Fastest way to unzip a zip file in Python

So the context is this; a zip file is uploaded into a web service and Python then needs extract that and analyze and deal with each file within. In this particular application what it does is that it looks at the file's individual name and size, compares that to what has already been uploaded in AWS S3 and if the file is believed to be different or new, it gets uploaded to AWS S3.

The challenge is that these zip files that come in are huuuge. The average is 560MB but some are as much as 1GB. Within them, there are mostly plain text files but there are some binary files in there too that are huge. It's not unusual that each zip file contains 100 files and 1–3 of those make up 95% of the zip file size.

At first I tried unzipping the file, in memory, and deal with one file at a time. That failed spectacularly with various memory explosions and EC2 running out of memory. I guess it makes sense. First you have the 1GB file in RAM, then you unzip each file and now you have possibly 2-3GB all in memory. So, the solution, after much testing, was to dump the zip file to disk (in a temporary directory in `/tmp`) and then iterate over the files. This worked much better but I still noticed the whole unzipping was taking up a huge amount of time. **Is there perhaps a way to optimize that?**

Baseline function

First it's these common functions that simulate actually doing something with the files in the zip file:

```
def _count_file(fn):
    with open(fn, 'rb') as f:
        return _count_file_object(f)

def _count_file_object(f):
    # Note that this iterates on 'f'.
    # You *could* do 'return len(f.read())'
    # which would be faster but potentially memory
    # inefficient and unrealistic in terms of this
    # benchmark experiment.
    total = 0
    for line in f:
        total += len(line)
    return total
```

Here's the simplest one possible:

```
def f1(fn, dest):
    with open(fn, 'rb') as f:
        zf = zipfile.ZipFile(f)
        zf.extractall(dest)

    total = 0
    for root, dirs, files in os.walk(dest):
        for file_ in files:
            fn = os.path.join(root, file_)
            total += _count_file(fn)
    return total
```

If I analyze it a bit more carefully, I find that it spends about 40% doing the `extractall` and 60% doing the looping over files and reading their full length.

First attempt

My first attempt was to try to use threads. You create an instance of `zipfile.ZipFile`, extract every file name within and start a thread for each name. Each thread is given a function that does the "meat of the work" (in this benchmark, iterating over the file and getting its total size). In reality that function does a bunch of complicated S3, Redis and PostgreSQL stuff but in my benchmark I just made it a function that figures out the total length of file. The thread pool function:

```
def f2(fn, dest):

    def unzip_member(zf, member, dest):
        zf.extract(member, dest)
        fn = os.path.join(dest, member.filename)
        return _count_file(fn)

    with open(fn, 'rb') as f:
        zf = zipfile.ZipFile(f)
        futures = []
        with concurrent.futures.ThreadPoolExecutor(
            for member in zf.infolist():
                futures.append(
                    executor.submit(
                        unzip_member,
                        zf,
                        member,
                        dest,
                    )
                )
            total = 0
            for future in concurrent.futures.as_com
                total += future.result()
    return total
```

Result: ~10% faster

Second attempt

So perhaps the GIL is blocking me. The natural inclination is to try to use multiprocessing to spread the work across multiple available CPUs. But doing so has the disadvantage that you can't pass around a non-pickleable object so you have to send just the filename to each future function:

```
def unzip_member_f3(zip_filepath, filename, dest):
    with open(zip_filepath, 'rb') as f:
        zf = zipfile.ZipFile(f)
        zf.extract(filename, dest)
    fn = os.path.join(dest, filename)
    return _count_file(fn)



def f3(fn, dest):
    with open(fn, 'rb') as f:
        zf = zipfile.ZipFile(f)
        futures = []
        with concurrent.futures.ProcessPoolExecutor()
            for member in zf.infolist():
                futures.append(
                    executor.submit(
                        unzip_member_f3,
                        fn,
                        member.filename,
                        dest,
                    )
                )
            total = 0
            for future in concurrent.futures.as_comp
                total += future.result()
    return total
```

Result: ~300% faster

That's cheating!

The problem with using a pool of processors is that it requires that the original `.zip` file exists on disk. So in my web server, to use this solution, I'd first have to save the in-memory ZIP file to disk, then invoke this function. Not sure what the cost of that it's not likely to be cheap.

Well, it doesn't hurt to poke around. Perhaps, it could be worth it if the extraction was significantly faster.

But remember! This optimization depends on using up as many CPUs as it possibly can. What if some of those other CPUs are needed for something else going on in `gunicorn`? Those other processes would have to patiently wait till there's a CPU available. Since there's other things going on in this server, I'm not sure I'm willing to let on process take over all the other CPUs.

Conclusion

Doing it serially turns out to be quite nice. You're bound to one CPU but the performance is still pretty good. Also, just look at the difference in the code between `f1` and `f2`! With `concurrent.futures` pool classes you can cap the number of CPUs it's allowed to use but that doesn't feel great either. What if you get the number wrong in a virtual environment? Of if the number is too low and don't benefit any from spreading the workload and now you're just paying for overheads to move the work around?

I'm going to stick with `zipfile.ZipFile(file_buffer).extractall(temp_dir)`. It's good enough for this.

Want to try your hands on it?

I did my benchmarking using a `c5.4xlarge` EC2 server. The files can be downloaded from:

```
wget https://www.peterbe.com/unzip-in-parallel/hack
wget https://www.peterbe.com/unzip-in-parallel/symb
```

The `.zip` file there is 34MB which is relatively small compared to what's happening on the server.

The `hack.unzip-in-parallel.py` is a hot mess. It contains a bunch of terrible hacks and ugly stuff but hopefully it's a start.

Follow **@peterbe** on Twitter

Previous: Make .local domains NOT slow in macOS29 January 2018 Related by Keyword: Unzip benchmark on AWS EC2 c3.large vs c4.large29 November 2017 Mozilla Symbol Server (aka. Tecken) load testing06 September 2017 A neat trick to zip a git repo with a version number01 September 2017 Fastest Redis configuration for Django11 May 2017 Fastest cache backend possible for Django07 April 2017 Related by Text: Concurrent Gzip in Python13 October 2017 Unzip benchmark on AWS EC2 c3.large vs c4.large29 November 2017 Fastest way to match a filename's extension in Python31 August 2017 Python package path when executed elsewhere14 December 2004 How to do performance micro benchmarks in Python24 June 2017