Streampie Documentation

Release 0.1

Luka Malisa

Apr 20, 2016

Contents

Streampie is a tiny library for simple and parallel execution of job processing tasks. The project heavily draws both concepts and code from the awesome stream.py project by Anh Hai Trinh. However, it is a leaner, cleaner re-implementation with the addition of simple distributed computation.

Streampie is released under the *MIT License*.

Contents:

CHAPTER 1

Getting Started

1.1 Installing

You can install the library through `pip`

```
pip install streampie
```

or head over to the github repository streampie. The whole library is contained in a single file.

1.2 Streams

Streams (*Stream*) are the basic class of the library. Each stream can be thought of as a class that takes an element from the input iterator, performs some work on the element, and then passes it on to the next stream in line. When working with streams, we overload the >> operator as it intuitively captures the notion of "passing on" the results from one stream into another.

Let's first start by including the library

```
In [1]: from streampie import *
```

To illustrate the concept of streams and processors, let's take the following example

```
In [2]: [0, 1, 2, 3] >> take(2) >> list
Out[2]: [0, 1]
```

In this example, we took our list and passed it to the *take* processor that took the first two elements of the list and discarded the rest. The final outcome was then converted to a plain list. Another common task in stream processing is splitting inputs into equally-sized chunks. For that purpose we can use *chop*.

```
In [3]: range(4) >> chop(2) >> list
Out[3]: [[0, 1], [2, 3]]
```

We are not limited to a single processor; we can chain arbitrarily many blocks

```
In [4]: range(4) >> chop(2) >> take(2) >> list
Out[4]: [[0, 1], [2, 3]]
```

For a full list of processors, see *streampie*. To illustrate where Streampie is useful, let's consider two examples that naturally benefit from paralellism.

1.3 URL Retrieval

Retrieving URLs is not a CPU-intensive task. To retrieve URLs in parallel (with four threads), we can utilize the *ThreadPool* to code the following program

```
import urllib2
from streampie import *

URLs = [
   "http://www.cnn.com/",
   "http://www.bbc.co.uk/",
   "http://www.economist.com/",
   "http://nonexistant.website.at.baddomain/",
   "http://slashdot.org/",
   "http://reddit.com/",
   "http://news.ycombinator.com/",
]

def retrieve(wid, items):
   for url in items:
      yield url, urllib2.urlopen(url).read()

for url, content in URLs >> ThreadPool(retrieve, poolsize=4):
   print url, len(content)
```

1.4 Integer Factorization

The second example is integer factorization, which is CPU intensive. Running a *ThreadPool* would not result in large performance gains due to the python global lock. However, we can use *ProcessPool*. Let's first look at a simple, iterative solution.

```
# A set of integers, each a product of two primes
ints = [2498834631017, 14536621517459, 6528633441793, 1941760544137, 7311548077279,
        8567757849149, 5012823744127, 806981130983, 15687248010773, 7750678781801,
        2703878052163, 3581512537619, 12656415588017, 468180585877, 19268446801283,
        5719647740869, 11493581481859, 366611086739]

def factor(n):
   """
      Integer factorization.
   """
   result = set()
   for i in range(1, int(n ** 0.5) + 1):
      div, mod = divmod(n, i)
      if mod == 0:
         result |= {i, div}
   return sorted(list(result))[:-1]

print map(factor, ints)
```

The program just iterates over the list of composite integers (each integer is a product of two primes). We can re-code the example in the following way.

```
from streampie import *

ints = [2498834631017, 14536621517459, 6528633441793, 1941760544137, 7311548077279,
        8567757849149, 5012823744127, 806981130983, 15687248010773, 7750678781801,
        2703878052163, 3581512537619, 12656415588017, 468180585877, 19268446801283,
        5719647740869, 11493581481859, 366611086739]

def factor(n):
   result = set()
   for i in range(1, int(n ** 0.5) + 1):
      div, mod = divmod(n, i)
      if mod == 0:
         result |= {i, div}
   return sorted(list(result))[:-1]

def do_work(wid, items):
   for i in items:
      yield factor(i)

print ints >> ProcessPool(do_work, poolsize=8) >> list
```

