Manual for Parallel Computing in Python

I. Introduction

The ability to parallelize code has become a necessary tool to any aspiring developer due to the rapid increase of multicore processors. This manual seeks to convey the knowledge of how to utilize parallel programming using the standard Python 3.9 modules. Parallel computing or parallel programming is a type of computing architecture in which several processors execute an application or computation simultaneously. This “parallelizing” of the code allows for more complex applications and/or a greater amount of computational power than can be accomplished with a single thread of code execution.

II. Intended Audience

This manual assumes that the reader has prior programming experience and a familiarity with Python. Having said this, there are some topics that should be summarized before beginning with the procedure. To reiterate, parallel computing is a type of programming that utilizes multiple sequences of code execution simultaneously to perform more machine instructions. In regards to this text, a thread is a single sequence of instructions that can be managed independently. The majority of projects that a beginning programmer studies are done with a single thread and can be considered serial programs. In addition to this, a process is an instance that is being executed by one or more threads. Multi-processing, is as its name implies, the ability to access multiple processes each with their own supply of threads to execute instructions within an application. With this knowledge, the progression from lowest complexity to highest for parallel programming is: serial process, multi-threaded process, multi-process application.

III. Preface

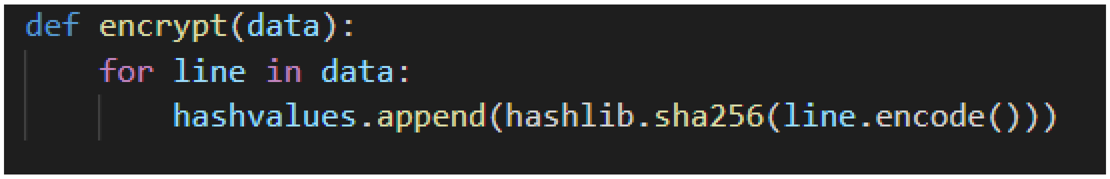
This section will go into detail on a general approach to parallelizing an I/O bound or CPU bound process using Python and its standard modules. The examples used may have modules that are not included with the current version of Python (Python 3.9.0) used by this manual. This text will include one example of an I/O bound process and one example of a CPU bound process.

1. To begin parallelizing I/O and CPU bound programs, first determine the “driver” function. This “driver” function is the function that will be performing the specific instructions that is to be parallelized. This function typically will be called many times by the program.

2. Once the driver function is determined, the input that is fed to function must be processed. The data must be partitioned correctly in order for the code to be parallelized correctly. The easiest way to accomplish this is to create a list object with all of the data stored inside.
3. After the list object is created, begin operating over the list object with a simple for loop which will pass this data into the appropriate thread or process object which will parallelize the program.

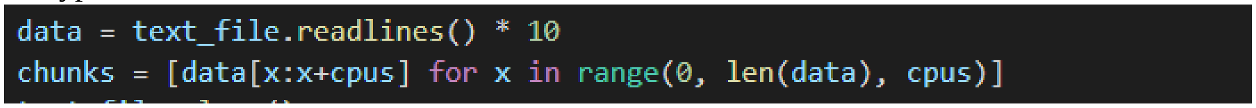
Example of Multithreading (CPU bound process):

This program's purpose is to encrypt a list of strings.



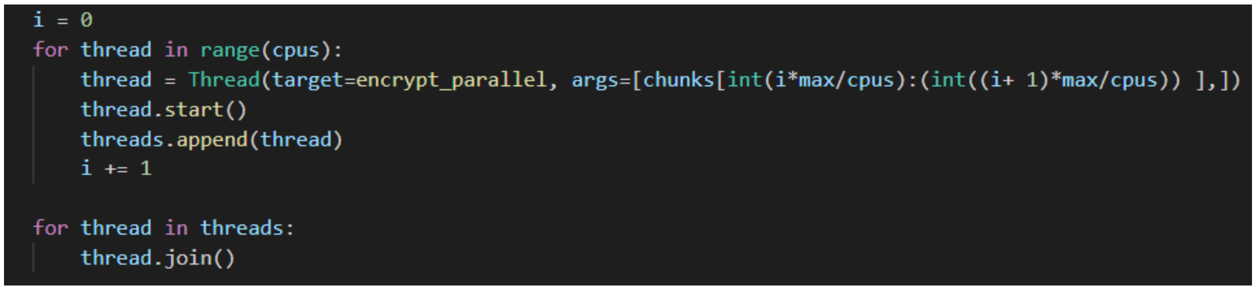
```python
def encrypt(data):
    for line in data:
        hashvalues.append(hashlib.sha256(line.encode()))
```

The list of strings is formed from a text file located within the same directory as the script. This list must then be split into a list of chunks. Each chunk can be fed to the appropriate thread for encryption.



```python
data = text_file.readlines() * 10
chunks = [data[x:x+cpus] for x in range(0, len(data), cpus)]
```

The threads are stored into a list and then are then iterated over using a for loop as shown below. In the context of this program, the number of threads is equal to the number of cores on the CPU.

