Day 14: Wrapper Scripts

Turn In Homework

Homework Review

Introduction

Wrapper Scripts

- Script that runs your real executable
- Named as executable in submit file
- Runs ***on the execute machine***

```
#!/usr/bin/env python
import os
# Do stuff before running real executable
os.system('real-job arg1 arg2 arg3 ...')
# Do stuff after running real executable
```

Example Submit File With Wrapper

```
executable = wrapper.py
transfer_input_files = real-job, input, …
```

- Condor automatically transfers file in **executable**
- But, ***real*** executable must be named explicitly
- Include with any other input files to transfer

Why Use a Wrapper Script?

Handle jobs with complex run-time requirements

- Before execution
  - Prepare files and/or executable
  - Set up environment variables
- Execution
  - Prepare complex command-line arguments
  - Batch together many little jobs
- After execution
  - Find, filter, and/or consolidate output files
  - Compress output files

Two Key Principles

Be Kind to Your Submit Machine

- Typically, submit machine is shared resource
  - Like submit-368 (only worse)
- Many tasks run there
  - condor_submit
  - condor_schedd
  - 1 condor_shadow per running job
  - DAGMan pre- and post-scripts
  - Maybe others
- Thus, avoid doing ***anything*** substantial there
  - Especially affecting CPU, memory, or disk

Bring It With You

- Applies to ***everything*** your job needs to run
- Obvious
  - Executable
  - Input data and command-line arguments
- Less obvious
  - Underlying software (e.g., R, MATLAB, Octave)
  - Run-time libraries and other software dependencies
  - Configuration and environment
  - Directory layouts
- Especially important in Open Science Grid

Before Execution

Unpacking Files

- May have files bundled together in archive
- May be compressed (but see next slide)
- Common tools: **tar**, **unzip**, **gunzip**, **bunzip2**
- Good to check exit status, messages, and a file or 2

```
cmd = ['tar', 'xzf', 'big-data.tar.gz']
status, stdout, stderr = my_system(cmd)
if status != 0:
    myfail('untar failed: %d' % (status))
if re.search(r'[Ee]rror', stderr):
    myfail('untar error: %s' % (stderr))
if not os.path.isdir('big-data-dir'):
    myfail('no data dir!')
```

Caveats About Large Input Data

- Remember the principle about submit machines
  - Compressing large files takes lots of CPU and disk I/O
  - Do ***not*** archive/compress big data on submit machine
    - Command-line
    - DAGMan pre-scripts
    - local or scheduler universe
- Great to do elsewhere, ahead of time
  - *Maybe* as vanilla universe job; still frowned upon
  - Otherwise, just transfer files or even whole directories
- ***Or*** place big data files elsewhere, and download to execute machine from wrapper script!

Prepare Files and Directories

- All input files end up in top-level execute directory
- Unpacking an archive may yield subdirectories
- Your job may need input files organized differently
- May need other directories/files (e.g., for output)

```
unpack_input_archive('big-data.tar.gz')

os.mkdir('input')

shutil.copy('params.txt', 'input/p.conf')
os.chmod(0400, 'input/p.conf')
shutil.move('big-data', 'input/samples')

os.mkdir('output')
```

Refresher: Environment Variables

- **`os.environ`** : dictionary of environment variables
- Readable and writable; inherited by subprocesses
- May need to prep environment for real executable
- Consult its documentation for names & meanings

```
home = os.getcwd()
r_file = os.path.join(home, 'R.env')
if os.path.exists(r_file):
    os.environ['R_ENVIRON_USER'] = r_file
else:
    print >> sys.stderr, 'No R environ!'
```

Finding Programs

- **PATH** tells system where to find programs to run
- Set if your executable runs another program that is in a weird location (e.g., that the job brought along)
- Usually, prepend to existing **PATH**; colon separated

```
home = os.getcwd()
myzip = os.path.join(home, 'myzip', 'bin')
if os.path.isdir(myzip):
    os.environ['PATH'] = myzip + ':' + \
        os.environ['PATH']
else:
    print >> sys.stderr, 'No myzip dir!'
```

Finding (Dynamic) Libraries

- When bringing along compiled code, may need to tell system where to find its libraries (**`*.so`**)
- Add to **`LD_LIBRARY_PATH`** environment variable
- May need to ask a sysadmin for help!

```
LLP = 'LD_LIBRARY_PATH'
home = os.getcwd()
myzip = os.path.join(home, 'myzip', 'lib')
if os.environ.has_key(LLP):
    os.environ[LLP] += ':' + myzip
else:
    os.environ[LLP] = myzip
```

Execution

Refresher: System Calls

- Run sub-shell, which runs command, no output:

```
exit_status = os.system('echo $PATH')
```

