knit Documentation

Release 0.2.4

Continuum Analytics

Jun 12, 2018

Contents

Knit launches YARN applications from Python.

Knit provides Python methods to quickly launch, monitor, and destroy distributed programs running on a YARN cluster, such as is found in traditional Hadoop environments.

Knit was originally designed to deploy Dask applications on YARN, but can deploy more general, non-Dask, applications as well.

Using conda, Knit can also deploy fully-featured Python environments within YARN containers, sending along useful libraries like NumPy, Pandas, and Scikit-Learn to all of the containers in the YARN cluster.

CHAPTER 1

Install

Use `pip` or `conda` to install:

```
conda install knit -c conda-forge
# or
pip install knit
```

CHAPTER 2

Launch Generic Applications

Instantiate `knit` with valid ResourceManager/Namenode IP/Ports and create a command string to run in all YARN containers

```
from knit import Knit
k = Knit(autodetect=True) # autodetect IP/Ports for YARN/HADOOP
```

Create a software environment with necessary packages

```
env = k.create_env('my-environment',
                   packages=['python=3.5', 'scikit-learn','pandas'],
                   channels=['conda-forge'])  # specify anaconda.org channels
```

Run a command within that environment on multiple containers

```
cmd = 'python -c "import sys; print(sys.version_info);"'
app_id = k.start(cmd, num_containers=2, env=env)  # start application
```

The `start` method also takes parameters to define the resource requirements of the application like `num_containers=`, `memory=`, `virtual_cores=`, `env=`, and `files=`.

CHAPTER 3

Launch Dask

Knit makes it easy to launch Dask on Yarn:

```
from knit.dask_yarn import DaskYARNCluster
env = k.create_env('my-environment',
                   packages=['python=3.6', 'scikit-learn', 'pandas', 'dask'],
                   channels=['conda-forge'])  # specify anaconda.org channels

cluster = DaskYARNCluster(env=env)
cluster.start(nworkers=10, memory=4096, cpus=2)

from dask.distributed import Client
client = Client(cluster)  # Connect local Dask client to cluster
```

If you want to connect to the remote Dask cluster from your local computer as is done in the last line then your local and remote environments should be similar.

CHAPTER 4

Packaged Environments

Yarn clusters typically lack strong Python environments with common libraries like NumPy, Pandas, and Scikit Learn. To resolve this, Knit creates redeployable conda environments that can be shipped along with your Yarn job, effectively bringing a fully-featured Python software environment to your Yarn cluster.

To achieve this Knit uses redeployable conda environments. Every time you create a new environment Knit will use conda locally to manage and download dependencies, and then will wrap those packages into a self-contained zip file that can be shipped to Yarn applications.

CHAPTER 5

Scope

Knit is not a full featured YARN solution. Knit focuses on the common case in computational workloads of starting a distributed process on many workers for a relatively short period of time. It does not provide fine-grained access to all Yarn functionality.

CHAPTER 6

Related Work

- Apache Slider: General purpose YARN application with a focus on long-running applications/services: HBase, Accumulo, etc.
- kitten: General purpose YARN application with Lua based configuration

See *the quickstart* to get started.

6.1 Installation

The runtime requirements of `knit` are python, lxml, requests, py4j. Python versions 2.7, 3.5 and 3.6 are currently supported. Dask is required to launch a Dask cluster. These are all available via conda (py4j on the conda-forge channel).

Testing depends on `pytest`.

6.1.1 Easy

Use `pip` or `conda` to install:

```
$ conda install knit -c conda-forge
or
$ pip install knit --upgrade
```

For dask clusters, you also need dask itself:

```
$ conda install dask distributed
```

6.1.2 Source

The following steps can be used to install and run `knit` from source.

Update and install system dependencies (e.g., for debian systems):

```
$ sudo apt-get update
$ sudo apt-get install git maven openjdk-7-jdk -y
```

or install these via conda

```
$ conda install -y -c conda-forge setuptools maven openjdk
```

Clone git repository and build maven project:

```
$ git clone https://github.com/dask/knit
$ cd knit
$ python setup.py install mvn
```

6.1.3 Testing on Docker

If you would like to test this package, but don't have a YARN cluster hanging around, you could make a small test one in your machine. This is essentially how the Continuous Integration tests work.

