MapScale Documentation

Release 0.1.0

Jonathan Sick

June 13, 2012

CONTENTS

1 Contents **3**

1.1 What can MapScale do for me? 3

1.2 Getting Started 3

2 Indices and tables **5**

MapScale is a Python tool for embarrassingly parallel python workflows. It works like Python's built-in multiprocessing *Pool.map()*, except MapScale uses ZeroMQ messaging to create work servers. This lets you reduce overhead–workers are setup just once, and can be re-used with several map calls–and allows you to add workers from across the Internet.

MapScale is being developed at https://www.github.com/jonathansick/mapscale. To install:

```
git clone https://github.com/jonathansick/mapscale.git
cd mapscale
python setup.py install
```

Note, you will need to also install pyzmq.

CHAPTER

ONE

CONTENTS

1.1 What can MapScale do for me?

We all know the drill. Our pipelines are embarrassingly parallel; our datasets are massive, and our computers are stocked with cores. So we reach for Python's `multiprocessing` module:

```
import multiprocessing
pool = multiprocessing.Pool(processes=multiprocessing.cpu_count())
results = pool.map(my_pipeline_function, job_list)
```

This works great, but there are two limitations:

1. **Pickling overhead.** Your pipeline function needs to be pickled and unpickled for every work process. For one-off *map* queues this is acceptable, but some loosely coupled parallel algorithms require several bursts of map jobs based on the previous job pool. A good example is the emcee ensemble sampling code: a large pool of walkers explore multiple points in parameter space in parallel; a new set of points is chosen, and the posterior is again sampled in parallel. In this case, worker functions are pickled and pickled for each map iteration. Can't we just pickle the mapping function once on setup, and treat the workers as servers? MapScale does this thanks to the ZeroMQ messaging framework.
2. **Scalability.** Multiprocessing works fine within a single box; but what if we want to tap into the processing power of other computers on our network? For this you need a multiprocessing framework that understands TCP. Thanks for ZeroMQ, MapScale talks over TCP. Support for communicating with workers over SSH through SSH tunnels is planned.

1.2 Getting Started

1.2.1 Writing Work Functions

Like *multiprocessing*, your work functions take a single argument (the job) and return a single result. In MapScale, we take the idea further, by requiring that work functions be *callable class instances*. This allows us to have flexible initializations to package data required by all workers, have setup and destruction methods to prepare each worker instance *in situ*.

MapScale work classes follow a common interface:

```
class WorkFunction(object):
    """A function that we want to compute in parallel. Data can be packaged
    into the workers during '__init__', that will be available to all
    worker instances.
    """
```

```
    def __init__(self, args):
        super(WorkFunction, self).__init__()
        self.args = args

    def __call__(self, x):
        """Your call method takes a single argument with the job
        data. This can be any python object (numpy arrays, lists, etc.)
        """
        return x ** 2.

    def setup(self):
        """The setup method will be run once for each worker instance.
        This is your chance to setup a worker's environment (cache
        data from disk, etc.)
        """
        print "Setting up"

    def cleanup(self):
        """The cleanup method is called once for each worker instance
        when the user tells the MapScale processor to shutdown. This is
        your chance to release any resources used by the worker, such
        as temporary files.
        """
        print "Cleaning up"
```

1.2.2 Running MapScale with your Worker Function

Once you have a work function, you can setup a MapScale processor, as so:

```
from mapscale import Processor
myfunc = WorkFunction()
mapper = Processor(myfunc, 2)
```

Here the *mapper* is a `Processor` instance that bakes the work function in. The second argument is the number of workers you want to create on your *localhost*; two in this case.

The *mapper* is equivalent to Python's built-in `map()` function, *except* that you only need to pass an iterable list of jobs for the worker function to process each job–the worker function is already baked in! That is:

```
jobList = range(5)
results = mapper(jobList)
```

Here the job queue consists of a list of five integers (*jobList*). The *results* are now a list of five results, in the same order as your *jobList* (just like running a Python `map()`). And that's it!

