Conveyor

Release v0.0.2

May 17, 2020

Contents

Conveyor is a multiprocessing framework for creating intuitive data pipelines. With Conveyor, you can easily create stream-based pipelines to efficiently perform a series of operations on data, a task especially useful in the fields of machine learning and scientific computing. Creating a pipelined job is as easy as writing a function.

CHAPTER 1

Why use Conveyor?

The answer is simple: throughput. It's like putting a second load of laundry in the washer while a previous load is in the dryer. By breaking down a problem into smaller serial tasks, we perform the smaller tasks in parallel and increase the efficiency of whatever problem we're trying to solve.

CHAPTER 2

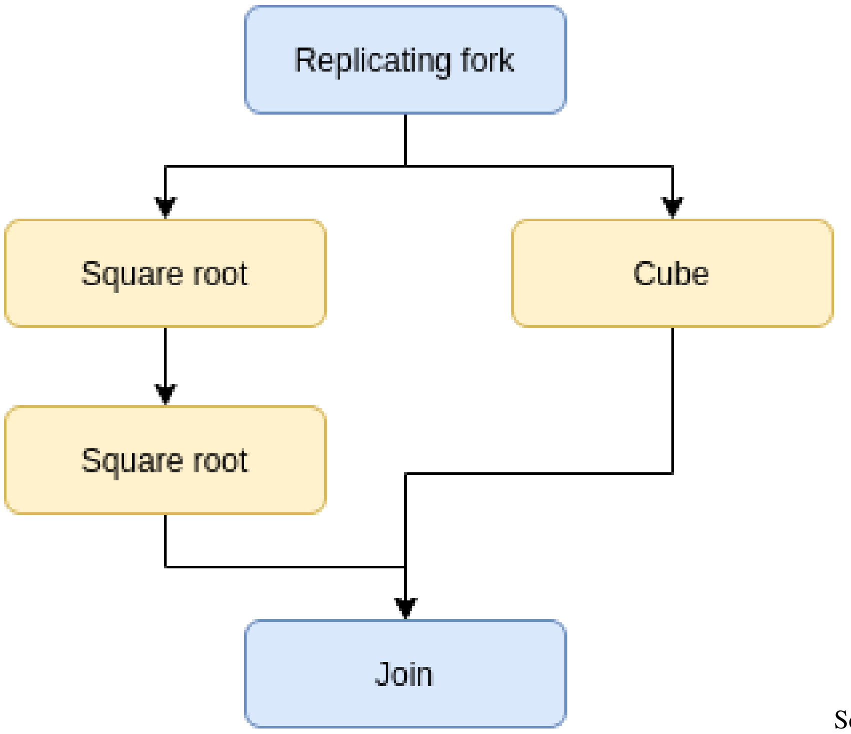
Installation

Install Conveyor with `pip install parallel-conveyor`.

CHAPTER 3

A quick and trivial example

Let's say we wanted to build a short pipeline that computed the fourth root of a number (done in two steps) and the cube of a number (in one step). In this case, we would describe this pipeline visually as such:



Schematic

To express it with Conveyor, we simply build the pipeline as follows

```
from conveyor.pipeline import Pipeline
from conveyor.stages import Processor, Pipe, ReplicatingFork, Join
from math import sqrt

```

(continues on next page)

(continued from previous page)

```python

def square_root(arg):
    return sqrt(arg)

def cube(arg):
    return arg ** 3

with Pipeline() as pl:
    # Duplicate the input
    pl.add(ReplicatingFork(2))

    # On first copy, compute the sqrt, on the second, the cube
    pl.add(Processor(square_root), Processor(cube))

    # On first copy, compute the sqrt, on the second, do nothing
    pl.add(Processor(square_root), Pipe())

    # Join the two data streams
    pl.add(Join(2))

    # Print the results
    pl.add(Processor(print))

    # Run the pipeline with three different inputs
    pl.run([16, 3, 81])
```

```
$ python3 sample.py
2.0
1.3160740129524924
3.0
4096
27
531441
```

CHAPTER 4

Contents

4.1 Quickstart guide

The entire idea behind Conveyor is to process data as if it is moving through a stream (in Conveyor, this is called a pipeline). Each pipeline contains a series of stages in which data is manipulated in some way before being passed out of the other end of the pipeline. Each stage runs as its own process, meaning that when processing large batches of data, each transformation of the data can happen in parallel.

