Birth of the Multiprocessing in Python

Group Members: Edward Zhu, Steven Tang, John Zavidniak

Introduction:

Python did not originally include the multiprocessing package in it's standard library. Before the implementation of the package, the default python interpreter was designed to be simplistic with a thread safe mechanism known as GIL (Global Interpreter Lock). To prevent conflicts between threads, the lock only enabled single statement executions (single threading). With the inclusion of the "multiprocessing" package in version 2.6, previously named pyProcessing, it allowed users to spawn processes using an API very similar to the threading module. This package mimics the threading module functionality to provide a process-based approach to threaded programming. This allows end-users to run multiple tasks with either local and remote concurrency, which effectively side-steps the global interpreter lock.

MultiThreading vs Multiprocessing:

Before delving further into this multiprocessing module, it is a good idea to look at the differences between multithreading and multiprocessing. In multithreading there exists one process, and each individual thread exists in this process. Therefore the threads have the ability to share memory and communicate with each other. While this may be beneficial in some cases, it can also be harmful in others. Allowing multiple threads to access the same shared memory can possibly lead to problems such as race conditions. Other problems with this approach may be synchronization errors and other miscommunications as well. This is where the concept of multiprocessing as opposed to multithreading comes into play as a benefit. With this approach, each process is separated and completely independent from all the others processes currently in use. As a result, there is no memory that is being shared between the different process, and this results in the previously mentioned problems being completely avoided. However, this does come at a cost. There is additional overhead that comes along with running and controlling multiple processes as opposed to multiple threads, but this can be manageable and not a big deal if the gain from the safeness and independence outweighs this negative. Therefore, it is often a good idea to make use of multiprocessing, and Python incorporated a nice, standard library module for doing so.

Why MultiProcess?:

What is the purpose behind utilizing the multiprocessing module in python and how can it be useful to programmers? To understand the features that this package provides, we must first delve into the practicality of multiprocessing. By definition, multiprocessing allows the usage of two or more central processing units within a single device's system. It allows for the allocation of different tasks to various processors so that work can be divided evenly (or not) among them. Since multiple processors are being utilized for the given tasks, performance is increased due to many processes being done simultaneously. For example, if we have one CPU respond to all hardware interrupts and the rest respond to multiple background programs, we won't find the need to execute hardware interrupt handlers and programs in separate time spans.

Basic Classes:

The multiprocessing package comes with many built-in options for creating a parallel application. The three basic (and safe to use) are the Process, Queue and Pool classes.

The 'Process' class:

The Process class is the most important since it's an abstraction that sets up a python process - the root of multiprocessing itself. Once imported, a process object can be instantiated to run a function, followed by a start method to execute it.

```python
from multiprocessing import Process

def f(name):
    print 'hello', name

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

The example above provides insight as to how processes are spawned using the module. The process object "p" was first instantiated and initialized to target the function f and passes in the argument 'bob'. Once the start method is called, the process executes and returns the output of the function - in this case, 'hello bob'. After the result is returned, we tell the process to complete by using the join method. This is necessary because the process will become inactive and not terminate, becoming a zombie that you must manually kill. You can create as many processes as you needed, but be aware that there is a limit on performance enhancement due to differing number of cores, system scheduler, and other factors in each computer. You will likely want to build in customizable settings in this regard.

The ‘Queue’ Class:

Once the process class divides up the job among several workers, the Queue Class enables communication between so that messages can be passed back and forth.The Queue class is essentially one of two primary methods of communication that is available in the multiprocessing package, the other one being Pipes. This class literally has the same functionality as Queue.queue(), a FIFO data structure. It’s internally synchronized, and therefore, thread and process safe.

```
import multiprocessing

class MyFancyClass(object):

    def __init__(self, name):
        self.name = name

    def do_something(self):
        proc_name = multiprocessing.current_process().name
        print 'Doing something fancy in %s for %s!' % (proc_name, self.name)


def worker(q):
    obj = q.get()
    obj.do_something()


if __name__ == '__main__':
    queue = multiprocessing.Queue()

    p = multiprocessing.Process(target=worker, args=(queue,))
    p.start()

    queue.put(MyFancyClass('Fancy Dan'))