We now use 8 parallel local processes, and the task of factoring the numbers will be ~8 times as fast. But what if we want to compute the same task on a small cluster (e.g., two machines)? For that purpose, we can use the *DistributedPool*.

```
from streampie import *

ints = [2498834631017, 14536621517459, 6528633441793, 1941760544137, 7311548077279,
        8567757849149, 5012823744127, 806981130983, 15687248010773, 7750678781801,
        2703878052163, 3581512537619, 12656415588017, 468180585877, 19268446801283,
        5719647740869, 11493581481859, 366611086739]

def factor(n):
   result = set()
   for i in range(1, int(n ** 0.5) + 1):
      div, mod = divmod(n, i)
      if mod == 0:
         result |= {i, div}
   return sorted(list(result))[:-1]

print ints >> DistributedPool(factor) >> list
```

This code will now wait for workers to perform the job. We can start a single-process worker with

```
python streampie.py
```

CHAPTER 2

Streampie Module

class **Stream**(*obj=None*)

This is our generic stream class. It is iterable and it overloads the >> operator for convenience.

next()

class **take**(*n*)

Take the first `n` elements, and drop the rest.

```
>>> range(4) >> take(2) >> list
[0, 1]
```

class **takei**(*indices*)

Take only the elements whose indices are given in the list.

```
>>> range(4) >> takei([0, 1]) >> list
[0, 1]
```

class **drop**(*n*)

Drop the first *n* elements, and take the rest.

```
>>> range(4) >> drop(2) >> list
[2, 3]
```

class **dropi**(*indices*)

Drop only the elements whose indices are given in the `indices` list.

```
>>> range(4) >> dropi([0, 1]) >> list
[2, 3]
```

class **chop**(*n*)

Split the stream into `n`-sized chunks.

```
>>> range(4) >> chop(2) >> list
[[0, 1], [2, 3]]
```

class **map**(*function*)

Call the function `func` for every element, with the element as input.

```
>>> square = lambda x: x**2
>>> range(4) >> map(square) >> list
[0, 1, 4, 9]
```

class **filter**(*function*)

Return only the elements for which the predicate `func` evaluates to `True`.

```
>>> even = lambda x: x % 2 == 0
>>> range(4) >> filter(even) >> list
[0, 2]
```

class **apply** (*function*)
Call the function `func` for every element, with the element as arguments.

```
>>> sum = lambda x,y: x+y
>>> range(4) >> chop(2) >> apply(sum) >> list
[1, 5]
```

class **takewhile** (*predicate*)
Keep taking elements until the predicate `func` is `True`, then stop.

```
>>> range(4) >> takewhile(lambda x: x < 3) >> list
[0, 1, 2]
```

class **dropwhile** (*predicate*)
Keep dropping elements until the predicate `func` is `True`, then stop.

```
>>> range(4) >> dropwhile(lambda x: x < 3) >> list
[3]
```

class **prepend** (*prep_iterator*)
Prepend elements to a stream.

```
>>> range(4) >> prepend([10, 9]) >> list
[10, 9, 0, 1, 2, 3]
```

class **flatten** (*obj=None*)
Flatten an arbitrarily-deep list of lists into a single list.

```
>>> [0,[1,[2,[3]]]] >> flatten() >> list
[0, 1, 2, 3]
```

class **LocalPool** (*function*, *poolsize=None*, *args=[]*)
A generic class shared by all local (executed on the same machine) pools.

stop ()
Terminate and wait for all workers to finish.

class **ProcessPool** (*function*, *poolsize=None*, *args=[]*)
Create a process pool.

Parameters

- **function** (*int*) – Function that each worker executes
- **poolsize** (*int*) – How many workers the pool should make
- **args** (*list*) – List of arguments to pass to the worker function

A simple that calls the `sum` function for every pair of inputs.

```
>>> def sum(wid, items):
...   # wid is the worker id
...   # items is an iterator for the inputs to the stream
...   for x, y in items:
...     yield x + y
>>> range(6) >> chop(2) >> ProcessPool(sum) >> list
[1, 5, 9]
```

Note that the order of the output list is not guaranteed, as it depends in which order the elements were consumed. By default, the class creates as many workers as there are cores. Here is a more advanced examples showing `poolsize` control and passing additional arguments.

```
>>> def sum(wid, items, arg1, arg2):
...   # arg1 and arg2 are additional arguments passed to the function
...   for x, y in items:
...     yield x + y
>>> sorted(range(6) >> chop(2) >> ProcessPool(sum, poolsize=8, args=[0, 1]) >> list)
[1, 5, 9]
```

The function can yield arbitrarily many results. For example, for a single input, two or more yields can be made.

```
>>> def sum(wid, items):
...   for x, y in items:
...     yield x + y
...     yield x + y
>>> sorted(range(6) >> chop(2) >> ProcessPool(sum) >> list)
[1, 1, 5, 5, 9, 9]
```

class **ThreadPool** (*function*, *poolsize=None*, *args=[]*)

Create a thread pool.