> $ export CONTAINER_ID=‘docker run -d mdurant/hadoop‘ $ docker exec -it $CONTAINER_ID bash # conda install dask distributed -y # conda install -c conda-forge lxml py4j knit # py.test -vv knit

6.2 Quickstart

6.2.1 Install

Use `pip` or `conda` to install:

```
$ pip install knit --upgrade
$ conda install knit -c conda-forge
```

6.2.2 Commands

Start

Instantiate `knit` with valid ResourceManager/Namenode IP/Ports and create a command string to run in all YARN containers

```
>>> from knit import Knit
>>> k = Knit(autodetect=True) # autodetect IP/Ports for YARN/HADOOP
>>> cmd = 'date'
>>> k.start(cmd)
'application_1454900586318_0004'
```

`start` also takes parameters: `num_containers`, `memory`, `virtual_cores`, `env`, and `files`

Status

After starting/submitting a command you can monitor its progress. The `status` method communicates with YARN's ResourceManager and returns a python dictionary with current monitoring data.

```
 >>> k.status()
{'allocatedMB': 512,
'allocatedVCores': 1,
'amContainerLogs': 'http://192.168.1.3:8042/node/containerlogs/container_
↪1454100653858_0011_01_000001/ubuntu',
'amHostHttpAddress': '192.168.1.3:8042',
'applicationTags': '',
'applicationType': 'YARN',
'clusterId': 1454100653858,
'diagnostics': '',
'elapsedTime': 123800,
'finalStatus': 'UNDEFINED',
'finishedTime': 0,
'id': 'application_1454100653858_0011',
'memorySeconds': 63247,
'name': 'knit',
'numAMContainerPreempted': 0,
'numNonAMContainerPreempted': 0,
'preemptedResourceMB': 0,
'preemptedResourceVCores': 0,
'progress': 0.0,
'queue': 'default',
'runningContainers': 1,
'startedTime': 1454276990907,
'state': 'ACCEPTED',
'trackingUI': 'UNASSIGNED',
'user': 'ubuntu',
'vcoreSeconds': 123}
```

Often we track the `state` of an application. Possible `states` include: `NEW`, `NEW_SAVING`, `SUBMITTED`, `ACCEPTED`, `RUNNING`, `FINISHED`, `FAILED`, `KILLED`

Further details on the current functioning of the cluster are available via the connected `yarn_api` class which can help with trouble shooting: `cluster_metrics()`, `nodes()`, `systems_logs`.

Logs

We retrieve log data directly from a `RUNNING` Application Master:

```
>>> k.logs()
```

Or, if log aggregation is enabled, we retrieve the resulting aggregated log data stored in HDFS. Note: aggregated log data is only available **after** the application has finished or been terminated, usually with a small lag of a few seconds while log aggregation takes place.

Kill

To stop an application from executing immediately, use the `kill` method:

```
>>> k.kill()
```

6.2.3 Python Applications

Python applications can be created by first making a conda environment for them to run within. This can be done directly with `CondaCreator` (and such environments are cached and reused) or with the `knit` instance itself.

A simple Python based application:

```
from knit import Knit
k = Knit()

env = k.create_env('test', packages=['python=3.5']])
cmd = 'python -c "import sys; print(sys.version_info); import random;␣
↪print(str(random.random()))"'
app_id = k.start(cmd, num_containers=2, env=env)
```

A long running Python application. Here we reuse the same environment create above:

```
from knit import Knit
k = Knit()

cmd = 'python -m SimpleHTTPServer'
app_id = k.start(cmd, num_containers=2, env=env)
```

6.2.4 Dask Cluster

Run a distributed dask cluster on YARN with a few lines like:

To start a dask cluster on YARN

```
import dask_yarn

# Specify conda packages and channels for execution environment
cluster = dask_yarn.DaskYARNCluster(packages=['python=3.6', 'scikit-learn', 'pandas',
↪'dask'],
                                    channels=['conda-forge'])

# each worker gets 4GB and two cores
cluster.start(nworkers=10, memory=4096, cpus=2)

from dask.distributed import Client
client = Client(cluster)
```

