Cyverse Apps

Here I will try to document how I create an app to run on the Stampede2 cluster at TACC. This is by no means a definitive solution, it just seems to work for me.

To begin, you will need accounts with Cyverse and TACC:

- http://cyverse.org
- https://portal.tacc.utexas.edu

It's not necessary to have the same username, but it is convenient.

In order to test your application, you will need to be added to the "iPlant-Collabs" allocation (what account to charge for the time your jobs use on Stampede2). It's best to get on the Agave Slack channel (agaveapi.slack.com) and talk to someone like Matt Vaughn or John Fonner.

You will also need to install the Cyverse SDK (https://github.com/cyverse/cyverse-sdk) and learn how to get that to work. In short, you need to `tenants-init` and then `clients-create` to get going.

Directories

I typically have these directories in each Github repo for my apps:

- *scripts*: code I write that I want to call in my app
- *singularity*: files to build the Singularity container
- *stampede* files needed to create and kick off the app

In this repo, I also have included a "misc" directory with utility programs I've written which you may find useful.

Let's look at each in more detail.

scripts

These are the Python/Perl/R programs I write that I will need to have available inside the Singularity container. I tend to write "install.r" and "requirements.txt" files to make it easier to install all the dependencies for R and Python, respectively.

singularity

Usually there are just two files here:

- image.def: Singularity recipe for building the image

- Makefile: commands to build the image

The "image.def" file contains all the directives like the base OS, the packages to install, the custom software to build, etc. In a weirdly circular fashion, the Singularity image almost always clones the Github repo into the image so that the "scripts" directory exists with the programs I'll call. With the Makefile, you can just `make img`.

stampede(2)

Typically there are:

- app.json: JSON file to describe the app, inputs, and parameters
- template.sh: template file used to kick off the app
- test.sh: required but not used by Agave that I can tell
- run.sh: my usual way to run pipeline or just pass arguments
- Makefile: commonly used commands to save me typing
- MANIFEST: the list of the files I actually want uploaded

app.json

The "app.json" file can be very tricky to create. TACC has created an interface to assist in getting this right, and I made my own, too:

- https://togo.agaveapi.co/app/#/apps/new
- http://test.hurwitzlab.org (The Appetizer)

The Agave ToGo interface requires a user to login first, and so one advantage mine has (IMHO) is that it does not. The idea behind The Appetizer is to crowd-source the creation of new apps. If we can get users to help describe all the command-line options to some tool they wish to integrate, then it saves us that much time.

You can copy an existing "app.json" from another app and edit it by hand or paste it into the "Manual Edit" mode of the "JSON" tab in The Appetizer. "Inputs" are assets (files/data) provided by the user which must be copied to the compute node before the job begins. "Parameters" are options that are indicated, e.g., p-value or k-mer size, etc. Read the "Help" tab on The Appetizer for more information.

template.sh

This file will contain placeholders for each of the input/parameter `id`s that you define in the "app.json." These will be expanded at run time into literal values passed from Agave. E.g., if you defined an "INPUT" argument to have a "-i" prepended, then if the user provides "myinput.txt" as the `INPUT` argument,

`${INPUT}` will be turned into `-i myinput.txt`; if the user does not provide an argument (i.e., it's not required), then there will be nothing substituted into the `${INPUT}` (i.e., you don't have to worry that there will be "-i" with nothing following it).

NB: If you change anything in the "app.json," be sure to update "template.sh" so that the argument is represented in the "template.sh" file.

The template can pass the users arguments to any program you like. I tend to write a "run.sh" script that is my main entry point. I use bash because it has no dependencies and is simple but powerful enough for most apps. You could instead call a Python program that exists in your Singularity container, but I wouldn't attempt using Python directly on the compute node as you couldn't be sure you have the right version and any dependencies you might need, esp. if the app is made public and therefore runs under a different user.

test.sh

I don't know why this is required, but it is. I often will indicate some test data I have and will `sbatch` this file to ensure my pipeline works.

run.sh

As I said, you don't have to use bash as the main entry point, but it's often sufficient. I have many examples where I write entire pipelines in bash (fizkin) and others where I merely pass all the arguments to some more capable program (graftM). If the pipeline needs to use the "launcher" to get parallelization of jobs, I will probably stick with bash since the launcher is controlled entirely via environmental variables.