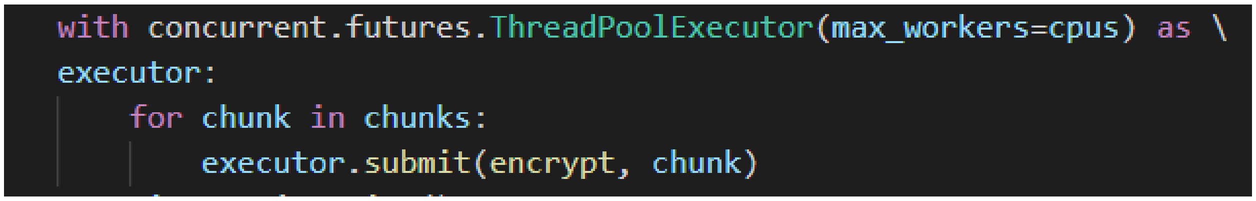


```python
i = 0
for thread in range(cpus):
    thread = Thread(target=encrypt_parallel, args=[chunks[int(i*max/cpus):(int((i+ 1)*max/cpus)) ],])
    thread.start()
    threads.append(thread)
    i += 1

for thread in threads:
    thread.join()
```

Example of Multithreading #2(CPU bound process):

This example displays another method for parallelizing the same script using a Thread pool.



```python
with concurrent.futures.ThreadPoolExecutor(max_workers=cpus) as \
executor:
    for chunk in chunks:
        executor.submit(encrypt, chunk)
```

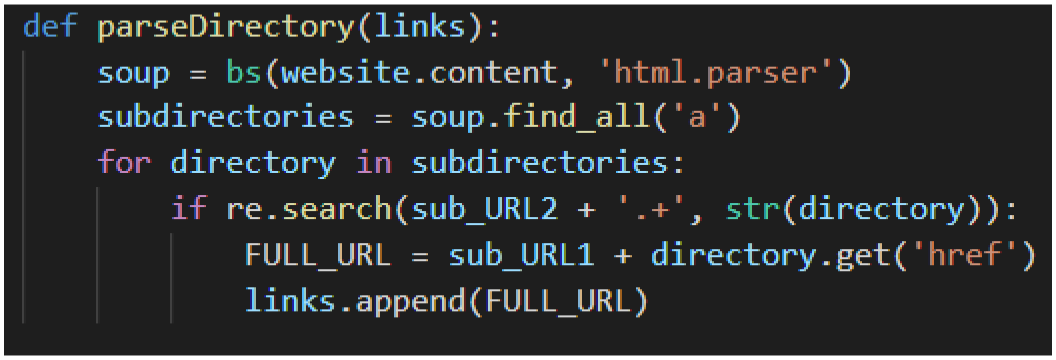
The syntax here is much simpler as it uses a prebuilt Python module, concurrent.futures, which takes care of all of the thread creating, starting, and joining behind the scenes.

Threads vs Processes in Python:

As a programming language, Python is known for its simplified syntax and efficiency for processing large amounts of data. It accomplishes this through the help of a tool known as the Global Interpreter Lock or GIL. One of the side effects of the GIL is that it locks CPU intensive multi-threaded programs to using a single thread. This problem is non-existent for I/O intensive multi-threaded programs as the GIL is shared between threads as they wait for more input. Another way to circumvent the GIL is to use multiple processes instead of multiple threads. Processes are independent of one another, each with their own interpreter and memory. This means that the GIL does not inhibit processes like it does to threads. The downside of this means that processes are more resource intensive than threads and do not scale into large projects as well as threads will. For more information on the GIL, please see the appendix.