6.3 Usage

`Knit` can be used in several novel ways. Our primary concern is supporting easy deployment of distributed Python runtimes; though, we can also consider other languages (R, Julia, etc) should interest develop. Below are a few novels ways we can currently use `Knit`

6.3.1 Python

The example below use any Python found in the `$PATH`. This is usually the system Python (i.e., on a cluster where it has already been installed for you).

```
>>> import knit
>>> k = knit.Knit()
>>> cmd = "python -c 'import sys; print(sys.path); import socket; print(socket.
↪gethostname())'"
>>> appId = k.start(cmd)
```

6.3.2 Zipped Conda Envs

Often nodes managed under YARN may not have desired Python libraries or the Python binary at all! In these cases, we want to package up an environment to be shipped along with the command. `knit` allows us to declare a zipped directory with the following structure typical of Python environments:

```
$ ll dev/
drwxr-xr-x+ 23 ubuntu  ubuntu   782B Jan 30 17:55 bin
drwxr-xr-x+ 20 ubuntu  ubuntu   680B Jan 30 17:55 include
drwxr-xr-x+ 39 ubuntu  staff    1.3K Jan 30 17:55 lib
drwxr-xr-x+  4 ubuntu  staff    136B Jan 30 17:55 share
drwxr-xr-x+  6 ubuntu  ubuntu   204B Jan 30 17:55 ssl
```

```
>>> appId = k.start(cmd, env='<full-path>/dev.zip')
```

When we ship `<full-path>/dev.zip`, knit uploads `dev.zip` to a temporary directory within the user's home HDFS space e.g. `/users/ubuntu/.knitDeps` and the following bash ENVIRONMENT variables will be available:

- `$CONDA_PREFIX`: full path to prefix location of zipped directory
- `$PYTHON_BIN`: full path to Python binary

With the ENVIRONMENT variables available users can build more nuanced commands like the following:

```
>>> cmd = '$PYTHON_BIN $CONDA_PREFIX/bin/dask-worker 8786'
```

`knit` also provides a convenience method with `conda` to help build zipped environments. The following builds an environment `env.zip` with Python 3.5 and a variety of popular data Python libraries:

```
>>> env_zip = k.create_env(env_name='dev', packages=['python=3', 'distributed',
...                                                  'dask', 'pandas', 'scikit-learn
↪'])
```

6.3.3 Adding Files

`Knit` can also pass local files to each container.

```
>>> files = ['creds.txt', 'data.csv']
>>> k.start(cmd, files=files)
```

With the above, we are send files `creds.txt` and `data.csv` to each container and can reference them as local file paths in the `cmd` command.

6.3.4 Dask Clusters

The previous methods can be combined to launch a full distributed dask cluster on YARN with code like the following

```
from dask_yarn import DaskYARNCluster
cluster = DaskYARNCluster(env='my/conda/env.zip')
cluster.start(8, cpu=2, memory=2048)
```

The object `cluster` starts a dask scheduler, and can also be used to start or stop more containers than the original 8 referenced above. The same set of config options apply as for a `Knit` object, in addition to conda creation options, which will define the environment in which the workers run.

To start a dask client in the same session, you can simply do

```
from dask.distributed import Client
c = Client(cluster)
```

and use as usual, or look at `cluster.scheduler_address` for clients connecting from other sessions.

Note that DaskYARNCluster can also be used as a context manager, which will ensure that it gets closed (and the corresponding YARN application killed) when the `with` context finishes.

6.3.5 Instance Connections

The main instances that you will handle in this library have attributes which are instances of other classes, and expose functionality. Generally, parameters are passed down, so that the constructor parameters for DaskYarnCluster will also be used for Knit (e.g., `replication_factor`), CondaCreator (e.g., `channels`) and YARNAPI (e.g., `rm`).

DaskYarnCluster:

- `.knit` is an instance of Knit, and exposes methods to check the yarn application

state, logs and to increase/decrease the container count - an instance of `CondaCreator` is created on-the-fly if making/zipping a conda environment - `.local_cluster` is an instance of `dask.distributed.LocalCluster`, with no local workers. The only parameter passed in is `ip`.