Makefile

To test and build an app, I will do the same steps repeatedly, so I tend to put them here so I can, e.g., `make clean` to get rid of previous runs, `make app` to submit the "app.json," `make up` to upload the assets into the execution system.

MANIFEST

The only files required to exist in the execution system are the "template" and "test" files you indicate in the "app.json." After that, you need to also include the Singularity image and any other programs that are referenced, e.g., "template.sh" might call "run.sh." I have a program called "copy_from_manifest.py" that looks for a file called "MANIFEST" and only uploads those files into the proper "applications" directory.

I also have a simple bash program called "upload-file.sh" that I use to upload a single file rather than everything in MANIFEST. Because of the time it takes to copy large files (see below), I usually only use "copy_from_manifest.py" the first time I upload my files. I found it necessary to write "upload-file.sh" because I found Agave would sometimes merge my new and old files into and unusable mishmash. This program first does a `files-delete` before uploading the new file.

Singularity

Back in the bad olde days, we'd install all the dependecies (R/Python modules, extra binaries, etc.) into our $HOME directory which would be mounted at runtime. This method fails, however, once the app is made public and runs under a different user and environment. The only sane way to package an app is to use a container that has all the programs and dependencies contained within. Docker basically works, but it requires root to run which is not going to happen on any shared HPC. Enter Singularity which basically extended Docker to fix this security hole. If you can make a Docker container, you can easily create one in Singularity.

Biocontainers

There are almost 6800 Biocontainer Singularity images that exist in "/work/projects/singularity/TACC/biocontainers" that you can use right away. For an example, see "trim-galore" (https://github.com/hurwitzlab/trim-galore).

Dockerfile

If you are trying to create a Singularity container from an existing Docker definition, you can use a definition like the one in "saffrontree" (https://github.com/hurwitzlab/saffrontree).

Scratch

If you are creating a container from scratch, you can follow the outline provided or look at more complicated installs like those for Fizkin (https://github.com/hurwitzlab/fizkin).

Image Size

You will want to keep the Singularity container size in the .5-2GB range. If you have large data files you need for your tool such as reference databases for BLAST or UProC, those should be place into the 40T disk space set aside for iMicrobe at "/work/05066/imicrobe/iplantc.org/data".

Making Singularity

Creating a Singularity container requires root access and cannot be done on a Mac; therefore, I tend to use our machine "lytic." After I `make img` in the "singularity" directory, I see something like this:

```
$ ls -lh
total 290M
-rw-r--r--. 1 kyclark staff  467 Apr 12 11:02 image.def
-rwxr-xr-x. 1 root    root  513M Apr 12 11:03 lc.img
-rw-r--r--. 1 kyclark staff  363 Apr 12 11:01 Makefile
```

I can verify that the image works as expected:

```
$ ./lc.img
Usage: lc.py FILE
$ singularity exec lc.img lc.py image.def
There are "27" lines in "image.def"
```

Getting Image to Stampede2

You can `scp` the image to Stampede2, but that will require MFA which always annoys me, so I tend to `iput` the file into the Data Store and then `iget` it on Stampede2 into the "stampede" directory (the deployment path so it will be uploaded):

```
[lytic@~/work/stampede2-template/singularity]$ iput lc.img
```

a [s2:login4@/work/03137/kyclark/stampede2/stampede2-template/stampede]$ iget lc.img h

