12.8 简单的并行编程¶

问题¶

你有个程序要执行CPU密集型工作，你想让他利用多核CPU的优势来运行的快一点。

解决方案¶

`concurrent.futures` 库提供了一个 `ProcessPoolExecutor` 类， 可被用来在一个单独的Python解释器中执行计算密集型函数。 不过，要使用它，你首先要有一些计算密集型的任务。 我们通过一个简单而实际的例子来演示它。假定你有个Apache web服务器日志目录的gzip压缩包：

```
logs/
   20120701.log.gz
   20120702.log.gz
   20120703.log.gz
   20120704.log.gz
   20120705.log.gz
   20120706.log.gz
   ...
```

进一步假设每个日志文件内容类似下面这样：

```
124.115.6.12 - - [10/Jul/2012:00:18:50 -0500] "GET /robots.txt ..." 200 71
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /ply/ ..." 200 11875
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /favicon.ico ..." 404 369
61.135.216.105 - - [10/Jul/2012:00:20:04 -0500] "GET /blog/atom.xml ..." 304 -
...
```

下面是一个脚本，在这些日志文件中查找出所有访问过robots.txt文件的主机：

```
# findrobots.py

import gzip
import io
import glob

def find_robots(filename):
    '''
    Find all of the hosts that access robots.txt in a single log file
    '''
    robots = set()
    with gzip.open(filename) as f:
        for line in io.TextIOWrapper(f,encoding='ascii'):
            fields = line.split()
            if fields[6] == '/robots.txt':
                robots.add(fields[0])
    return robots

def find_all_robots(logdir):
    '''
    Find all hosts across and entire sequence of files
    '''
    files = glob.glob(logdir+'/*.log.gz')
    all_robots = set()
    for robots in map(find_robots, files):
        all_robots.update(robots)
    return all_robots

if __name__ == '__main__':
    robots = find_all_robots('logs')
    for ipaddr in robots:
        print(ipaddr)
```

前面的程序使用了通常的map-reduce风格来编写。 函数 `find_robots()` 在一个文件名集合上做map操作，并将结果汇总

为一个单独的结果， 也就是 `find_all_robots()` 函数中的 `all_robots` 集合。 现在，假设你想要修改这个程序让它使用多核CPU。 很简单——只需要将map()操作替换为一个 `concurrent.futures` 库中生成的类似操作即可。 下面是一个简单修改版本：

```
# findrobots.py

import gzip
import io
import glob
from concurrent import futures

def find_robots(filename):
    '''
    Find all of the hosts that access robots.txt in a single log file

    '''
    robots = set()
    with gzip.open(filename) as f:
        for line in io.TextIOWrapper(f,encoding='ascii'):
            fields = line.split()
            if fields[6] == '/robots.txt':
                robots.add(fields[0])
    return robots

def find_all_robots(logdir):
    '''
    Find all hosts across and entire sequence of files
    '''
    files = glob.glob(logdir+'/*.log.gz')
    all_robots = set()
    with futures.ProcessPoolExecutor() as pool:
        for robots in pool.map(find_robots, files):
            all_robots.update(robots)
    return all_robots

if __name__ == '__main__':
    robots = find_all_robots('logs')
    for ipaddr in robots:
        print(ipaddr)
```

通过这个修改后，运行这个脚本产生同样的结果，但是在四核机器上面比之前快了3.5倍。 实际的性能优化效果根据你的机器CPU数量的不同而不同。

讨论¶

`ProcessPoolExecutor` 的典型用法如下：

```
from concurrent.futures import ProcessPoolExecutor

with ProcessPoolExecutor() as pool:
    ...
    do work in parallel using pool
    ...
```

其原理是，一个 `ProcessPoolExecutor` 创建N个独立的Python解释器， N是系统上面可用CPU的个数。你可以通过提供可选参数给 `ProcessPoolExecutor(N)` 来修改 处理器数量。这个处理池会一直运行到with块中最后一个语句执行完成，然后处理池被关闭。不过，程序会一直等待直到所有提交的工作被处理完成。

被提交到池中的工作必须被定义为一个函数。有两种方法去提交。 如果你想让一个列表推导或一个 `map()` 操作并行执行的话，可使用 `pool.map()`：

```
# A function that performs a lot of work
def work(x):
    ...
    return result

# Nonparallel code
```

```
results = map(work, data)

# Parallel implementation
with ProcessPoolExecutor() as pool:
    results = pool.map(work, data)
```

另外，你可以使用 `pool.submit()` 来手动的提交单个任务：

```
# Some function
def work(x):
    ...
    return result

with ProcessPoolExecutor() as pool:
    ...
    # Example of submitting work to the pool
    future_result = pool.submit(work, arg)

    # Obtaining the result (blocks until done)
    r = future_result.result()
    ...
```

如果你手动提交一个任务，结果是一个 `Future` 实例。 要获取最终结果，你需要调用它的 `result()` 方法。 它会阻塞进程直到结果被返回来。

如果不想阻塞，你还可以使用一个回调函数，例如：

```
def when_done(r):
    print('Got:', r.result())

with ProcessPoolExecutor() as pool:
     future_result = pool.submit(work, arg)
     future_result.add_done_callback(when_done)
```

回调函数接受一个 `Future` 实例，被用来获取最终的结果（比如通过调用它的result()方法）。 尽管处理池很容易使用，在设计大程序的时候还是有很多需要注意的地方，如下几点：

- 这种并行处理技术只适用于那些可以被分解为互相独立部分的问题。
- 被提交的任务必须是简单函数形式。对于方法、闭包和其他类型的并行执行还不支持。
- 函数参数和返回值必须兼容pickle，因为要使用到进程间的通信，所有解释器之间的交换数据必须被序列化
- 被提交的任务函数不应保留状态或有副作用。除了打印日志之类简单的事情，

一旦启动你不能控制子进程的任何行为，因此最好保持简单和纯洁——函数不要去修改环境。

- 在Unix上进程池通过调用 `fork()` 系统调用被创建，

它会克隆Python解释器，包括fork时的所有程序状态。 而在Windows上，克隆解释器时不会克隆状态。 实际的fork操作会在第一次调用 `pool.map()` 或 `pool.submit()` 后发生。

- 当你混合使用进程池和多线程的时候要特别小心。

你应该在创建任何线程之前先创建并激活进程池（比如在程序启动的main线程中创建进程池）。