Knit:

- `.yarn_api` is an instance of YARNAPI, which provides commands to be directly

executed by the ResourceManager, including several informational calls, mostly via REST - an instance of `CondaCreator` is created on-the-fly if making/zipping a conda environment

6.4 Troubleshooting

6.4.1 Outline

YARN is a complex system. This library contains core python classes for getting YARN information, creating and managing an Application, and building a Dask cluster on top of it. Typical usage involves building a .zip-file containing a full python environment, pushing this to HDFS, getting Yarn to start an Application, which in turn starts containers that use the environment to bootstrap python processes.

Aside from python code, there is also code in Scala for a Yarn Client (which runs locally) and, separately, a Yarn ApplicationMaster (which runs in a special container allocated by Yarn). Communication between python and the two Scala/JVM processes is via sockets managed by py4j and buffered in a background thread on the python side.

6.4.2 Sources of information

Should an application fail or hang, here are a number of places to look first for thr source of the problem.

console feedback

Both the python and the Scala code are fairly verbose to inform the user of the stage currently occurring. On the python side, you can use standard logging to set the logging level to DEBUG and get more information. Most exceptions result in a message giving some possible remedies for the situation.

Application/Container logs

If Yarn started any containers, they will emit logs. The first of these will be the ApplicationMaster, where you will see debug logging while the application is setting itself up. Logs also are created by each of the python processes being run in the worker containers. Any exception in the python processes should be visible in the logs.

Whilst the application is running, you can display logs for containers as with `k.print_logs()`, where `k` is the Knit instance, or `cluster.knit.print_logs()` where `cluster` is the DaskYarnCluster instance.

Logs are available for containers sol ong as they are alive. After an application finishes (including if it is killed), the logs will be stored to HDFS and be accessible to you with the same commands *if log aggregation is enabled on the cluster*. Typically, aggregation isn't immediate, so there may be a lag after application end while the logs are not available. If log aggregation is off, the logs are lost after a container ends, although they may still be available locally on the machine that hosted the container.

Cluster Information

A YARNAPI instance (usually the `.yarn_api` attribute of a Knit instance) gives access to various information about the connected Yarn cluster and pplications registered on it

Some methods of interest:

apps Names of all apps known to Yarn

apps_info, app_attempts, app_containers Detailed information about the current status of a given app

cluster_info, cluster_metrics Global cluster information, including whether the Resource Manager is up and happy, and the global resource (memory, cpu) constraints it has to work with

nodes Information about the connected Node Managers, whether they are healthy and the resources each has available. If unhealthy, there should be a message stating why.

RM/NM Logs

The Resource Manager and Node Managers keep logs of their activity in a local directory on their respective machines. If, for some reason, an application is rejected at time of submission, or an application is killed without any exception in the application logs, the reason may well be given in here.

Because these logs are only written to local discs, direct access to the machine is neede to read them - they may well not be available to you.

If run on the same machine as the Resource Manager and/or Node Manager (such as the special case of a single-node pseudo-cluster, useful for testing), the YARNAPI method `system_logs` will attempt to find the location of logs, which you can view with usual system tools such as `tail`.

6.4.3 Specific issues

Here follow some specific cases that have caused problems in the past, with guidance of what might be done.

Don't make a .zip of the conda root

CondaCreator can zip up any directory, and makes copies of files referenced by symbolic links. Normally you would use this with a conda environment directory, usually in the `/envs/` directory of a conda installation, or created by Knit in a local directory especially for this purpose. If you attempt to use the root environment (i.e., a directory *containing* `/envs/`), zip will fail with link recursion, probably after attempting make an enormous file.

Newer java version if .zip > 2GB

It is easy to make a conda environment zip with size > 2GB, if including many libraries. At that threshold, the Zip64 extension is invoked, although file-sizes up to 4GB are supposed to be possible without it.

If the java version running Yarn is old enough, it will not be able to handle these larger .zip files. `java.util.zip.ZipException: invalid CEN header (bad signature)` will be printed in the ResourceManager logs. Either reduce the size of the conda environment, or update java on the cluster.

Dask client and workers versions

Dask requires the versions of dask and other auxiliary libraries to match between the clients and workers, so that functions and data can be deserialised. If you create an environment using `packages=`, then this will pull the latest versions from the repo, unless you specify exact versions. Also note that the channels passed should match your system settings, as some packages are not compatible between defaults and conda-forge or other channels.

