PyPA: Allow `pip` to Download in Parallel

Nguyễn Gia Phong

March 31, 2020

1 About Me

1.1 Name

My legal name is Nguyễn Gia Phong, however on the Internet, I often go by the alias of McSinyx. Under this username, one may easily find me on PyPI, GitHub, Reddit, Twitter as well as Matrix (and IRC via the Matrix bridge).

1.2 Enrollment

At the moment, I am a second-year bachelor student majoring in ICT at USTH[1], Hà Nội, Việt Nam. If everything goes as planned, I will graduate in autumn, 2021.

1.3 Contact Information

My main email address is mcsinyx@disroot.org.

1.4 Timezone

I currently live in Hà Nội, Việt Nam, which uses Indochina Time (UTC+07:00).

1.5 Experience

In a long summer vacation in 2015, I picked up *Think Python 2e* by Allen B. Downey to overcome boredom. Ever since, Python has been my language of choice to play around with TUI (ncurses, argparse), indie game development (pygame, ModernGL) which involves some basic networking (socket) and concurrency (mostly using thread).

[1] `https://usth.edu.vn/en/`

Many of my pet projects are distributed on PyPI. Over the years, in order to solve personal distribution problems, I ended up reading documentations as well as the source code of various projects under the hood of PyPA, namely setuptools and pip, and thus was slowly gaining familarity with the them, especially after finding a way to distribute a Cython extension module earlier this year. At the same time, I also grew interest in the codebase of `pip` and I wish to be able help improving it.

Outside of the Python ecosystem, I am fluent in Lua (after refactoring a collection of desktop widgets to be asynchronous) and have a background in competitive programming and Lisp. Please visit my GitHub profile for a more complete list of my activities and my developer story on Stack Overflow[2] for highlights.

2 Code Contribution

So far, my contributions to `pip` are solely minor behavioral enhancements (GH-7828[3] and GH-7929[4]) and a tiny documentation whitespace nitpick (GH-7928).

3 Project Information

3.1 Sub-Org Name

Python Packaging Authority (PyPA)

3.2 Project Abstract

Parallel downloading is an enhancement that has been long-awaited[5]. Even back then, the feature could have helped reduce the installation time (of an end-package, as well as of dependencies for isolated wheel builds). Now that a new dependency resolver with backtracking is being implemented, downloading packages in parallel may significantly improve the resolver performance.

[2] `https://stackoverflow.com/cv/mcsinyx`

[3] Fail early when install path is not writable.

[4] Use better temporary files mechanism.

[5] The feature was proposed back in 2013: GH-825

3.3 Detailed Description

3.3.1 Rationale and Goals

At the time of writing, all `pip`'s operations are synchronous. For a few tasks such as downloading and installation, speedups may be achieved by running them asynchronously, with the I/O time in parallel. In this document, I am trying to propose an approach to achieve the earlier. Python 3's couroutines and tasks might be more natural to be implemented; however since `pip` will only drop Python 2 support in a distant future, by either multithreading or multiprocessing might be a more suitable choice.

Based on (manual) routine call analysis[6], the downloading is deeply nested within the dependency resolver. For parallel download to be future-proof, the implementation should not rely on the relationships between resolutions that want to fetch the packages. To achieve this, it might be a good idea to isolate the networking calls from the resolver. In other words, routines involving fetching packages in `pip._internal.operations.prepare` need to be moved to `pip._internal.network.download`, inside a stateful batch downloader. This class shall hold a collection of unique URLs needed for the current tier of resolution, i.e. the ones whose dependencies are known.

After the networking code gets refactored, the dependencies resolution procedure can be restructured as follows:

1. Gather the collection of requirements, from which construct the batch downloader.

2. Start batch downloading of known required packages.

3. Partially resolve the fetched requirements. If new dependencies arise, go back to step 1.

Since a collection of unique URLs is obtain, a job pool can download them in parallel without the fear of multiple jobs fetching the same package and wasting bandwidth. Furthermore, it will not be too difficult to display all current download in progress, and thus the backward compatibility for user interface is maintained.

When the implementation is completed, careful testing and benchmarking must be undertaken, since both downloading and dependency resolution are essential tasks of `pip`. Moreover, with the new dependency resolver being implemented, my changes need to be splitted into a series of self-containing patches for the ease of integration.

[6] `https://gist.github.com/McSinyx/990a2d1ed39a3e0e884e116a2daf0ecb`

3.3.2 Existing Approaches

The discussion above is loosely based on Omer Katz's attempt in GH-3981. The main problems blocking the pull request from being ready for merging are (1) parallelization of the user interface (diplay and clean up of progress bars) and (2) vendoring of the `futures` package. To tackle (1), I will try to draw the progress bars from the main thread, with the status information about each downloader gathered in the downloader collection. On (2), 'multiprocessing.dummy' should be able provide the thread pool.

