parallelize Documentation

Release 0.1.0

Ismail Uddin

Feb 10, 2019

Contents:

1 API documentation **3**
1.1 API 3

2 Considerations with using `multiprocessing` **5**
2.1 Potential considerations 5

3 Quickstart **7**

4 Usage **9**

parallelize

A Python module for quickly parallelising functions using Python's `multiprocessing` module.

—

CHAPTER 1

API documentation

Documentation for the functions of *parallelize*

1.1 API

CHAPTER 2

Considerations with using `multiprocessing`

Python's `multiprocessing` module enables parallel processing by spawning new processes, within which your code can run. Once completed, the processes are joined back into the main process. This approach introduces a number of potential problems discussed below.

2.1 Potential considerations

2.1.1 Functions must support pickling

The method by which Python spawns new processes includes pickling the object using the `pickle` module. Certain types of classes and objects may not be able to be pickled, in which case using this module or the `multiprocessing` will fail.

2.1.2 Large outputs add a significant overhead

Since parallel processing is achieved by spawning new processes, which have their own memory, the output of the function needs to be transferred as well back to the main process. If this object is particularly large, the transfer process can become very slow. This can sometimes mean despite using multiple cores to speed up the computation, the data transfer ends up taking as long or longer to take place. The end result being no benefit from parallel processing, or worse an even longer wait time.

To circumvent this issue, `parallelize` offers the option to pickle the output to a temporary file using the `cPickle` module and the fastest pickling protocol. The files are then read back by the main process and merged into one file. This can sometimes offer a speed up, however the process of pickling an object is still relatively slow.

This option can be enabled using the argument `write_to_file` in the `parallelize.parallel()` function:

```
>>> from parallelize import parallelize

>>> def foo(iterable: list) -> int:
...     output = 0
```

(continues on next page)

(continued from previous page)

```
...     for i in iterable:
...         output = i**4
...     return output

>>> parallelize.parallel(foo, numbers, n_jobs=6, write_to_file=True)
```

This module allows you to quickly convert a Python function into one taking advantage of parallel processing using Python's `multiprocessing` module.

The `multiprocessing` module works by spawning a new process for your function, which will automatically be assigned to a different processor core to take advantage of multi-core setup.

The key advantage of using this module is by overcoming the need to re-write all the boilerplate code for setting up the `multiprocessing` module's `Queue` or `Pool` class.

CHAPTER 3

Quickstart

`parallelize` is designed to operate on functions which accept an iterable to operate over. Thus, to use this package, your code must be encapsulated in a function which accepts as a first argument the iterable being a list.

In the below example, an example function `foo` which takes about 21 seconds to compute on a single core, can be sped up by nearly 4 times using a six-core CPU using `parallelize.parallel()`:

```
>>> from parallelize import parallelize

>>> def foo(iterable: list) -> int:
...     output = 0
...     for i in iterable:
...         output = i**4
...     return output

>>> numbers = list(range(50000000))
>>> %time foo(numbers)
Wall time: 21.5 s
>>> parallelize.parallel(foo, numbers, 6)
Completed 'parallel' in 6.2743 secs
```

The ability to speed up your function using parallel processing in Python is dependent on many different factors. For an overview, see *Considerations with using multiprocessing*.

CHAPTER 4

Usage

More advanced functions can also be passed to `parallelize.parallel()` using the `args` and `kwargs` arguments. Simply add your arguments as a tuple within the `args` argument, and/or the keyword arguments as a dictionary in the `kwargs` argument.

If your function returns a large object such as a NumPy array, you may find despite the computation being sped up, returning that data back to the main process will be quite slow. One possible solution is to enable the `write_to_file` argument which writes the data to disk using `cPickle`, and reads back in the main process. This process is also heavily dependant on your main disk being sufficiently fast (i.e. an SSD).

`parallelize.parallel()` will aim to return your output result as a flattend list. That is if your function returns a list ordinarily, the lists from each spawned process will be merged into one big list as would be done if run using a single process. If your function simply returns a single object, the outputs of all the processes will be returned as an ordered list. The list returned is always sorted according to the iterable provided, i.e. in the order the function would encounter the items if run with a single process.

If your function does not return any output, i.e. it makes a request to a server or writes to disk, the output will be a series of `None` s in a list.