Thanks to notsoprocoder (https://www.reddit.com/user/notsoprocoder) for this
contribution based on pathos. If you are familiar with pandas dataframes but want
to get hands-on and master it, check out these pandas exercises
(https://www.machinelearningplus.com/python/101-pandas-exercises-python/).

8. Exercises

Problem 1: Use Pool.apply() to get the row wise common items in list_a and
 list_b .

```
list_a = [[1, 2, 3], [5, 6, 7, 8], [10, 11, 12], [20, 21]]
list_b = [[2, 3, 4, 5], [6, 9, 10], [11, 12, 13, 14], [21, 24, 25]]
```

Show Solution

Problem 2: Use `Pool.map()` to run the following python scripts in parallel.
Script names: 'script1.py', 'script2.py', 'script3.py'

Show Solution

Problem 3: Normalize each row of 2d array (list) to vary between 0 and 1.

```
list_a = [[2, 3, 4, 5], [6, 9, 10, 12], [11, 12, 13, 14], [21, 24, 25, 26]]
```

Show Solution

9. Conclusion

Hope you were able to solve the above exercises, congratulations if you did!

In this post, we saw the overall procedure and various ways to implement parallel processing using the multiprocessing module. The procedure described above is pretty much the same even if you work on larger machines with many more number of processors, where you may reap the real speed benefits of parallel processing.

Happy coding and I'll see you in the next one!

What do you think?
41 Responses

👍 Upvote   😍 Love

Comments **Community** 🔒 **Privacy Policy** ① **Login** ▾

♡ **Recommend** 1   **Sort by Newest** ▾

```
Join the discussion…
```

LOG IN WITH   **OR SIGN UP WITH DISQUS** ⑦

```
Name
```

gwangjin • 3 months ago • edited
generalized as functions:

```
###############################################
# pmap (unary functions)
###############################################

def pmap(func_unary, lst, cpu_exclude_n=0):
    """Inserts x over lst as func(x)"""
```

Feedback

```
with mp.Pool(mp.cpu_count() - cpu_exclude_n) as pool:
return pool.map(func_unary, lst)


###############################################
# pstarmap (list of function arguments packed in tuples)
###############################################


def pstarmap(func_multi_args, lst_of_args_list,
cpu_exclude_n=0):
"""Inserts elements of lists as *element"""
with mp.Pool(mp.cpu_count() - cpu_exclude_n) as pool:
return pool.starmap(func_multi_args, lst_of_args_list)
```

I think # papply is superfluous, because pstarmap already covers case of non-unary functions.

⌃ | ⌄ · Reply ·

disqus_q77Vm0Uw78 · 6 months ago · edited

It is taking a lot of time for me. Without apply or map it takes around 2 seconds in my laptop but using apply or map it's been running for over 5 minutes. Doesn't make any sense and I can't see what the problem is, I'm running the same code as yours

⌃ | ⌄ · Reply ·

gwangjin · 9 months ago · edited

Thank you for your tutorial! Very helpful!

I generalized your examples to functions for future use:

```
###############################################
# map (unary functions), starmap (multi arg functions)
# apply (multi arg functions)
###############################################


import multiprocessing as mp
```

```
def pmap(func_unary, lst, cpu_exclude_n=0):
"""Inserts x over lst as func(x)"""
with mp.Pool(mp.cpu_count() - cpu_exclude_n) as pool:
result = pool.map(func_unary, lst)
return result


def pstarmap(func_multi_args, lst_of_args_list,
cpu_exclude_n=0):
"""Inserts x over lst as func(*x)"""
with mp.Pool(mp.cpu_count() - cpu_exclude_n) as pool:
result = pool.starmap(func_multi_args,
lst_of_args_list) # inserts x as *x
return result


def papply(func_multi_args, lst_of_args_list,
cpu_exclude_n=0):
with mp.Pool(mp.cpu_count() - cpu_exclude_n) as pool:
result = [pool.apply(func_multi_args, args=x) for x in
lst_of_args_list]
return result
```

There I had question to `pool.apply()`, what does it parallelize then actually?
The loop `for x in lst_of_args_list` is outside the pool.apply() function - so parallelization over the `lst_of_args_list` I would exclude in your (and my generalized example). Is this a logical mistake which you made? Or did I misunderstood what exactly gets

Feedback

parallelized?

1 ∧ | ∨ • Reply •

Syntax Error • 10 months ago • edited

Thank you for your post. Unfortunately, I don't find it so useful. It would be very nice if more details and discussion are given.

∧ | ∨ 1 • Reply •

Meng • 10 months ago

I got an attribution error when running step 5.1. There appears to be a bug with the multiprocessing feature. Check out https://stackoverflow.com/q... for a workaround.

∧ | ∨ • Reply •

Kavindu Zoysa • a year ago

In the 5.1 part [5.1. Parallelizing using Pool.apply()], it takes comparatively a long time to execute "results = [pool.apply(howmany_within_range, args=(row, 4, 8)) for row in data]" . It is about 23 seconds. Is this the expected behaviour?

Feedback