But there's one last thing. Your worker functions are still alive. Think of workers as server processes that return a result whenever called. This design is beneficial because we can run *mapper(jobList)* a second (or Nth) time without having to setup the worker pool again (which is why we're using MapScale in the first place), but it means we need to manually shutdown the worker servers. To do this, we shut them down:

```
mapper.shutdown()
```

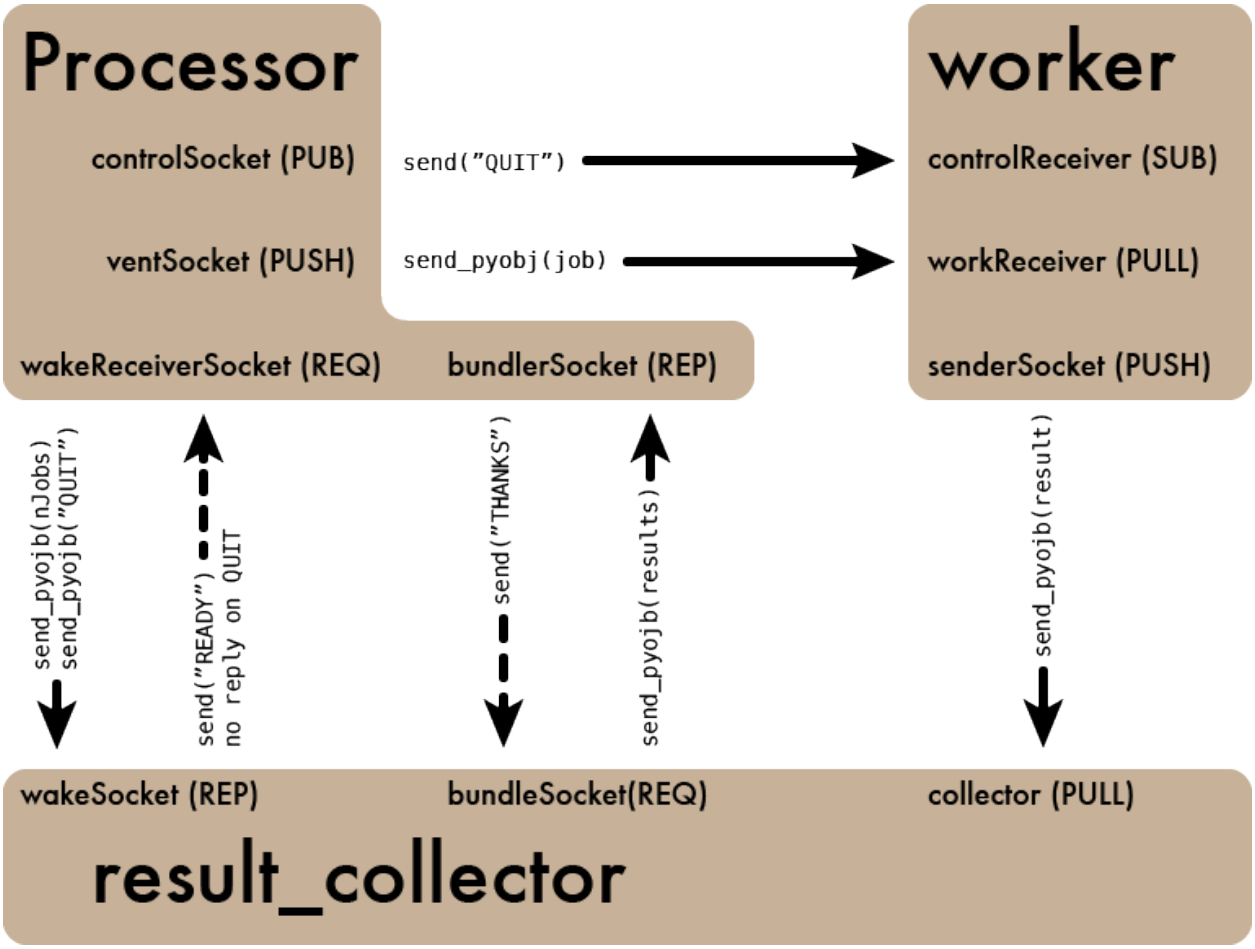
Once this is done, the work servers are shutdown, their *cleanup()* methods called, and *mapper* itself is rendered inert.

1.3 MapScale's Architecture

MapScale uses ZeroMQ (via pyzmq) to talk to workers and collect results. The architecture is based on the ventilator pattern, with additional patterns being employed to administer the workers and receiver. In this section, we'll document each communication channel that MapScale uses to aid developers. End users likely don't need to care about this. Also note that this architecture is likely to change as I (Jonathan Sick) learn more of ZeroMQ's best practices. This page will be updated as the architecture evolves.