Once a dask cluster is running, you can see the versions on the workers by starting a client. In the same session as the DaskYarnCluster you could do:

REST routing to YARN

Although application submission and launching are handled via RPCs in scala, several informational calls are made using the Yarn REST end-points. These must be reachable.

If you see HTTP connection errors, then there is a possibility that the end-points are protected by a proxy/gateway such as Knox. You will need to find the appropriate host, port and path to supply to YARNAPI, such as:

The example would set the API end-point to `'proxy.server.org:9999/default/resourcemanager/ws/v1/'`, and whether access is HTTP or HTTPS would depend on the value of `'yarn.http.policy'`.

There is no way for Knit to be able to automatically determine the right URL to contact, the information must come from systems operations.

REST auth

The REST end-points may require Kerberos authentication, which will generally depend on the value of configuration parameter `hadoop.http.authentication.type`. The extra package request-kerberos is required, but otherwise the connection should be seamless, so long as a valid ticket exists.

Alternatively, in the case that the authentication is simple, but anonymous access is disallowed, you must provide a password upon instantiation and perhaps a user-name different from the apparent user who own the session.

IP of scheduler

Upon startup, the Dask cluster scheduler guesses its own IP as `socket.gethostbyname(socket.gethostname())` - this is the value that workers will be passed.

On some networks, it is possible that the IP that workers need in order to be able to contact the scheduler is different from the value that would be guessed. The parameter `ip=` can be passed to set the correct value.

Console language of workers for click

The dask worker is executed as a console application, with the library `click` being used to parse command-line options. `click` needs to interpret the character encoding of the command line it receives, with the result that if the language is not specified, you will see `Exit 1` and an informational statement about setting the language in the worker logs. A `lang=` is provided to set the effective language setting that the worker processes see. However, there are further constraints on what languages are permitted, set by the host system - an unwise choice may cause errors like "LC.ALL=xxx: not a valid identifier", and no python process at all.

System constraints

Yarn has a large number of system parameters that it matches, and constraints that must be simultaneously met in order to launch an application. Failures due to obvious unmet conditions (e.g., asking for more memory than the total available to the cluster) will probably be flagged before attempting to launch a cluster, if `checks=True` in `.start()`.

However, there are more subtle fail cases. For example, the minimum memory allotment to a container is rarely less than 1GB, often more, so an application may take much more than the request passed to Knit suggests.

An even more subtle example: Yarn Node Managers watch disc usage, and if the used fraction goes above a predetermined threshold (default: 90%), the disc will be labelled "bad", logging will be prevented, and the entire node will refuse to take jobs until the situation is rectified.

Build the .jar if running from source

If installing from the repo source, the .jar file needs to be created before/during installation:

```
python setup.py install mvn
```

which requires `maven` to be available, as well as java.

Configuration

Knit does its best to find configuration files, but it is always best to check the contents of the `.conf` attribute of a Knit instance (a dictionary) to make sure that inference was successful, and provide any overrides that might be necessary.

6.5 API

`CondaCreator`([conda_root, conda_envs, ...])	Create Conda Env
`CondaCreator.create_env`(env_name[, ...])	Create zipped directory of a conda environment
`CondaCreator.zip_env`	

`YARNAPI`(rm, rm_port[, scheme, gateway_path, ...])	REST interface to YARN
`YARNAPI.apps`	App IDs known to YARN
`YARNAPI.app_containers`([app_id, info])	Get list of container information for given app.

Continued on next page

Table 2 – continued from previous page

YARNAPI.logs(app_id[, shell, retries, delay])	Collect logs from RM (if running) With shell=True, collect logs from HDFS after job completion
YARNAPI.container_status(container_id)	Ask the YARN shell about the given container
YARNAPI.status(app_id)	Get status of an application
YARNAPI.kill_all([knit_only])	Kill a set of applications
YARNAPI.kill(app_id)	Method to kill a yarn application

Knit([autodetect, upload_always, hdfs_home, ...])	Connection to HDFS/YARN.
Knit.start(cmd[, num_containers, ...])	Method to start a yarn app with a distributed shell
Knit.logs([shell])	Collect logs from RM (if running) With shell=True, collect logs from HDFS after job completion
Knit.status()	Get status of an application
Knit.kill()	Method to kill a yarn application
Knit.create_env(env_name[, packages, ...])	Create zipped directory of a conda environment

class knit.core.**Knit**(*autodetect=True*, *upload_always=False*, *hdfs_home=None*, *knit_home='/home/docs/checkouts/readthedocs.org/user_builds/knit/checkouts/stable/knit/java_libs'*, *hdfs=None*, *pars=None*, ***kwargs*)

Connection to HDFS/YARN. Launches a single "application" master with a number of worker containers.

