PySpeedup Documentation

Release 0.1.0.3

Chris Dusold

February 26, 2017

Contents

Contents:

CHAPTER 1

Indices and tables

- genindex
- modindex
- search

CHAPTER 2

About

A collection of speed ups that make full use of multiprocess and hyperthreaded systems. Caveat aliquam, here there be experiments.

Available from PyPI, and easily installed through *pip install PySpeedup*. Documentation available at Read The Docs and source available on Github.

CHAPTER 3

Concurrent Module

A module containing algorithm parallelization classes and methods.

The motivation for this module was to simplify the often overly convoluted structure of the Python `multiprocessing` for common and simple uses of parallelization.

Concurrent Buffer

class `pyspeedup.concurrent.`**`Buffer`**(*generator*, *buffersize=16*, *haltCondition=<function uniformly-NonDecreasing>*)

An implementation of a concurrent buffer that runs a generator in a separate processor.

Note: The default halting condition requires the values be uniformly non decreasing (like the primes, or positive fibonnaci sequence). This halting condition currently stops the search once the value reached is greater than or equal to the one being searched for.

The resultant buffered object can be referenced as a list or an iterable. The easiest way to use this class is by utlizing the utility function *`buffer()`*.

For example, one can use it in the following way:

```
>>> @buffer(4)
... def count():
...     i=0
...     while 1:
...         yield i
...         i+=1
...
>>> count[0]
0
>>> count[15]
15
>>> for v,i in enumerate(count):
...     if v!=i:
...         print("Fail")
...     if v==5:
...         print("Success")
...         break
...
Success
```

It can also be used as a generator by calling the object like so:

```
>>> for v,i in enumerate(count()):
...     if v!=i:
...         print("Fail")
...     if v==5:
...         print("Success")
...         break
...
Success
```

The sequence generated is cached, so the output stored will be static.

To create your own halting condition, you need to provide a function with the first argument taken in as the buffer object, the second argument for the item that we're deciding whether we've passed (or given up on) during the search, and the third argument being how many times we've checked the halting condition during this search.

For example, if your buffered sequence isn't uniformly non decreasing, but is instead absolutely non decreasing, you could create the following halting condition function:

```
>>> def absolutelyNonDecreasing(buffer, item, attempts):
...     if abs(buffer._cache[-1])>abs(item):
...         return True
...     return False
...
>>> @buffer(haltCondition = absolutelyNonDecreasing)
... def complexSpiral():
...     i = 1
...     while True:
...         yield i
...         i *= 1.1j
...
>>> complexSpiral[1]
1.1j
>>> -1.21 in complexSpiral
True
```

Be careful in creating your halting condition, as if it is false for the sequence you are buffering, you may not see expected results, or you may find your program in an infinite loop. Be sure to consider asymptotes and other possibilities in your results. It may not be a bad idea to have it bail out after a certain number of attempts.

Note: As of yet all values are stored in a list on the backend. There is no memory management built in to this version, but is planned to be integrated soon. Be careful not to accidentally cache too many or too large of values, as you may use up all of your RAM and slow down computation immensely.

`pyspeedup.concurrent.`**`buffer`**(*buffersize=16*, *haltCondition=<function uniformlyNonDecreasing>*)

A decorator to create a concurrently buffered generator.

Used with `@buffer([buffersize,[haltCondition]])` as described in *Buffer*'s documentation.

Concurrent Cache

class `pyspeedup.concurrent.`**`Cache`**(*func*)

An asynchronous cache implementation. Maintains multiple recursive calls stably.

The resultant object operates just like a function, but runs the code outside the main process. When calls are started with `apply_async()`, a new process is created to evaluate the call.

A simple cache can reduce recursive functions such as the naive Fibonacci function to linear time in the input space, whereas a parallel cache can reduce certain problems even farther, depending on the layout of the call and the number of processors available on a computer. The code below demonstrates using *Cache* as a simple cache:

```
>>> @Cache
... def fibonacci(n):
...     if n < 2: # Not bothering with input value checking here.
...         return 1
...     return fibonacci(n-1)+fibonacci(n-2)
...
>>> fibonacci(5)
8
```

Using cache to take advantage of the ability to handle recursion branching, that same code would become:

```
>>> @Cache
... def fibonacci(n):
...     if n < 2: # Not bothering with input value checking here.
...         return 1
...     fibonacci.apply_async(n-1)
...     fibonacci.apply_async(n-2)
...     return fibonacci(n-1)+fibonacci(n-2)
...
>>> fibonacci(100)
573147844013817084101L
```

Note: Be careful when picking how to call your functions if you are looking for speed. Given that the fibonacci sequence is roughly linear in dependencies with caching, there isn't a significant speedup. When in doubt, `cProfile` (or `profile`) are your friends.

A good use for this would be in less sequential computation spaces, such as in factoring. When a pair of factors are found, each can be factored asynchronously to find all the prime factors recursively. When a factor in a factor pair is found that are known to be prime, or otherwise has its factors known, then only one needs to be factored further. At this point, blindly branching and factoring will have one side yield the cached value, and the other creating a new process. Given the Fibonacci example above, this will happen on every call that isn't the first call, yielding to *n* processes being spawned and using system resources. Simply caching the naive Fibonacci function is just about the fastest way to use it.