Other than the case `pip`, package managers usually do not have to fetch the whole package to get dependency information. Therefore, the only similar model I have yet to found are package managers for golang (namely the builtin `go get` and `dep`). From there documentations, it does not seem that there perform downloading in parallel, but `dep` do admit that fetching the whole package just to retrieve the information about dependencies is slowing down the program.[7]

3.3.3 Implementation Details

After this project is carried out, the new resolve-download proces, at the beginning of install/download/wheel should look like the following

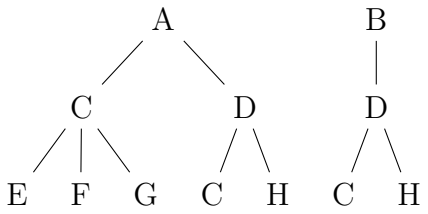
1. Initialize of the downloader collection.
2. Attempt to resolve the given requirements.
3. Prepare the unsolved requirements, in the process extract URLs and add them to the downloader collection.
4. Download all packages available in the collection, then clear it.
5. Get the new list of requirements. If conflicts arise, return to step 2 (assuming resolvelib is used). If the list is non-empty, return to step 3.

The downloader collection can be any kind of mapping, e.g. a wrapper around `dict` of URL to downloader. The downloader should have the information about the progress for display, as well as the resulting file object to be used by the resolver. As the packages being written, the downloader must be able to draw the progress bars to the output stream.

Concerning the dependency resolver, after each attempt to solve a requirement, we can obtain from it a sequence of the requirement's dependencies, then we only stop to download when there is not any requirement that we

[7] FAQ: Why is dep slow?

have not checked. For example, given the following dependency tree and assume the most trivial case where there is no conflict:



To install A and B, at first `pip` should download these two packages. As A is being resolved, first C is seen and added to the batch downloader. In order to continue the resolution of A, we put the undownloaded C in the place of the dependency to be returned as a promise-like object.[8] Next, we do the same thing to D and move to resolve B. Since the sole dependency of B is already in the collection, it will not be duplicated; and as the first *tier* finishes, we download C and D.

When the downloads finishes, `pip` returns to the resolution of C and find E, F and G and add them to the (now-emptied) batch downloader. As D is being resolved, H *and* C are also added to the same tier. However, since C's wheel is already cached, only E, F, G and H will be downloaded in the next batch. Afterwards, E, F, G and H can be trivially resolved, then the resolver can finish the resolution process recursively.

3.3.4 Backup Plans

In order to isolate the networking code from the preparation operations, admittedly, I will have to refactor some fragile code that I may not fully understand. Moreover, with the on-going refactoring of the resolver, there may be integration issues that will block me from progressing with this project. In the worst case that the original plan does not seem to work out, it is tempting to try to make `pip` (and package repositories like PyPI) use dependencies metadata for resolution.

However, the scope of this is much larger than my original goal. As suggested by Pradyun Gedam, it might be much wiser to switch focus to improve `pip`'s argument parsing logic, which I have more experience on. At the time of writing, `pip` has been using the low-level parser `optparse`, which led to cumbersome code constructing of callables using `functools.partial` scattered all over `pip._internal.cli.cmdoptions` for common options. Additionally,

[8] I mean a kind of object that hold the reference to the package after download, for convenient the batch download can be a collection of these.

with `pip` growing in size and complexity (GH-6221), many subcommands' options start to overlap and conflict (e.g. GH-4575 and its linked issues).

GH-4659 suggests to move to `click` for a better abstraction as a start to enhance the parsing logic. From there, it is possible to build the argument parser with a more proper (inheritance) base and more straightforward implementation of edge cases like mutually exclusive options and more explicit control of execution path.

3.4 Weekly Timeline

Community Bonding *May 4–May 31*

- Research other languages' package managers' approach on handling download with recursive dependency resolution.
- Watch `pip`'s issue tracker to familiarize myself with the conventions used by the project.
- Patch related (or non-related but doable) issues to learn more about `pip`'s codebase.

Week 1 *June 1–June 7:* Move package download calls from the preparation operation to the networking package.

Week 2 *June 8–June 14:* Re-structure the resolution process into three steps (prepare–download–resolve).

Week 3 *June 15–June 21:* Implement a basic downloader collection that supports thread pool.

Week 4 *June 22–June 28:* Swap in the downloader collection and apply a thread pool map in place of batch download (as an experimental option).

Week 5 *June 29–July 5:* Write unit and functional tests (most unit tests should be adapted during the execution of the previous steps though).

Week 6 *July 6–July 12:* Fix newly discovered bugs (~~if any~~ there will be).

Week 7 *July 13–July 19:* Make the parallel behavior the default and add CLI arguments to disable it as well as set the maximum number of connections.

Week 8 *July 20–July 26:* Benchmark and optimize.

Week 9 *July 27–August 2:* Document relavant pages, e.g. GH-3116.

Week 10 *August 3–August 9:* Buffer week and stretch plan.

Week 11 *August 10–August 16:* Buffer week and stretch plan.

Week 12 *August 17–August 23:* Buffer week and stretch plan.

Week 13 *August 24–August 30:* Wrap up the project.

4 Other Commitments

If nothing changes, I will still have classes until the middle of June[9]. While I cannot predict everything, from my own experience, usually I can still balance school and free software development.

[9] `https://usth.edu.vn/en/timetable/ict/bachelor-2-ict-18.html`