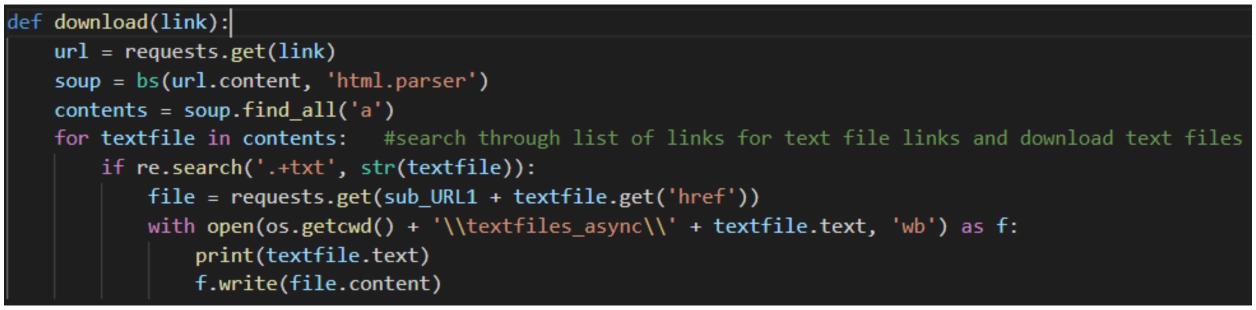
Example of Multiprocessing (I/O bound process):

This program's purpose is to download a series of text files from the U.S. Securities and Exchange Commission. It accomplishes this by parsing through the HTML code of the given URL and building a list of string objects named "links" which is filled with the URLs of all of the subdirectories contained within. This is implemented by a function named "parseDirectory" which takes a list for its single argument.



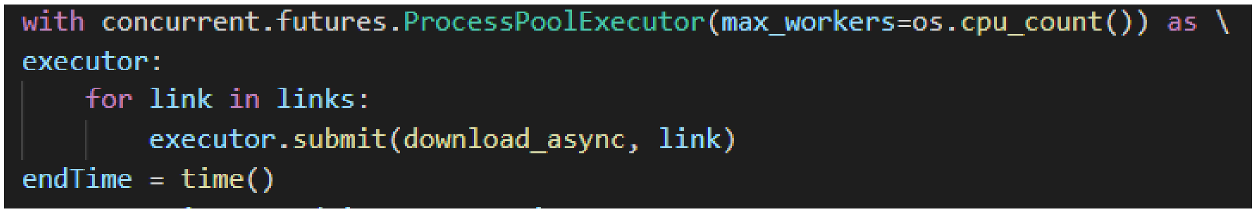
```python
def parseDirectory(links):
    soup = bs(website.content, 'html.parser')
    subdirectories = soup.find_all('a')
    for directory in subdirectories:
        if re.search(sub_URL2 + '.+', str(directory)):
            FULL_URL = sub_URL1 + directory.get('href')
            links.append(FULL_URL)
```

This list of links is then passed into the driver function named "download".



```python
def download(link):
    url = requests.get(link)
    soup = bs(url.content, 'html.parser')
    contents = soup.find_all('a')
    for textfile in contents:   #search through list of links for text file links and download text files
        if re.search('.+txt', str(textfile)):
            file = requests.get(sub_URL1 + textfile.get('href'))
            with open(os.getcwd() + '\\textfiles_async\\' + textfile.text, 'wb') as f:
                print(textfile.text)
                f.write(file.content)
```

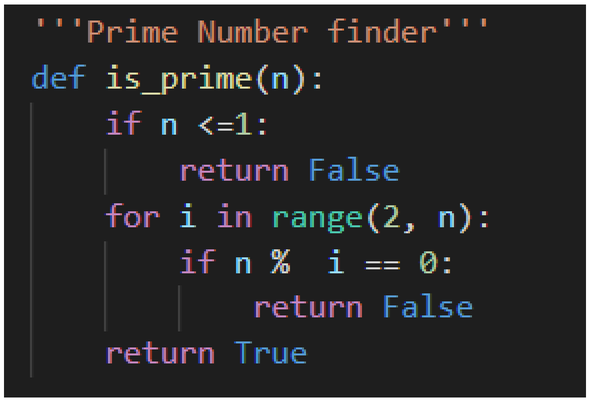
This function takes in a singular link (string) as its argument in order to coordinate with Python's process pool executor.



```
with concurrent.futures.ProcessPoolExecutor(max_workers=os.cpu_count()) as \
executor:
    for link in links:
        executor.submit(download_async, link)
endTime = time()
```