To test that the image works on Stampede2, you must `idev` to get a compute node an interactive session as Singularity use is not allowed on head nodes:

```
$ idev


 -> Checking on the status of development queue. OK

 -> Defaults file    : ~/.idevrc
```

```
 -> System           : stampede2
 -> Queue            : development    (idev  default       )
 -> Nodes            : 1              (idev  default       )
 -> Tasks per Node   : 1              (~/.idevrc           )
 -> Time (minutes)   : 30             (idev  default       )
 -> Project          : iPlant-Collab  (~/.idevrc           )

------------------------------------------------------------------
          Welcome to the Stampede2 Supercomputer
------------------------------------------------------------------

No reservation for this job
--> Verifying valid submit host (login4)...OK
--> Verifying valid jobname...OK
--> Enforcing max jobs per user...OK
--> Verifying availability of your home dir (/home1/03137/kyclark)...OK
--> Verifying availability of your work dir (/work/03137/kyclark/stampede2)...OK
--> Verifying availability of your scratch dir (/scratch/03137/kyclark)...OK
--> Verifying valid ssh keys...OK
--> Verifying access to desired queue (development)...OK
--> Verifying job request is within current queue limits...OK
--> Checking available allocation (iPlant-Collabs)...OK
Submitted batch job 1099613

 -> After your idev job begins to run, a command prompt will appear,
 -> and you can begin your interactive development session.
 -> We will report the job status every 4 seconds: (PD=pending, R=running).

 -> job status:  PD
 -> job status:  R

 -> Job is now running on masternode= c455-062...OK
 -> Sleeping for 7 seconds...OK
 -> Checking to make sure your job has initialized an env for you....OK
 -> Creating interactive terminal session (login) on master node c455-062.

Last login: Fri Apr  6 12:00:43 2018 from login4.stampede2.tacc.utexas.edu
TACC Stampede2 System
Provisioned on 24-May-2017 at 11:47

$ module load tacc-singularity
$ ./lc.img app.json
Usage: lc.py FILE
$ singularity exec lc.img lc.py app.json
There are "59" lines in "app.json"
```

Testing HPC

Stampede apps will be submitted to SLURM to run, so the first step in testing is that you can successfully submit a job and run to completion. Here I'll `sbatch test.sh` and see how it goes. Notice this script uses the "development" queue with a max run time of 2 hours. This queue is specfically for debugging, and jobs get picked up almost immediately. Your actual job will most likely go into the "normal" queue which may take minutes to hours to pick up your job.

```
$ make test
sbatch test.sh

------------------------------------------------------------------
          Welcome to the Stampede2 Supercomputer
------------------------------------------------------------------

No reservation for this job
--> Verifying valid submit host (login4)...OK
--> Verifying valid jobname...OK
--> Enforcing max jobs per user...OK
--> Verifying availability of your home dir (/home1/03137/kyclark)...OK
--> Verifying availability of your work dir (/work/03137/kyclark/stampede2)...OK
--> Verifying availability of your scratch dir (/scratch/03137/kyclark)...OK
--> Verifying valid ssh keys...OK
--> Verifying access to desired queue (development)...OK
--> Verifying job request is within current queue limits...OK
--> Checking available allocation (iPlant-Collabs)...OK
Submitted batch job 1099677
[s2:login4@/work/03137/kyclark/stampede2/stampede2-template/stampede]$ qs
JOBID    PARTITION    NAME     USER     ST  TIME  NODES  NODELIST(REASON)
1099677  development  lc-test  kyclark  R   0:01  1      c455-062
$ qs
JOBID    PARTITION    NAME     USER     ST  TIME  NODES  NODELIST(REASON)
1099677  development  lc-test  kyclark  R   0:06  1      c455-062
$ qs
JOBID    PARTITION    NAME     USER     ST  TIME  NODES  NODELIST(REASON)
1099677  development  lc-test  kyclark  CG  0:08  1      c455-062
$ cat slurm-1099677.out
I will process NUM_INPUT "2" at PVALUE "5"
     1  /work/03137/kyclark/stampede2/stampede2-template/stampede/../singularity/image.def
     2  /work/03137/kyclark/stampede2/stampede2-template/stampede/../singularity/Makefile
Starting Launcher
/opt/apps/launcher/launcher-3.1/paramrun: line 171: [: -eq: unary operator expected
Launcher: Setup complete.

------------ SUMMARY ---------------
```

```
   Number of hosts:    1
   Working directory:  /work/03137/kyclark/stampede2/stampede2-template/stampede
   Processes per host: 2
   Total processes:    2
   Total jobs:         2
   Scheduling method:  interleaved
/opt/apps/launcher/launcher-3.1/paramrun: line 211: [: -eq: unary operator expected

------------------------------------
Launcher: Starting parallel tasks...
Launcher: Task 1 running job 2 on c455-062.stampede2.tacc.utexas.edu (singularity exec lc.in
Launcher: Task 0 running job 1 on c455-062.stampede2.tacc.utexas.edu (singularity exec lc.in
There are "27" lines in "/work/03137/kyclark/stampede2/stampede2-template/stampede/../singul
There are "17" lines in "/work/03137/kyclark/stampede2/stampede2-template/stampede/../singul
Launcher: Job 2 completed in 1 seconds.
Launcher: Task 1 done. Exiting.
Launcher: Job 1 completed in 1 seconds.
Launcher: Task 0 done. Exiting.
Launcher: Done. Job exited without errors
Ended LAUNCHER
Done.
```