1.3.1 Overview

Users interact directly with an instance of the *Processor* class, but there are two other types of components that operate in individual process: the *result_collector* and several *worker* instances. Interactions between these components, including socket and message types are outlined in this diagram:



1.3.2 The MapScale Life Cycle

Its easier to understand the network of sockets by outlining the sequential life cycle of a MapScale *Processor* instance.

Start-Up Phase

When a *Processor* instance is created, the *result_collector* is started in another process. The *result_collector* acts as a sink in our ventilator pattern, collecting results from workers. A number of local workers can also be booted up. When a worker starts up, the *startup()* method of your worker class is automatically run.

Note: We do not *yet* have any communication for when all the worker's *startup's are complete. You may need to include a 'sleep()* call if the startup() is lengthy.

Mapping Phase

When a sequence of jobs is submitted to your *Processor* instance, the first task is to tell the *result_collector* how many results to expect. This is done with Processor's *wakeReceiverSocket*, sending a python integer. This is a request-reply transaction, and Processor blocks until the *result_collector* acknowledges receipt.

Next, the jobs are sent to workers over the Processor's *ventSocket*. By using the ventilator pattern, jobs are automatically load balanced across workers. Once jobs are sent, the *Processor* instance blocks with a *bundlerSocket.recv_pyobj()* call, waiting for results to be made available from the *result_collector*.

Turning our attention to the workers, we see that workers nominally run infinite loops–they are persistent servers until we shut them off. In the infinite runtime loop, the worker checks for messages using a *zmq.Poller*. This poller effectively multiplexes several sockets together. If a message is received over the *workReceiver* socket, the job message is receiver thought the socket's *recv_pyobj* method. The job is passed as the sole argument to the user's work function The work function's result is then sent via the *senderSocket*'s *send_pyobj* method to the *result_collector*. As the worker continues to poll, it will process additional jobs.

Following the results to the *result_collector*, we see that it too has an infinite runtime loop that continuously polls sockets. One socket that it checks for is the *wakeSocket*, where the *Processor* instance originally told the *result_collector* how many jobs to expect. In this case, the *result_collector* blocks until that number of messages is received over the *collector* socket from the workers. The results are bundled into a list, and sent over the *bundlerSocket* back to the *Processor* instance, which has been blocking in anticipation of the results.

Before the result list is returned to the user, they must be sorted into the same order as the jobs were originally presented. This is easy since each job was enumerated as it was pushed to workers, and that enumeration tag is carried with the result back to the *Processor*.

The Shutdown Phase

Once jobs have been mapped once, or perhaps several times, the Processor is not longer needed. We need to tell the server processes (the *result_collector* and all the workers) to shutdown. This is achieved when the user calls the *shutdown* method. Here, two signals are send. First, over the *controlSocket*, a *"QUIT* message is broadcast to all workers. When workers encounter this message in their polling they run the *cleanup* method in the user's work function and exit their runtime loop. Second, a *"QUIT"* message is send to the *work_receiver*, which also terminates its runtime loop when that message is encountered. Note that to aid shutdown, *setsockopt(zmq.LINGER, 0)* is called on each socket in the *Processor*, telling those sockets that they can exit immediately without waiting for replies.

1.3.3 Python ZeroMQ Reading List

The ZeroMQ messaging framework takes a bit of time to wrap your head around, but luckily a good deal of lucid documentation has been written (particularly around the Python ZMQ package). Here is a list of pages that could be consulted before working on the MapScale code base:

- The ZeroMQ Guide is hilariously written and spells out the various sockets and design patterns used by ZeroMQ. MapScale takes advantage of the ventilator pattern, the pubsub pattern and the request-reply pattern, in particular.
- Brian Knox (Tao te Tek) wrote an excellent article on ZeroMQ as a Python multiprocessing alternative, which inspired this project. He also compared the performance of ZeroMQ and Queue in a separate post.
- Nicholas Piel wrote a broad introduction to ZeroMQ and PyZMQ.
- In this three-part series, Stefan Scherfke goes in-depth with Python ZeroMQ best-practices:
  1. Designing a PyZMQ Application
  2. Unit Testing PyZMQ Applications
  3. Process and System Testing
- Brian Knox also wrote about a multiprocessing design with batch acknowledgements, which may keep queue sizes smaller and improve stability.
- In Responsible workers with ZeroMQ, Samuel Tardin proposes a broker for works that could improve load balancing. Its an interesting design that we might consider for MapScale; particularly when workers-across-networks is introduced.

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*