Parallel Python using the Multiprocess(ing) Package

K. Cooper\textsuperscript{1}

\textsuperscript{1}Department of Mathematics
Washington State University

2018
Caveats

My understanding of Parallel Python is not mature, so anything said here is somewhat questionable. There seem to be two prominent packages for parallel Python:

1. Parallel Python - pp
2. Multiprocessing, or a fork of that called Multiprocess.

This concerns only the second.
The `multiprocess` package allows us to run multiple instances of our Python program.

In Linux, the original program *forks* the new processes. This duplicates existing memory and continues execution in each process from the current line.

In Windows the original program simply starts copies of itself which run from the start.

Multiprocess pretty much uses a master-slave paradigm for parallel processing.
The basic format for a multiprocessing program goes like this.

```python
import multiprocessing as mp

def function_known_to_every_instance(args):
    Code for the function

if __name__ == '__main__':
    Code that only runs in master program goes here
```
Threading

The multiprocess(ing) packages are actually slightly modified/augmented version of the threading package, which allows multithreaded computing.

The principal changes have to do with the addition of a Pool method for generating threads.
There are two ways to spawn processes for parallel execution:

**Process**: basically inherited from the `threading` package, creates the required number of processes and loads them all at once.

**Pool**: creates as many processes as the number of processors you specify, and waits until those finish to create any more processes.

Pool seems generally more efficient, but Process might be more intuitive, and for small numbers of processes, might require less overhead.
def parallel(arguments):
    commands
    if __name__ == '__main__':
        p = []
        for i in range(nProcs):
            p.append(mp.Process(
                target=parallel, args=(arguments))
            p[i].start()
        for i in range(nProcs):
            p[i].join()
Process

Process: Sets up the call to the function that will run in parallel.

start: Starts the separate process.

join: Blocks execution of the originating code until the specified process completes.
Multiprocess(ing) Considerations

Blocking

In general, we want all the processes to run asynchronously. However, before we gather results, we must be sure those processes are finished.

That is the function of `join()` - it waits until the process it belongs to is finished.

Typically after a join you will `get()` results.
Results

results = [mp.Queue() for i in range(nProcs)]
p = [mp.Process(
    target=parallel, args=(args,))
    for i in range(nProcs)]
for i in range(nProcs):
    p[i].start()
for i in range(nProcs):
    p[i].join()
results.append(results[i].get())
The Queue is a class for gathering results from parallel function instances.

We created a list of Queues, each corresponding to a parallel instance.

The `get()` method collects the results from the corresponding instance.

This procedure is slow.
Queue

Note that to return results to a Queue we do not use a return statement.

Fill an array with results, then use `put()` method in the queue, which was passed to the function.

```python
def parallel(resQueue, arguments):
    commands...
    resQueue.put(theResults)
```
Pools are in some ways easier to use.

```python
p = mp.Pool(processes=nProcs)
results = p.map(parallel, [List_of_args])
p.close()
p.join()
```
Pool makes a process for every collection of arguments in the argument list when a processor is free to accept that process. 

map() maps the arguments to the function parallel.

close() says we are done making processes.

join() blocks until all the processes are finished.

The results from all the processes are in results.

The format might vary.
The Pool function can return values, instead of using `put()`.

```python
def parallel(arguments):
    commands...
    return theResults
```

`map()` returns a list of lists of results.

`map_async()` returns a list of tuples of results.
Single Machine

The `multiprocessing` package is intended for a single node. It cannot be used effectively on a distributed cluster. Communication is not really an issue... Memory and cache are huge issues.
Multiprocessing makes several copies of your process, together with all the memory it requires.

For programs that use significant memory, it is common to lose all parallel gains to cache faults.

Multiprocessing in this context should only be used on programs that are computationally intensive, but that use little memory.

If your processor does not have, say, at least four cores with 2MB L2 cache each, just don’t bother.