You'll notice I'm using my `qs` alias. Add this to your "~/.bashrc" or "~/.profile" to use it:

```
alias qs='squeue -u kyclark | column -t'
```

In you directory, you will now notice a file like "slurm-*.out" and/or "*.param" which are leftover from testing. I have defined a `make clean` target to get rid of them.

Deploying Assets

Now that we have determined the app runs on Stampede2, we can upload the assets into our "deploymentSystem" (as defined in the "app.json"). Since my "copy_from_manifest.py" uses the directory name for the target location, the first thing I do is to create a versioned name for the directory:

```
[s2:login4@/work/03137/kyclark/stampede2]$ mv stampede2-template/ stampede2-template-0.0.1
```

I create a "MANIFEST" file of just the assets needed to run the app (so no JSON files, the MANIFEST itself, etc.). I have a `make up` target to help:

```
$ cat MANIFEST
lc.img
run.sh
template.sh
```

```
test.sh
$ make up
copy_from_manifest.py
Looking in "/work/03137/kyclark/stampede2/stampede2-template-0.0.1/stampede"
Found 1 MANIFEST file in "/work/03137/kyclark/stampede2/stampede2-template-0.0.1/stampede"
Processing /work/03137/kyclark/stampede2/stampede2-template-0.0.1/stampede/MANIFEST
  1: lc.img
  2: run.sh
  3: template.sh
  4: test.sh
Creating directory kyclark/applications/stampede2-template-0.0.1/stampede ...
Uploading /tmp/tmpd4ihfnhi/stampede2-template-0.0.1/stampede/lc.img...
######################################################################## 100.0%
Uploading /tmp/tmpd4ihfnhi/stampede2-template-0.0.1/stampede/run.sh...
######################################################################## 100.0%
Uploading /tmp/tmpd4ihfnhi/stampede2-template-0.0.1/stampede/template.sh...
######################################################################## 100.0%
Uploading /tmp/tmpd4ihfnhi/stampede2-template-0.0.1/stampede/test.sh...
######################################################################## 100.0%
Done, check "kyclark/applications/stampede2-template-0.0.1"
```

If you do a `files-list`, you may be tempted to think everything is good to go:

```
$ files-list kyclark/applications/stampede2-template-0.0.1/stampede
.
lc.img
run.sh
template.sh
test.sh
```

But you would be mistaken:

```
$ files-list --rich kyclark/applications/stampede2-template-0.0.1/stampede
| name        | length | permissions | type | lastModified         |
| ----        | ------ | ----------- | ---- | ------------         |
| .           | 0      | READ        | dir  | Apr 12, 2018 1:44 pm |
| lc.img      | 0      | READ        | file | Apr 12, 2018 1:44 pm |
| run.sh      | 2408   | READ        | file | Apr 12, 2018 1:44 pm |
| template.sh | 43     | READ        | file | Apr 12, 2018 1:45 pm |
| test.sh     | 175    | READ        | file | Apr 12, 2018 1:45 pm |
```

Because the all the files are being uploaded via the Agave API, the larger files such as the Singularity container will take several minutes to actually land. Once you see that all the bytes are present, you can move on with submitting a test job to Agave.

```
$ files-list --rich kyclark/applications/stampede2-template-0.0.1/stampede
| name        | length    | permissions | type | lastModified         |
| ----        | ------    | ----------- | ---- | ------------         |
```

```
| .           | 0         | READ        | dir  | Apr 12, 2018 1:44 pm |
| lc.img      | 536870944 | READ        | file | Apr 12, 2018 1:44 pm |
| run.sh      | 2408      | READ        | file | Apr 12, 2018 1:44 pm |
| template.sh | 43        | READ        | file | Apr 12, 2018 1:45 pm |
| test.sh     | 175       | READ        | file | Apr 12, 2018 1:45 pm |
```

Making App

Now you can submit your "app.json" to Agave to create the app. The `make app` target will help:

```
$ make app
apps-addupdate -F app.json
Successfully added app stampede2-template-0.0.1
```