    # Wait for the worker to finish
    queue.close()
    queue.join_thread()
    p.join()
```

```
$ python multiprocessing_queue.py

Doing something fancy in Process-1 for Fancy Dan!
```

This example above passes a single message to a worker, then the main process allows for the worker to complete its job. The queue in this case initializes an instance of MyFancyClass, then outputs the string when the do_something method is called. Basically, a process needs a queue in order to receive the results.

The ‘Pool’ class:

The Pool class provides an excellent way to manage parallel-processing tasks. The Pool class may be instantiated with a certain number of processes, and this will limit all operations performed by the object to that maximum number. One useful function in the Pool class is map. This allows a certain function passed as the first argument to be ran a certain number of times based on the second argument. As permitted by the maximum number of processes allowed, each function ran in map will be ran on a new process and in parallel. Here is an example code snippet:

```
def cube(x):
    return x**3
pool = mp.Pool(processes=6)
results = pool.map(cube, range(1,7))
print(results)

[1, 8, 27, 64, 125, 216]
```

As seen in this example, a function named cube is defined which takes a number and returns the cube of that number. Then a new pool object is created with a maximum of six processes. The pool.map function then takes the function cube, and calls cube(1), cube(2), . . . , cube(6). Since the maximum number of processes is six and cube needs to be called for six different arguments, each call to cube will occur on a different process. If the number of calls to cube was greater than the maximum number of process, processes would start to get reused. Finally, the variable results is set equal to the array of results returned by pool’s map function, and then results is then printed out to the console giving the results shown. This example illustrates the power of the pool class and how easily one can use Python’s multiprocess module to get things done.

General Drawbacks:

Even though the functionalities of the multiprocessing package can be useful, there are some drawbacks when using it. Minor synchronization issues aside, sharing data by messaging requires programmers to make copies of everything that they'd like to share. For example, a parent process sends data to each child process. In this scenario, copying the data over isn't an expensive operation, but retrieving the results from it is. Each child is required to send back the contents of the items that were updated. If we have quite a bit of information and only a few processes, the result could be quite substantial (in terms of memory usage of course). Unfortunately, this will hinder performance tremendously since copying over data from a child process to the parent process multiple times will not help in speeding up the execution of a program. However, there is a workaround for this limitation. We can utilize the two state sharing methods available in multiprocessing: shared memory and server processing. Shared memory uses Value and Array classes. An update to any instance of these two objects will immediately be visible to any other process that can access the objects. Since all the processes can access the shared memory location much like regular working memory, it allows multiple programs to communicate among each other without creating redundant copies of data. Server processes utilizes Manager classes. The object holds Python objects and allows processes to manipulate them by using proxies. The manager objects are more flexible than value and array objects because they can support arbitrary object types. Server processes are, however, slower than shared memory. There are also open issues by the developer, as there is no remote connection capabilities. If users will need to enable the remote security mechanisms for classes they want to run multiprocessing on. Also some APIs such as qsize(), task_done(), and join() have not been added yet.

OS Restrictions:

There are also some restrictions between different platforms due to capabilities and resources that different operating systems run. Although on both Unix and Windows based machines the majority of the multiprocessing library is available for development, Windows has just a few restrictions. Because it lacks the os.fork() capability, Windows will have a harder time instantiating global variables as well as some implementations of simple process calls.

```
from multiprocessing import Process

def foo():
    print 'hello'

p = Process(target=foo)
p.start()
```

The code above will fail under Windows; to compensate this restrictions, users must protect the “entry point” of the program by using the **name** == ‘**main**’ which will allow the newly spawned Python interpreter to safely import the module and then run the modules inner functions. There are also similar restrictions with the multiprocessing pool or manager modules when they are created in the main module, therefore similar workaround must be implemented.

Conclusion:

The introduction of the multiprocessing module to the standard python library provided a new era of programming for the powerful language that Python has evolved to be. Leaving the original convention of multithreading or single, linear data processing, this multiprocessing module increased the speed of of programs by a maximum of nearly three times, in almost all numerical computing and data fetching/manipulation benchmarks. This module was a crucial addition to the roadmap of Python; there were many downfalls of Python in it’s early days, one being the lack of powerful multiprocessing, but this module changed that completely and also spawned many other innovations and additions to implementation of Python over the golden years to come.

Citations

[1] http://legacy.python.org/dev/peps/pep-0371/

[2] https://wiki.python.org/moin/ParallelProcessing

[3] https://docs.python.org/2/library/multiprocessing.html

[4] https://code.google.com/p/python-safethread/

[5] https://code.google.com/p/unladen-swallow/wiki/MemoryModel