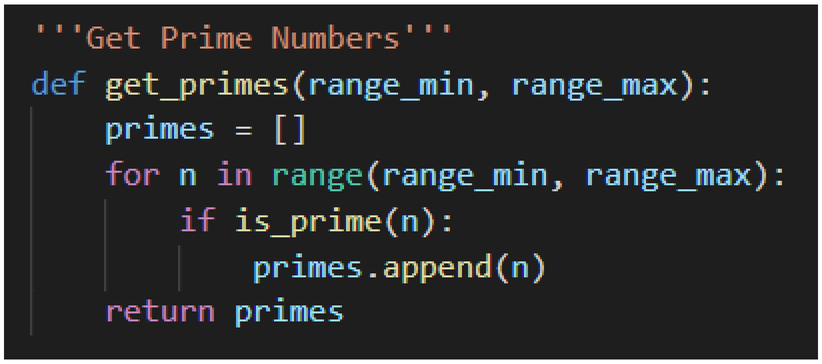
Example of Multiprocessing #2 (CPU Bound Process):

This program uses multiprocessing to calculate and determine prime numbers and then append those numbers to a list.



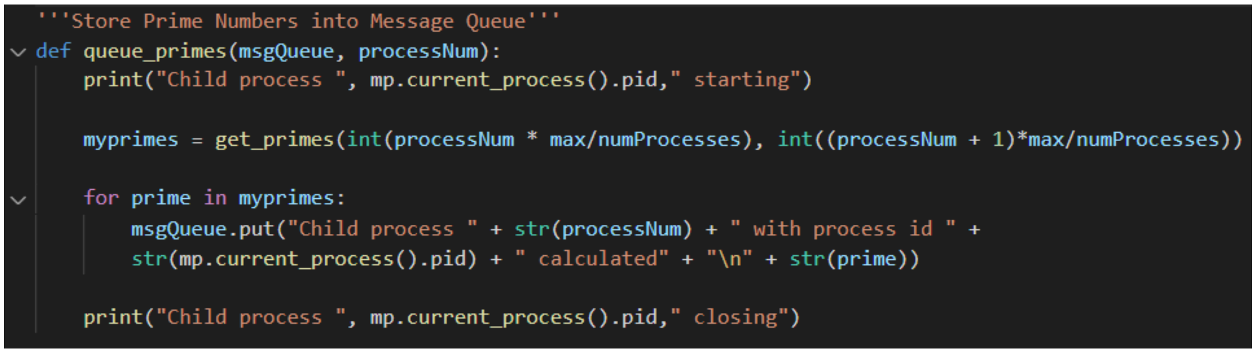
```
'''Prime Number finder'''
def is_prime(n):
    if n <=1:
        return False
    for i in range(2, n):
        if n %  i == 0:
            return False
    return True
```

It uses a simple algorithm to determine if the given number is prime or not.



```
'''Get Prime Numbers'''
def get_primes(range_min, range_max):
    primes = []
    for n in range(range_min, range_max):
        if is_prime(n):
            primes.append(n)
    return primes
```

This function is used to find all of the prime numbers within a given range of numbers and appends them to a list.



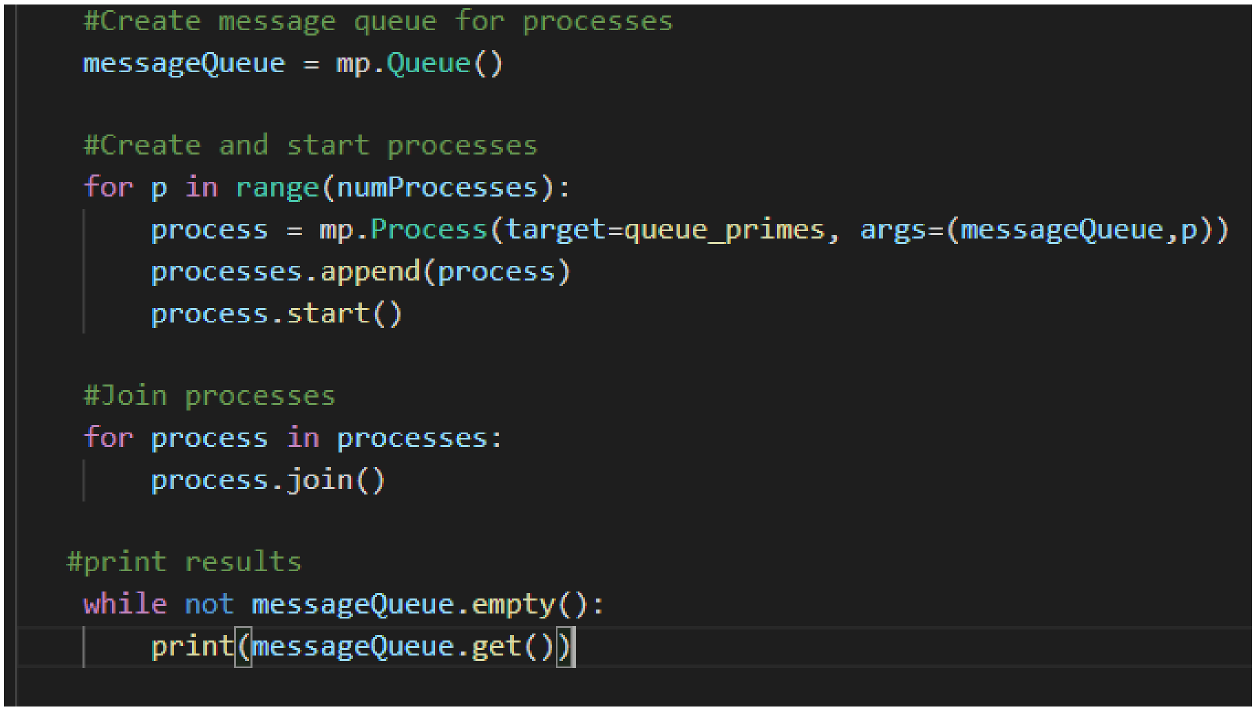
```python
'''Store Prime Numbers into Message Queue'''
def queue_primes(msgQueue, processNum):
    print("Child process ", mp.current_process().pid," starting")

    myprimes = get_primes(int(processNum * max/numProcesses), int((processNum + 1)*max/numProcesses))

    for prime in myprimes:
        msgQueue.put("Child process " + str(processNum) + " with process id " +
        str(mp.current_process().pid) + " calculated" + "\n" + str(prime))

    print("Child process ", mp.current_process().pid," closing")
```