4.1.1 Installation

Install Conveyor with

```
pip install parallel-conveyor
```

4.1.2 Pipelines

Pipelines form the core of Conveyor. They are the object through which data flows and is transformed. Initialize a pipeline with the following:

```python
from conveyor.pipeline import Pipeline

pl = Pipeline()
```

We can add stages to a pipeline using the `.add()` function, and run them using `.run()`. Additionally, we can use the `with` keyword to define a Pipeline to automatically free up resources used after we're done using the pipeline. You will often see this method used in our documentation.

```python
from conveyor.pipeline import Pipeline

with Pipeline() as pl:
    # Code involving the pipeline goes here
```

4.1.3 Stages

We now need to add stages to our pipeline. Stages will allow us to process our data and acheive throughput to enhance the performance of our program. Stages are added using the Pipeline object's `.add()` function. Stages come in 4 main types:

Processors

Processors are the Conveyor's logical compute cores. Each processor will accept some data as input, transform that input and optionally return an output. Processors can maintain an internal state. Each of these processors is user-defined and wrapped in its own Python instance, allowing parallel execution of multiple processors at the same time. A user can specify sets of processors that should execute serially as joined by pipes or sets of processors that should act in parallel as defined by forks.

We can create a Processor with `Processor(job)` where the argument `job` serves as a callback to function that will run in parallel. Each callback should be written such that it has a single argument that represents a single block of data to be processed. This argument can be an integer, string, tuple, object or any other data type. At the end of the function, we must return another single block of data to be handled by the next stage in the pipeline.

Writing jobs as functions is relatively straightforward: when the pipeline runs, each stage will receive a data block as a function argument, perform its processing, then spit out a result as a return value. Additionally, single-argument functions from other libraries or Python's standard library can be used (ie: `print()`).

```python
from conveyor.pipeline import Pipeline
from conveyor.stages import Processor

def job(arg):
    return arg + 1

with Pipeline() as pl:
    pl.add(Processor(job))
```

Pipes

Pipes serve as links between processors. They act as a stream, transporting data from one point in the pipeline to another. Pipes are implemented as a producer-consumer buffer where a leading processor acts as the producer and a trailing processor acts as a consumer. They don't use any processing power on their own and can be implicitly added by Conveyor to fill in gaps between processors.

```python
from conveyor.pipeline import Pipeline
from conveyor.stages import Processor, Pipe

def job(arg):
    return arg + 1

with Pipeline() as pl:
    pl.add(Processor(job))
    pl.add(Pipe())
    pl.add(Processor(job))
```

Equivalently, we could write the same code without the Pipe, as Conveyor will add it implicitly.

```python
from conveyor.pipeline import Pipeline
from conveyor.stages import Processor, Pipe

```

(continues on next page)

(continued from previous page)

```
def job(arg):
    return arg + 1

with Pipeline() as pl:
    pl.add(Processor(job))
    pl.add(Processor(job))
```

Forks

Forks serve as a way of splitting data travelling through the pipeline in two different ways. Forks can act in a *replicating* fashion, where they duplicate all data coming in and push it to many output processors or in a *balancing* fashion, where they perform a load-balancing operation, dividing input data blocks over a number of output processors.

Both types of forks take in a single argument: the number of output pipes to which they will attach. The number of stages added in parallel with the next call to `.add()` must match the number of output pipes at the previous stage. Processors, forks and pipes can all exist in parallel at the same stage in a pipeline.

Replicating Forks

Replicating Forks allow one processor to split output data into multiple copies so that multiple processors can then perform operations using the entire output data. For example, this would allow multiple different ML models to be trained and tested in parallel. The input-output numbering of the many-to-one relationship of forks is primarily user defined.

```
from conveyor.pipeline import Pipeline
from conveyor.stages import Processor, Pipe, ReplicatingFork, Join

def job(arg):
    return arg + 1

with Pipeline() as pl:
    pl.add(ReplicatingFork(2))
    pl.add(Processor(job), Pipe())
```

Balancing Forks

Balancing Forks allow one processor to balance a stream of data over multiple consumer processors. This will serve to minimize the effect of pipe stalling for larger data sets. The input-output numbering of the many-to-one relationship of forks is primarily determined by pipe stalling detected at runtime and the number of physical cores available.

```
from conveyor.pipeline import Pipeline
from conveyor.stages import Processor, Pipe, BalancingFork, Join

def job(arg):
    return arg + 1

with Pipeline() as pl:
    pl.add(BalancingFork(2))
    pl.add(Processor(job), Pipe())
```

Joins

Joins allow multiple processors to combine their data streams into one logical pipe. The combined stream can then be forked again for the next step of the pipeline, processed by a single processor, or serve as output at the end of the pipeline.

This means that the outputs from the previous parallel stages can be interleaved arbitrarily. Joins are useful when output from multiple different processors need to be serialized and work in a first-come-first-serve manner.

```
from conveyor.pipeline import Pipeline
from conveyor.stages import Processor, Pipe, ReplicatingFork, Join

def job(arg):
    return arg + 1

with Pipeline() as pl:
    pl.add(ReplicatingFork(2))
    pl.add(Processor(job), Pipe())
    pl.add(Join(2))
    pl.add(Processor(print))
```

4.1.4 Running a pipeline

The final step is to run our pipeline that we just created.To do so, we call `Pipeline.run()` on an array of input data objects, where each item will be passed through the pipeline in the same order as in the array. At any given time, multiple items in the array will be in the pipeline, albeit at different stages. Running the following trivial example:

```
from conveyor.pipeline import Pipeline
from conveyor.stages import Processor, Pipe, ReplicatingFork, Join

def job(arg):
    return arg + 1

with Pipeline() as pl:
    pl.add(ReplicatingFork(2))
    pl.add(Processor(job), Pipe())
    pl.add(Join(2))
    pl.add(Processor(print))
    pl.run([2, 9, 11])
```

Yields this output (separated by new lines): `3 10 12 2 9 11`.