- More complexity, more control:
  - Sub-shell only on demand
  - Get output and sane exit status code
  - Command and arguments as sequence elements

```
def my_system(command, shell=False):
    p = subprocess.Popen(command, shell=shell,
                         stdout=subprocess.PIPE,
                         stderr=subprocess.PIPE)
    (stdout, stderr) = p.communicate()
    return (p.returncode, stdout, stderr)
status, stdout, stderr = my_system(['foo', 'arg'])
```

Parameter Conversions I

- Command arguments can be complicated & messy
- Wrapper can offer simpler command-line interface

```
% R CMD BATCH --args arg1 arg2 foo.R
```

```
% Rscript foo.R arg1 arg2
```

- Wrapper scripts could:
  - Hardcode "extra" arguments (e.g., **`CMD BATCH --args`**)
  - Compute arguments from simpler one(s) (e.g., fractal)
  - Look up arguments in table (e.g., dictionary, file)

Parameter Conversions II

- Sketch of a file/dictionary conversion

```
params = {}
pfile = open('parameters.txt')
for line in pfile:
    parts = line.strip().split()
    params[parts[0]] = parts[1:]
pfile.close()

case = sys.argv[1]
if not params.has_key(case):
    print >> sys.stderr, 'Bad case', case
    sys.exit(1)
arguments = params[case]
```

Batching I

- Remember: Ideal job duration is 10 min – 4 hours
- Imagine app. runs for 3 secs… but there are 100K!
  - Total CPU time is 300K secs = **3d 11h 20m**
  - If 60 secs overhead; total time is 6.3M secs = **72d 22h**
- One solution: Group many small tasks per job
  - 100 jobs × 3000 runs; 60 s overhead; 306K secs (**+2%**)
- Good case for a DAG
  - Script creates job-sized units of work, creates inputs
  - Wrapper script responsible for running app. *N* times
  - Final node brings together all results

Batching II

- Sketch of a batching wrapper
- Similar to the prime-number counter in many ways

```
start, end = sys.argv[1:3]

for i in xrange(start, end + 1):

  cmd = ['foo'] + calculate_args(i)
  status, stdout, stderr = my_system(cmd)

  if status != 0:
    # Handle error; continue, break, exit?

  record_output(i, stdout)
```

After Execution

Prepare Output Files I

- Program may put key output files in strange places
- By default, Condor transfers only new and changed files ***in top-level directory*** on execute machine
- Two approaches (use alone or in combination):
  - Tell Condor where to expect your output files
  - Move output files to where Condor expects them
- Rename files to identify better or avoid conflicts
- Also, consider archiving and compressing output (similar caveats apply as with input files)

Prepare Output Files II

- Suppose CSV output is scattered among subdirs

```
# Condor submit file
transfer_output_files = main.out, outputs/
```

```
os.mkdir('outputs')
n = 0
for dir, x, f in os.walk('foo-out'):
  for file in fnmatch.filter(f, '*.csv'):
    src = os.path.join(dir, file)
    new_fn = '%04d_%s' % (n, file)
    dst = os.path.join('outputs', new_fn)
    shutil.move(src, dst)
    n += 1
```

Being Selective About Output

- Maybe only a small fraction of output data matters
- Take time ***on execute machine*** to shrink output files

```
original = open(output_filename)
realdata = open(new_output_filename, 'w')
for line in original:
    if re.search(r'wibble', line):
        realdata.write(line)
realdata.close()
original.close()

cmd = 'gzip -9 ' + new_output_filename
exit_status = os.system(cmd)
# check for failure!
```

Complex Runtimes

The MATLAB Syndrome

- Need a license to run "normal" MATLAB
- But not compiled MATLAB
- But, runtime version must match compiler version
- Many CHTC/MATLAB jobs are forwarded to OSG
- No idea what MATLAB will exist, if any
- Also, may need non-standard libraries…
- Plus configuration…
- Yikes!

Some Approaches

- Essentially, bring ***everything*** with the job
  - MATLAB runtime (~ 200 MB comp., ~ 500 MB uncomp.)
  - All software and library dependencies
  - Extra MATLAB libraries & configuration
  - Compiled MATLAB script(s), inputs, arguments
- Moving toward virtual machines (cf. Amazon EC2)
  - Take entire Linux machine with you!
  - Literally replicates your entire environment
  - There is a performance penalty, but do you care?
- CDE: Bring everything you need, but not whole VM
  `http://www.stanford.edu/~pgbovine/cde.html`

Homework

Homework

- Play cards… a lot! (10M–100M times)
- Write a wrapper script for a C program
  - Batch runs
  - Filter output
- Optional: Do post-processing analysis and graph

Course Evaluations

- Must be enrolled to fill out
- Use #2 pencil only
- Be sure to fill out top part:
  Instructor: Tim Cartwright Course #: 368 Section #: 004
- Please write constructive comments on back!
- Need volunteer to take forms and pencils to Cathy Richard, Comp Sci 5360