This is the driver function for this program. It creates a list using the get_primes function and then places each of those prime numbers into a queue which will be accessed by the specific process that is calling this function at the time.



```python
    #Create message queue for processes
    messageQueue = mp.Queue()

    #Create and start processes
    for p in range(numProcesses):
        process = mp.Process(target=queue_primes, args=(messageQueue,p))
        processes.append(process)
        process.start()

    #Join processes
    for process in processes:
        process.join()

   #print results
    while not messageQueue.empty():
        print(messageQueue.get())
```

This is the main function for this program. It creates a queue using the multiprocessing module and then creates the processes that will be used to queue the primes. Once a process is created, it is appended to a list and then the process is started. Then we join each of the processes using the built-in join function. Note that this method is almost identical to the method used to create threads in example 1.

IV. Conclusion

Parallel computing, when done correctly, can be a gamechanger for any developer looking to add to their skillset. This manual covered only a fraction of potential methods that can be used for parallelizing the Python code. For any additional details regarding some of the material used in the manual, please see the attached appendix.

V. Appendix

A.1 Threads:

In Python, the thread class represents objects that run with their own sequence of execution. Each thread object shares the same memory space and therefore can share data between one another. They also share the same instance of the Global Interpreter Lock and will be inhibited by the GIL if the threads are used to perform CPU intensive tasks.

Threads have two main methods that are used in this manual. They are Thread.start() and Thread.join(). Thread.start() does as its name implies and starts the current thread's line of execution. The start method must be called at most once per thread object. The Thread.join() method tells the calling thread to wait until the current thread terminates. A thread object can be joined multiple times.

A.2 Thread Pool Executor:

This executor object uses a group or "pool" of threads to execute tasks in parallel. This object does all of the creation, starting, and joining of threads behind the scenes which leads to simpler syntax. As of Python 3.8, it utilizes up to 32 CPU cores for CPU bound tasks which release the GIL. It is also resource friendly implicitly on many-core machines.

A.3 Processes:

In Python, the process class represents objects that run their own sequence of execution, similar to the thread object discusses in A.1. Unlike threads, processes have their own memory space and their own instance of the Python interpreter. This leads to processes using more resources than threads but allows for them to accomplish some tasks easier than threads. Since each thread has its own instance of the GIL, the processes are able to run in parallel unimpeded by the lock of the other processes. This makes multiprocessing a good choice when parallelizing CPU bound programs in Python.

The Process class in Python does require slightly more code in order to function as intended as opposed to threads. In order to spawn processes, the set_start_method() must be called at the beginning of the main function to be able to spawn processes objects within that scope of code. This process should not be used more than once within a program. As with the thread object, the process object has its own implementation of the start() and join() method. They function essentially the same as with threads and can be called in the same manner.

A.4 Process Pool Executor:

This is the process object implementation similar to the thread pool executor. It uses the multiprocessing module, which allows the object to bypass the GIL.