To avoid unnecessary branching automatically, you can use the batch_async method similarly to the apply_async method, except each set of arguments, even if they're singular, must be wrapped in a tuple. Applying this to the Fibonacci function yields.

```
>>> @Cache
... def fibonacci(n):
...     if n < 2: # Not bothering with input value checking here.
...         return 1
...     fibonacci.batch_async((n-1,),(n-2,))
...     return fibonacci(n-1)+fibonacci(n-2)
...
>>> fibonacci(200)
453973694165307953197296969697410619233826L
```

This makes the branching optimal whenever possible. Race conditions might cause issues, but those caused by python's built in Manager cannot be mitigated easily. For the fibonnacci sequence, this will likely just revert the

computation to a mostly synchronous and sequential calculation, which is optimal for this version of calculating the Fibonacci sequence.

Note: There are much better algorithms for calculating Fibonacci sequence elements; some of which are better suited for this type of caching.

CHAPTER 4

Algorithm Module

CHAPTER 5

Memory Module

A module containing storage classes that maintain small RAM usage and original structure access order.

The motivation for this module was to provide constant size RAM usage while maintaining normal use of Python Dictionaries and possibly other structures for semi-big data, where it isn't large enough to warrant more big data centric solutions.

Disk Based Dictionary

class pyspeedup.memory.**DiskDict**(*file_basename*, *size_limit=1024*, *max_pages=16*, *file_location='/home/docs/.PySpeedup'*)

A dictionary class that maintains O(1) look up and write while keeping RAM usage O(1) as well.

This is accomplished through a rudimentary (for now) hashing scheme to page the dictionary into parts.

The object created can be used any way a normal dict would be used, and will clean itself up on python closing. This means saving all the remaining pages to disk. If the file_basename and file_location was used before, it will load the old values back into itself so that the results can be reused.

There are two ways to initialize this object, as a standard object:

```
>>> diskDict = DiskDict("sample")
>>> for i in range(10):
...     diskDict[i] = chr(97+i)
...
>>> diskDict[3]
'd'
>>> del diskDict[5]
>>> ", ".join(str(x) for x in diskDict.keys())
0, 1, 2, 3, 4, 6, 7, 8, 9
>>> "b" in diskDict
True
```

Or through context:

```
>>> with DiskDict("test") as d:
...     for i in range(10):
...         d[i] = chr(97+i)
...     print(d[3])
3
```

If there is a way to break dict like behavior and you can reproduce it, please report it to the GitHub issues.

Note: This class is not thread safe, nor is it process safe. Any multithreaded or multiprocessed uses of this class holds no guarantees of accuracy.

You can configure how this class stores things in a few ways.

The file_basename parameter allows you to keep multiple different stored objects in the same file_location, which defaults to .PySpeedup in the user's home folder. Using a file_basename of the empty string may cause a small slowdown if more than just this object's files are in the folder. Using a file_location of the empty string will result in files being placed in the environment's current location (i.e. what *os.getcwd()* would return).

The size_limit parameter determines how many items are kept in each page, and the max_pages parameter determines how many pages can be kept in memory at the same time. If you use smaller items in the dict, increasing either is probably a good idea to get better performance. This setting will only use about 128 MB if standard floats or int32 values. Likely less than 200 MB will ever be in memory, which prevents the RAM from filling up and needing to use swap space. Tuning these values will be project, hardware and usage specific to get the best results. Even with the somewhat low defaults, this will beat out relying on python to use swap space.

Disk Based List

class pyspeedup.memory.**DiskList**(*file_basename*, *size_limit=1024*, *max_pages=16*, *file_location='/home/docs/.PySpeedup'*)

A list class that maintains O(k) look up and O(1) append while keeping RAM usage O(1) as well. Unfortunately, insert is O(n/k).

This is accomplished through paging every size_limit consecutive values together behind the scenes.

The object created can be used any way a normal list would be used, and will clean itself up on python closing. This means saving all the remaining pages to disk. If the file_basename and file_location was used before, it will load the old values back into itself so that the results can be reused.

There are two ways to initialize this object, as a standard object:

```
>>> diskList = DiskList("sample")
>>> for i in range(10):
...     diskList.append(i)
...
>>> diskList[3]
3
>>> ", ".join(str(x) for x in diskList)
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
>>> del diskList[5]
>>> ", ".join(str(x) for x in diskList)
0, 1, 2, 3, 4, 6, 7, 8, 9
```

Or through context:

```
>>> with DiskList("test") as d:
...     for i in range(10):
...         d.append(i)
...     print(d[3])
3
```

If there is a way to break list like behavior and you can reproduce it, please report it to the GitHub issues.

Note: This class is not thread safe, nor is it process safe. Any multithreaded or multiprocessed uses of this

class holds no guarantees of accuracy.

You can configure how this class stores things in a few ways.

The file_basename parameter allows you to keep multiple different stored objects in the same file_location, which defaults to .PySpeedup in the user's home folder. Using a file_basename of the empty string may cause a small slowdown if more than just this object's files are in the folder. Using a file_location of the empty string will result in files being placed in the environment's current location (i.e. what *os.getcwd()* would return).

The size_limit parameter determines how many items are kept in each page, and the max_pages parameter determines how many pages can be kept in memory at the same time. If you use smaller items in the list, increasing either is probably a good idea to get better performance. This setting will only use about 64 MB if standard floats or int32 values. Likely less than 200 MB will ever be in memory, which prevents the RAM from filling up and needing to use swap space. Tuning these values will be project, hardware and usage specific to get the best results. Even with the somewhat low defaults, this will beat out relying on python to use swap space.

p

B

C

D

P