Parameter definition (nn, nn_port, rm, rm_port): those parameters given to __init__ take priority. If autodetect=True, Knit will attempt to fill out the others from system configuration files; fallback values are provided if this fails.

Parameters

nn: str Namenode hostname/ip

nn_port: int Namenode Port (default: 9000)

rm: str Resource Manager hostname

rm_port: int Resource Manager port (default: 8088)

lang: str Environment variable language setting, required for `click` to successfully read from the shell. (default: 'C.UTF-8')

user: str ('root') The user name from point of view of HDFS. This is only used when checking for the existence of knit files on HDFS, since they are stored in the user's home directory.

hdfs_home: str Explicit location of a writable directory in HDFS to store files. Defaults to the user 'home': hdfs://user/<username>/

replication_factor: int (3) replication factor for files upload to HDFS (default: 3)

autodetect: bool Autodetect configuration

upload_always: bool(=False) If True, will upload conda environment zip always; otherwise will attempt to check for the file's existence in HDFS (using the hdfs3 library, if present) and not upload if that matches the existing local file in size and is newer.

knit_home: str Location of knit's jar

hdfs: HDFileSystem instance or None Used for checking files in HDFS.

Note: for now, only one Knit instance can live in a single process because

of how py4j interfaces with the JVM.

Examples

```
>>> k = Knit()
>>> app_id = k.start('sleep 100', num_containers=5, memory=1024)
```

add_containers(*num_containers=1*, *virtual_cores=1*, *memory=128*)
Method to add containers to an already running yarn app

num_containers: int Number of containers YARN should request (default: 1) * A container should be requested with the number of cores it can

saturate, i.e.

- the average number of threads it expects to have runnable at a time.

virtual_cores: int Number of virtual cores per container (default: 1) * A node's capacity should be configured with virtual cores equal to * its number of physical cores.

memory: int Memory per container (default: 128) * The unit for memory is megabytes.

check_needs_upload(*path*)
Upload is needed if file does not exist in HDFS or is older

static create_env(*env_name*, *packages=None*, *remove=False*, *channels=None*, *conda_pars=None*)
Create zipped directory of a conda environment

Parameters

env_name [str]

packages [list]

conda_root: str Location of conda installation. If None, will download miniconda and produce an isolated environment.

remove [bool] remove possible conda environment before creating

channels [list of str] conda channels to use (defaults to your conda setup)

conda_pars: dict Further pars to pass to CondaCreator

Returns

path: str path to zipped conda environment

Examples

```
>>> k = Knit()
>>> pkg_path = k.create_env(env_name='dev',
...                         packages=['distributed', 'dask', 'pandas'])
```

get_container_statuses()
Get status info for each container

Returns dict where the values are the raw text output.

get_containers()
Method to return active containers

Returns

container_list: List List of dicts with each container's details

kill()
Method to kill a yarn application

Returns

bool: True if successful, False otherwise.

list_envs()
List knit conda environments already in HDFS

Looks in staging directory for zip-files

Returns: list of dict Details for each zip-file.

logs(*shell=False*)
Collect logs from RM (if running) With shell=True, collect logs from HDFS after job completion

Parameters

shell: bool Shell out to yarn CLI (default False)

Returns

log: dictionary logs from each container (when possible)

print_logs(*shell=False*)
print out a more console-friendly version of logs()

remove_containers(*container_id*)
Method to remove containers from a running yarn app

Calls removeContainers in ApplicationMaster.scala

Be careful removing the ...0001 container. This is where the applicationMaster is running

Parameters

container_id: str

Returns

None

runtime_status()
Get runtime status of an application

Returns

str: status of application

start(*cmd*, *num_containers=1*, *virtual_cores=1*, *memory=128*, *files=None*, *envvars=None*, *app_name='knit'*, *queue='default'*, *checks=True*)
Method to start a yarn app with a distributed shell

Parameters

cmd: str command to run in each yarn container

num_containers: int Number of containers YARN should request (default: 1) * A container should be requested with the number of cores it can

saturate, i.e.

