knit Documentation

Release 0.1.1+18.gaea8d39.dirty

Continuum Analytics

Dec 29, 2016

Contents

Knit enables data scientists to quickly launch, monitor, and destroy distributed programs on a YARN cluster.

Many YARN applications are simple distributed shell commands – running a shell command on several nodes in a cluster. Knit enables users to express and deploy applications and software environments managed under YARN through Python or directly on the command line.

CHAPTER 1

Motivation

Knit was built to support batch-oriented non-JVM applications. For example, Knit can deploy Python based distributed applications such as IPython Parallel and Dask+Distributed. Knit was built with the following motivations in mind:

- **PyData Support** Bring the PyData stack into the Hadoop/YARN ecosystem
- **Easy Setup:** Support a minimal installation effort and the common cases with easy to use CLI and Python interface.
- **Deployable Runtimes:** Build and ship self contained environments along with the application. Knit uses conda to resolve library dependencies and deploy user libraries without IT infrastructure and management

CHAPTER 2

Scope

Knit enables data scientists to quickly launch, monitor, and destroy simple distributed programs.

Knit is not a full featured YARN solution. Knit focuses on the common case in scientific workloads of starting a distributed process on many workers for a relatively short period of time. Knit does not handle dynamic container management nor is it suitable for running long-term infrastructural applications.

2.1 IPython Parallel Example

As an example we install and deploy the popular IPython Parallel project on a YARN cluster. This example can be performed by anyone with user privileges on a YARN cluster and a local Python installation. It does not depend on root privileges nor on Python being widely deployed throughout the cluster.

We install and start the IP Controller on the head node:

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

The IPController creates 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` (plus other dependencies) and start `ipengine`

```
>>> from knit import Knit
>>> k = Knit(autodetect=True)
>>> env = k.create_env(env_name='ipyparallel', packages=['numpy', 'ipyparallel',
↪'python=3'])
>>> controller = '/home/ubuntu/.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]
```

CHAPTER 3

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.

3.1 Installation

3.1.1 Easy

Use `pip` or `conda` to install:

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

3.1.2 Source

The following steps can be used to install and run `knit` from source. These instructions were tested on Ubuntu 14.04, CDH 5.5.1, and Hadoop 2.6.0.

Update and install system dependencies:

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

Clone git repository and build maven project:

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

3.2 Quickstart

3.2.1 Install

Use `pip` or `conda` to install:

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

3.2.2 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(app_id)
 {'app': {'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`

Logs

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

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

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 using the `shell=True` keyword argument:

```
>>> k.logs(app_id, shell=True)
```

Kill

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

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

3.2.3 Python Applications

A simple Python based application:

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

cmd = 'python -c "import sys; print(sys.version_info); import random;␣
↪print(str(random.random()))"'
app_id = k.start(cmd, num_containers=2)
```

A long running Python application:

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

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

3.3 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`

3.3.1 Python

The example below use Python found in the `$PATH`. This is usually the system Python.

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

3.3.2 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/dworker 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
↪'])
```

3.3.3 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.

3.3.4 JVM CLI

We can also call out to the HADOOP jar directly:

```
$ hadoop jar ./knit-1.0-SNAPSHOT.jar io.continuum.knit.Client --help
   knit x.1
   Usage: scopt [options]

     -n <value> | --numContainers <value>
           Number of YARN containers
     -m <value> | --memory <value>
           Amount of memory per container
     -c <value> | --virtualCores <value>
           Virtual cores per container
     -C <value> | --command <value>
           Command to run in containers
     -p <value> | --pythonEnv <value>
           Number of YARN containers
     --help
           command line for launching distributed python

$ hadoop jar ./knit-1.0-SNAPSHOT.jar io.continuum.knit.Client --numContainers 1 \
  --command "python -c 'import sys; print(sys.path); import random; print(str(random.
↪random()))'"
```

Helpful aliases

```
$ alias yarn-status='yarn application -status'
$ alias yarn-log='yarn logs -applicationId'
$ alias yarn-kill='yarn application -kill'
```

3.4 Examples

3.4.1 Dask + Distributed

Install *Dask* + *Distributed* and start the `dscheduler`:

```
$ conda install distributed -c dask
$ dscheduler
distributed.scheduler - INFO - Start Scheduler at:         172.31.54.49:8786
distributed.scheduler - INFO -            http at:         172.31.54.49:49616
```

The scheduler now awaits workers to connect to it. Knit runs the `dworker` command with a specific IP:Port to connect to.

```
>>> from knit import Knit
>>> k = Knit(autodetect=True)
>>> env = k.create_env(env_name='dev', packages=['dask', 'distributed', 'pandas'])
>>> cmd = '$PYTHON_BIN $CONDA_PREFIX/bin/dworker 172.31.54.49:8786'
```

The scheduler will report back the new connection from the workers:

```
distributed.core - INFO - Connection from 172.31.54.48:51508 to Scheduler
distributed.scheduler - INFO - Register ('172.31.54.48', 34959)
distributed.core - INFO - Connection from 172.31.54.45:46071 to Scheduler
distributed.scheduler - INFO - Register ('172.31.54.45', 43406)
distributed.core - INFO - Connection from 172.31.54.47:53764 to Scheduler
distributed.scheduler - INFO - Register ('172.31.54.47', 44928)
```

3.4.2 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 = '/home/ubuntu/.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]
```

3.5 API

Knit([nn, nn_port, rm, rm_port, autodetect, ...])	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)
Knit.status()	Get status of an application
Knit.kill([timeout])	Method to kill a yarn application
Knit.create_env(env_name[, packages, ...])	Create zipped directory of a conda environment

class knit.core.**Knit**(*nn='localhost'*, *nn_port=8020*, *rm='localhost'*, *rm_port=8088*, *autodetect=False*, *validate=True*)

Connection to HDFS/YARN

Parameters **nn: str**

Namenode hostname/ip

nn: str

Namenode hostname/ip

nn_port: str

Namenode Port (default: 9000)

rm: str

Resource Manager hostname

rm_port: str

Resource Manager port (default: 8088)

autodetect: bool

Autodetect NN/RM IP/Ports

validate: bool

Validate entered NN/RM IP/Ports match detected config file

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.

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

Parameters **env_name** : str

packages : list

conda_root : str, optional

remove : bool

remove possible conda environment before creating

Returns path: str

path to zipped conda environment

Examples

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

kill(*timeout=10*)
Method to kill a yarn application

Parameters **app_id: str**

YARN application id

timeout: int

Time in seconds to wait for completion before killing (default 10s)

Returns bool:

True if successful, False otherwise.

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

Parameters **app_id: str**

A yarn application ID string

shell: bool

Shell out to yarn CLI (default False)

Returns log: dictionary

logs from each container (when possible)

runtime_status()
Get runtime status of an application

Returns str:

status of application

start(*cmd*, *num_containers=1*, *virtual_cores=1*, *memory=128*, *env=''*, *files=[]*, *app_name='knit'*, *queue='default'*)
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.

env: string

Full Path to zipped Python environment

files: list

list of files to be include in each container

app_name: String

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

queue: String

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

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

Index