Testing App

To test a job submission, we need to create a JSON file that describes a job. We can use the `jobs-template` command to create such a file that we can edit. The `make template` target can help:

```
$ make template
jobs-template -A stampede2-template-0.0.1 > job.json
$ cat job.json
{
  "name":"stampede2-template test-1523559134",
  "appId": "stampede2-template-0.0.1",
  "batchQueue": "serial",
  "executionSystem": "tacc-stampede-kyclark",
  "maxRunTime": "12:00:00",
  "memoryPerNode": "32GB",
  "nodeCount": 1,
  "processorsPerNode": 16,
  "archive": true,
  "archiveSystem": "data.iplantcollaborative.org",
  "archivePath": null,
  "inputs": {
    "INPUT": [
    ]
  },
  "parameters": {
    "PVALUE": 0.01
  },
```

```
  "notifications": [
    {
      "url":"https://requestbin.agaveapi.co/1e619zy1?job_id=${JOB_ID}&status=${JOB_STATUS}",
      "event":"*",
      "persistent":true
    },
    {
      "url":"kyclark@gmail.com",
      "event":"FINISHED",
          "persistent":false
    },
    {
      "url":"kyclark@gmail.com",
      "event":"FAILED",
      "persistent":false
    }
  ]
}
```

Noice the "PVALUE" has a default, but we need to supply something for the "INPUT" which will need to be a path in the Data Store. We can `files-list` our Cyverse "home" directory to find something appropriate. I will use "kyclark/data/dolphin/fasta" and then `make job` to submit the updated JSON:

```
$ make job
jobs-submit -F job.json
Successfully submitted job 8145541748635865576-242ac113-0001-007
```

I can use the `jobs-status` command with the "-W" flag to "watch" the job go through it's normal staging/queueing/running phases. When the job reaches "FINISHED," the watch will end:

```
Watching job 8145541748635865576-242ac113-0001-007
Thu Apr 12 14:01:18 CDT 2018
FINISHED
Terminating watch
```

Staging

It's important to understand that all the "inputs" for a job are copied to the compute nodes before the job is run, so files that are in the Data Store (located in Tucson) must be transferred to Austin. Inputs are not limited to file in the DS – they can also be indicated by URL (HTTP, FTP, etc.). In addition, the files in the "deploymentPath" (from "app.json") are copied from the "deploymentSystem" to the compute node. This means that all files will essentially be "local." Agave

will turn “kyclark/data/dolphin/fasta” into just “fasta” and pass that as the “-i” argument.

Results

Any files you write will be copied along with all the inputs back into the users DS. The Singularity image will be deleted. You can use `jobs-output-list` to see what the job created:

```
$ jobs-output-list 8145541748635865576-242ac113-0001-007
fasta
.agave.log
stampede2-template-test-1523563854-8145541748635865576-242ac113-0001-007.err
stampede2-template-test-1523563854-8145541748635865576-242ac113-0001-007.out
```

Here you can see the “fasta” directory that was copied to the node:

```
$ jobs-output-list 8145541748635865576-242ac113-0001-007 fasta
Dolphin_1_z04.fa
Dolphin_2_z09.fa
Dolphin_3_z11.fa
Dolphin_4_z12.fa
Dolphin_6_z21.fa
Dolphin_7_z22.fa
Dolphin_8_z26.fa
```

Along with the standard “.agave.log” file. To view that, we can bring down the file:

```
$ jobs-output-get 8145541748635865576-242ac113-0001-007 .agave.log
################################################################################ 100.0%
$ cat .agave.log
[2018-04-12T15:19:40-0500]
[2018-04-12T20:32:01-0500] {"status":"success","message":null,"version":"2.2.19-rb3e2018","r
[2018-04-12T20:32:09-0500] {"status":"success","message":null,"version":"2.2.19-rb3e2018","r
```