- the average number of threads it expects to have runnable at a time.

virtual_cores: int Number of virtual cores per container (default: 1) * A node's capacity should be configured with virtual cores equal to * its number of physical cores.

memory: int Memory per container (default: 128) * The unit for memory is megabytes.

files: list list of files to be include in each container. If starting with *hdfs://*, assume these already exist in HDFS and don't need uploading. Otherwise, if hdfs3 is installed, existence of the file on HDFS will be checked to see if upload is needed. Files ending with *.zip* will be decompressed in the container before launch as a directory with the same name as the file: if myarc.zip contains files inside a directory stuff/, to the container they will appear at ./myarc.zip/stuff/* .

envvars: dict Environment variables to pass to AM *and* workers. Both keys and values must be strings only.

app_name: String Application name shown in YARN (default: "knit")

queue: String RM Queue to use while scheduling (default: "default")

checks: bool=True Whether to run pre-flight checks before submitting app to YARN

Returns

applicationId: str A yarn application ID string

status()
Get status of an application

Returns

log: dictionary status of application

wait_for_completion(*timeout=10*)
Wait for completion of the yarn application

Returns

bool: True if successful, False otherwise

DaskYARNCluster
DaskYARNCluster.start
DaskYARNCluster.stop
DaskYARNCluster.close
DaskYARNCluster.add_workers
DaskYARNCluster.remove_worker

6.6 Configuration

Several methods are available for configuring `Knit`.

The simplest is to load values from system `.xml` files. Knit will search typical locations and reads default configuration parameters from there. The file locations may also be specified with the environment variables `HADOOP_CONF_DIR`, which is the directory containing the XLM files, `HADOOP_INSTALL`, in which case the files are expected in subdirectory `hadoop/conf/`.

It is also possible to pass parameters when instantiating `Knit` or `DaskYARNCluster`. You can either provide individual common overrides (e.g., `rm='myhost'`) or provide a whole configuration as a dictionary (`pars={}`) with the same key names as typically contained in the XML config files. These parameters will take precedence over any loaded from files, or you can disable using the default configuration at all with `autodetect=False`.

6.6.1 Connection with hdfs3

Some operations, such as checking for uploaded conda environments, optionally make use of hdfs3. The configuration system, above, and that for hdfs3 are very similar, so you may well not have to make any extra steps to get this working correctly for you; normally the files defining values for Yarn should be in the same location as those for HDFS. However, you may well wish to be more explicit about the configuration of the HDFileSystem instance you want knit to use. In this case, create the instance as usual, and assign it to the Knit instance as follows

```
hdfs = HDFileSystem(...)
k = Knit(..., hdfs=hdfs)
```

or, similarly for a Dask cluster

```
cluster = DaskYARNCluster(..., hdfs=hdfs)
```

6.7 Examples

6.7.1 IPython Parallel

Install IPython Parallel and start IP Controller:

```
$ conda install ipyparallel
or
$ pip ipyparallel
$ ipcontroller --ip=*
```

IPController will create a file: `ipcontroller-engine.json` which contains metadata and security information needed by worker nodes to connect back to the controller. In a separate shell or terminal we use knit to ship a self-contained environment with `ipyparallel` (and other dependenices) and start `ipengine`

```
>>> from knit import Knit
>>> k = Knit(autodetect=True)
>>> env = k.create_env(env_name='ipyparallel', packages=['numpy', 'ipyparallel',
↪'python=3'])
>>> controller = '<HOMEDIR>/.ipython/profile_default/security/ipcontroller-engine.json
↪'
>>> cmd = '$PYTHON_BIN $CONDA_PREFIX/bin/ipengine --file=ipcontroller-engine.json'
>>> app_id = k.start(cmd, env=env, files=[controller], num_containers=3)
```

IPython Parallel is now running in 3 containers on our YARN managed cluster:

```
>>> from ipyparallel import Client
>>> c = Client()
>>> c.ids
[2, 3, 4]
```

Index