Barcelona GSE Data Science Center

Distributing Workloads in Python

September, 2018

Definitions

We need to understand a bit about how modern computers, CPUs, and operating systems work. Specifically we need to understand:

- Cores
- Procesess
- Threads

Barcelona GSE Data Science Center

Cores

A simple CPU is a machine for executing processes, one at a time.

Modern CPU's have the ability to execute multiple processes at *exactly* the same time. It's like having multiple CPU's on the same machine. These are called "cores".

Some modern CPU's have "virtual cores" via hyperthreading. For our purposes, we will treat these as "cores."

Processes and Threads

Our operating systems provide two useful concepts: Processes and Threads

1. Processes are intended for independent applications, they get allocated their own memory and exist in separate worlds.
2. Each process, however, can have multiple threads. Threads are independent sequences of work, that share the same memory as the process, but can be scheduled separately by the operating system.

Gettings things done

Sometimes, we do things sequentially: we move on to the next task only after we have finished the first.

Other times, we want to get multiple things done “at the same time.” There are two different ways to do that:

1. Work on two things simultaneously (having one person clean while one person cooks).
2. Juggle two pieces of work (having one person clean while cooking, while waiting for water to boil).

Parellelism vs. Concurrency

Naturally, it makes sense to juggle if the work has some “wait time.” However, not all work has wait time!

In computing terms, we can divide the work which we want to “get things done at the same time” into:

1. CPU bound workloads, where we want to parallelize the computation itself and there’s no waiting needed.
2. I/O bound workloads, where we are waiting on some other part of the system (i.e. HTTP requests to another computer), and thus can “juggle” multiple activities concurrently.

I/O Bound Workloads - Concurrency

When your computer makes an HTTP request, it waits for a response.

Often, this response takes tens or hundreds of milliseconds.

What does your CPU do while waiting? Clearly, even if you only have on core on your CPU, it can be doing something else!

This handling of multiple tasks simultaneously, even if there is no “execution” being done at the same time, is called “concurrency.”

Concurrency in Python

Concurrency in Python is primarily achieved via threading.

We create a bunch of threads, equal to the number of tasks we think we can handle simultaneously. Then we give each thread a different task.

How many threads do we want? If we are waiting a lot, we can handle many tasks at once!

Concurrency in Python

```
from concurrent.futures import ThreadPoolExecutor
from time import sleep

work = range(200)
def fn(x):
    sleep(1)
    return x

with ThreadPoolExecutor(50) as pool:
    results = pool.map(fn, work)

print(list(results))
```

Complexity

This is great when we have a list of items to process, in advance, and we can map over them.

Often, however, we need more complex dataflows.

Consider scraping: there is no list of urls to scrape, we might start with only *one* url! When the scraper scrapes that first url, it might generate dozens of other urls that need to be scraped. And each of those dozens of urls might generate dozens of others.

Communication

How could these parallelized scrapers communicate the new work to other scrapers?

Barcelona GSE Data Science Center

Queues

One powerful architecture is to use a queue.

A queue is exactly what it sounds like! It's a first-in-first-out list of data. It holds items of work that need to be done, like URLs that need to be scraped.

We pair the queue with a pool of "workers".

Each worker waits until an item becomes available on the queue.

As soon as an item becomes available, the worker processes it.

It's the Queue's responsibility to make sure no item goes to more than one worker.

Queues

Imagine if you were trying to organize a group todo list. In a classroom, for example, on the blackboard.

You would want some way for each person to get an item, and mark that they are “working on it.”

You wouldn’t want multiple people working on the same item at the same time.

You want another way for each person to mark that they are “done” working on the item. Then you can erase that item from the list!

You also want each person to be responsible for getting another item once they’ve finished their last one.

Barcelona GSE Data Science Center

Queues

Queues are extremely simple, powerful tools that are used in a lot of distributed computing!

Modern software companies will run separate servers just for queues for their systems. They run queue libraries such as: Kafka, RabbitMQ, 0MQ, Redis, Celery, etc.

However, Python also has simple queues implemented natively!

Barcelona GSE Data Science Center

Python Queues

The most important methods to understand with Python queues:

- queue.get() – this gets an item out of the queue. If there is no item, it will block until an item is avaialable.
- queue.put() – this puts an item onto the queue.
- queue.task_done() – this tells the queue that one (non-identified) task has been finished.
- queue.join() – this blocks until “task_done” was called as many times as “put” (i.e. all the tasks are finished).

Threads vs Processes

Remember the difference between threads and processes?

Threads share memory, processes do not.

Multithreading can, therefore, be very dangerous! With multiple threads sharing the same memory, they might erase each other's work.

Python handles this via Global Interpreter Lock. This means that the “interpreter”, the thing that runs and executes code, can only be accessed by one thread at a time.

This is a bit brutal, but it ensures multithreading is easy and works the way you want it.

The downside, however, is that you can’t achieve true parellelism!

For concurrent workloads with a lot of waiting, this if fine. However, if we want to literally perform actions at the same time, as we do in cpu-bound parallelism, we need another way.

Barcelona GSE Data Science Center

CPU-Bound Parellelism

CPU-bound parallelism in Python is done with multiprocessing.

When Python performs multiprocessing, it launches a whole new Python process at the operating system level.

However, because every process has its own memory, Python needs to copy everything from the current process over to the new one.

Serialization

To send things from memory from one process to another takes:

1. The ability to turn this thing into something “non-Python” (i.e. a string)
2. The time required to do that

Serialization is the name for taking something in memory, be it a function or a piece of data, and turning it into a string. This is done so that the information can be sent “over a wire” – to another process, or even to another computer!

Barcelona GSE Data Science Center

Multiprocess communication

How do we deal with communication between processes in Python?

The recommended way is with a queue!

The multiprocessing library comes with a special Queue implementation, just for this.

Different Kinds of Parallelism

Quick recap. In Python:

- I/O-bound parallelism, which is usually concurrency more than true parallelism, is performed via multithreading.
- CPU-Bound parallelism is performed via multiprocessing, which requires serialization.

Both can be achieved with exactly the same API via the conccurent.futures module. We will see this in the exercises.

Distributed Computing

When you have a lot of work to do, especially CPU-bound work, you can:

1. Distribute the computation among the different cores of one computer.
2. Distribute the computation among different computers.
3. Both.

Barcelona GSE Data Science Center

Distributed Computing

Distributed computing refers to the act of distributing work among different physical computers, separated by network.

Many modern abstractions which give us distributed computing also allow for parellelization on one computer.

Queues are also very important elements of distributed computing!

Barcelona GSE
Data Science Center