Our job didn’t create any output on disk, just STDOUT which we can view in the jobs “*.out" file:

```
$ jobs-output-get 8145541748635865576-242ac113-0001-007 stampede2-template-test-1523563854-8
$ cat stampede2-template-test-1523563854-8145541748635865576-242ac113-0001-007.out
I will process NUM_INPUT "7" at PVALUE "0.01"
     1  fasta/Dolphin_7_z22.fa
     2  fasta/Dolphin_1_z04.fa
     3  fasta/Dolphin_6_z21.fa
     4  fasta/Dolphin_3_z11.fa
     5  fasta/Dolphin_4_z12.fa
```

```
     6  fasta/Dolphin_2_z09.fa
     7  fasta/Dolphin_8_z26.fa
Starting Launcher
Launcher: Setup complete.

------------- SUMMARY ---------------
   Number of hosts:    1
   Working directory:  /work/03137/kyclark/stampede2/kyclark/job-8145541748635865576-242ac11
   Processes per host: 7
   Total processes:    7
   Total jobs:         7
   Scheduling method:  interleaved

-------------------------------------
Launcher: Starting parallel tasks...
Launcher: Task 0 running job 1 on c450-103.stampede2.tacc.utexas.edu (singularity exec lc.im
Launcher: Task 1 running job 2 on c450-103.stampede2.tacc.utexas.edu (singularity exec lc.im
Launcher: Task 2 running job 3 on c450-103.stampede2.tacc.utexas.edu (singularity exec lc.im
Launcher: Task 3 running job 4 on c450-103.stampede2.tacc.utexas.edu (singularity exec lc.im
Launcher: Task 5 running job 6 on c450-103.stampede2.tacc.utexas.edu (singularity exec lc.im
Launcher: Task 6 running job 7 on c450-103.stampede2.tacc.utexas.edu (singularity exec lc.im
Launcher: Task 4 running job 5 on c450-103.stampede2.tacc.utexas.edu (singularity exec lc.im
There are "153611" lines in "fasta/Dolphin_4_z12.fa"
Launcher: Job 5 completed in 3 seconds.
Launcher: Task 4 done. Exiting.
There are "206963" lines in "fasta/Dolphin_2_z09.fa"
Launcher: Job 6 completed in 3 seconds.
There are "243058" lines in "fasta/Dolphin_3_z11.fa"
Launcher: Task 5 done. Exiting.
Launcher: Job 4 completed in 3 seconds.
Launcher: Task 3 done. Exiting.
There are "368606" lines in "fasta/Dolphin_8_z26.fa"
Launcher: Job 7 completed in 3 seconds.
Launcher: Task 6 done. Exiting.
There are "426666" lines in "fasta/Dolphin_1_z04.fa"
Launcher: Job 2 completed in 3 seconds.
Launcher: Task 1 done. Exiting.
There are "461033" lines in "fasta/Dolphin_6_z21.fa"
Launcher: Job 3 completed in 3 seconds.
Launcher: Task 2 done. Exiting.
There are "585382" lines in "fasta/Dolphin_7_z22.fa"
Launcher: Job 1 completed in 3 seconds.
Launcher: Task 0 done. Exiting.
Launcher: Done. Job exited without errors
Ended LAUNCHER
Done.
```

You can use the `jobs-history` to see all the steps Agave took to stage and run the job:

```
$ jobs-history 8145541748635865576-242ac113-0001-007
Job accepted and queued for submission.
Attempt 1 to stage job inputs
Identifying input files for staging
Copy in progress
Job inputs staged to execution system
Preparing job for submission.
Attempt 1 to submit job
Fetching app assets from agave://data.iplantcollaborative.org/kyclark/applications/stampede2
Staging runtime assets to agave://tacc-stampede2-kyclark/kyclark/job-8145541748635865576-242
HPC job successfully placed into normal queue as local job 1100184
Job started running
Job completed execution
Beginning to archive output.
Attempt 1 to archive job output
Archiving agave://tacc-stampede2-kyclark/kyclark/job-8145541748635865576-242ac113-0001-007-s
Job archiving completed successfully.
Job complete
```

Sharing and Publishing App

You can now use your app, submitting jobs from any place you have installed the Cyverse SDK. You can also share you app with another user, but you will also have to share your execution and deployment systems.

To deploy an app to iMicrobe, it's necessary to request (via Agave Slack, probably Fonner) to have the app be made public. The public app ID will usually have "u+" appended, e.g., "stampede2-template-0.0.1u1". If you provide this ID to me, I can add it to the apps listed on iMicrobe so that our users can run it.

See Also

https://cyverse.github.io/cyverse-sdk/docs/cyversesdk.html