Parameters

- **function** (*int*) – Function that each worker executes
- **poolsize** (*int*) – How many workers the pool should make
- **args** (*list*) – List of arguments to pass to the worker function

```
>>> def sum(wid, items):
...   # wid is the worker id
...   # items is an iterator for the inputs to the stream
...   for x, y in items:
...     yield x + y
>>> range(6) >> chop(2) >> ThreadPool(sum) >> list
[1, 5, 9]
```

class **StandaloneProcessPool** (*function*, *poolsize=None*, *args=[]*)

The standalone process pool is exactly like the *ProcessPool* class, other than the fact that it does not take any input, but constantly yields output.

Parameters

- **function** (*int*) – Function that each worker executes
- **poolsize** (*int*) – How many workers the pool should make
- **args** (*list*) – List of arguments to pass to the worker function

To illustrate, here is an example of a worker that constantly returns random numbers. Since there is no input stream, the pool needs to be manually terminated.

```
>>> import random
>>> def do_work(wid):
...   yield random.random()
>>> pool = StandaloneProcessPool(do_work)
>>> for x, r in enumerate(pool):
...   if x == 2:
...     pool.stop()
...     break
...   print r
```

```
0.600151963181
0.144348185086
```

class Job (*target_id*, *args=[]*)

This class is our unit of work. It it fetched by a *Worker*, it's `target` is executed, the result (`ret`) and exception (if any) is stored and sent back to the JobQueue.

Parameters

- **target_id** (*int*) – ID of the code to execute. See the source of *JobQueue.enqueue* for details.
- **args** (*list*) – List of arguments to pass to the worker function

class Worker (*host='localhost'*, *port=6379*, *db=10*)

The workhorse of our implementation. Each worker fetches a job from Redis, executes it, then stores the results back into Redis.

Parameters

- **host** (*str*) – Redis hostname
- **port** (*int*) – Redis port
- **db** (*int*) – Redis database number

run ()

In an infinite loop, wait for jobs, then execute them and return the results to Redis.

class JobQueue (*host='localhost'*, *port=6379*, *db=10*)

> **Warning:** The *JobQueue* flushes the selected Redis database! Be sure to specify an unused database!

The queue that allows submission and fetching of completed jobs.

Parameters

- **host** (*str*) – Redis hostname
- **port** (*int*) – Redis port
- **db** (*int*) – Redis database number

That being said, here is an example of how to use the queue.

```
>>> def sum(x, y):
...   return x + y
>>> q = JobQueue()
>>> q.enqueue(sum, (1, 2))
>>> q.enqueue(sum, (2, 3))
>>> q.enqueue(sum, (3, 4))
>>> q.finalize()
>>> for r in q:
...   print r.ret
3
5
7
```

next ()

enqueue (*target*, *args*)

Add a job to the queue.

Parameters

- **target** (*function*) – Function to be executed
- **args** (*list*) – Arguments provided to the job

finalize()
Indicate to the queue that no more jobs will be submitted.

class DistributedPool(*function*, *host='localhost'*, *port=6379*, *db=10*)
The distributed pool is a simple wrapper around the *JobQueue* that makes is even more convenient to use, just like *ProcessPool* and *ThreadPool*.

Parameters

- **host** (*str*) – Redis hostname
- **port** (*int*) – Redis port
- **db** (*int*) – Redis database number

First, on one machine let's start a single worker.

```
python streampie.py
```

We then execute:

```
>>> def mul(x, y):
...   return x * y
>>> range(4) >> chop(2) >> DistributedPool(mul) >> list
[0, 6]
```

stop()
Currently not implemented. Is it even needed?

CHAPTER 3

License

```
The MIT License (MIT)

Copyright (c) 2016 Luka Malisa

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

S