4.2 Efficiency guide

If you're using Conveyor, you're using it because you want to squeeze every possible drop of performance out of your application. This should serve as a short guide of performance-breaking scenarios and how to avoid them.

4.2.1 Run pipelines using `with` statements

Pipelines can be run in two different ways, as demonstrated with the code snippets below:

```
pl = Pipeline()
pl.add(job)

pl.run([data, data2, data3])
pl.run([data4, data5, data6])
```

or…

```
with Pipeline() as pl:
    pl.add(job)

    pl.run([data, data2, data3])
    pl.run([data4, data5, data6])
```

When running a pipeline multiple times, Conveyor encourages users to use the second option described above. In the first case, heavyweight processes are created and killed (called 'opening' and 'closing' the pipeline, respectively) at the start and end of each invocation of `.run()`. This is disadvantageous, because creating and killing processes takes a large length of time. It would be much better to create the processes in the pipeline, use them on the first invocation of `.run()`, keep them running, and then use them again on the second invocation of `.run()`. This is what the second case does, where the pipeline is implicitly opened and closed at the start and end of the `with` statement.

4.2.2 Use shared memory for read-only data

Passing large quantities of data between processors is expensive. It's advantageous to use a read-only memory cache whenever feasible. See the *shared memory guide* for more details.

4.3 Shared memory

Note: the `common_memory` feature should be considered "very alpha" in that its specification and functionality are likely to change without warning. Additionally, `common_memory` requires Python 3.8.

4.3.1 Overview

In very specific instances, it's important to share memory across pipeline jobs and the main process without having to pass it through the pipeline. This is especially useful for when one needs to access variables or data "out of band" in that they are not part of the data stream on which the pipeline operates. Some use cases for this might be:

Collecting metrics

For measuring processor metrics, it would make sense to maintain a global database, rather than returning a metrics object describing the operation of the processor. It's best to keep control and data variables separate for both programmer's sanity and for the sake of saving inter-processor bus bandwidth.

Maintaining inter-processor state

In certain situations, multiple processors must be made stateful and must refer to a state variable stored elsewhere such that all processors can read it.

Large, static blocks of data (ie: a pandas DataFrame)

This is especially true for training machine learning datasets. While it is possible to send a large DataFrame object to multiple processors via a replicating fork, a better idea would be to use a global read-only store. This would prevent duplication of large quantities of redundant data.

4.3.2 Common pitfalls

It can be tempting to declare a global variable in your main application and then access it from within job function definitions. While this won't throw an error, it is important to remember that while Conveyor processors are written as functions, they are ultimately run as processes with their own view of a virtual memory space. This means that each processor, while all sharing a global variable name, will have its own independent copy of that variable.

Additionally, when modifying memory shared between processors, race conditions can occur. If concurrent reads/writes occur on shared memory, be sure to surround variable accesses with appropriate locks. Needless to say, these locks should also live in shared memory.

4.3.3 Conveyor's `common_memory` functionality

When creating a pipeline, one can pass an optional argument of `shared_memory_amt`. This is the number of bytes in memory to allocate for global access. You can access this data from within a processor with `shared_memory.SharedMemory(name=common_memory)`.

```python
from conveyor.pipeline import Pipeline
from conveyor.stages import Processor
from conveyor import common_memory

from multiprocessing import shared_memory

def worker1_task(args):
    shmem = shared_memory.SharedMemory(name=common_memory)
    buffer = shmem.buf
    buffer[:4] = bytearray([00, 11, 22, 33])
    shmem.close()

    return args

def worker2_task(args):
    shmem = shared_memory.SharedMemory(name=common_memory)
    buffer = shmem.buf
    buffer[0] = 44
    shmem.close()

    return args

def cleanup_task(args):
    shmem = shared_memory.SharedMemory(name=common_memory)
    import array
    print(array.array('b', shmem.buf[:4]))
    assert shmem.buf[0] == 44
    assert shmem.buf[1] == 11
    assert shmem.buf[2] == 22
    assert shmem.buf[3] == 33

    shmem.close()
```

(continues on next page)

(continued from previous page)

```
    shmem.unlink()

    return args

pipeline = Pipeline(shared_memory_amt=10)
pipeline.add(Processor(worker1_task))
pipeline.add(Processor(worker2_task))
pipeline.add(Processor(cleanup_task))

with pipeline as pl:
    pl.run(['abc'])
```

Yields this output: `array('b', [44, 11, 22, 33])`

CHAPTER 5

Other links

- Source code
- PyPI project page