Day 13: Scripting Workflows II

DAGMan
Turn In Homework
Homework Review
Advanced DAGMan
Retrying Nodes

```
RETRY name count UNLESS-EXIT value
```

- Specifies number of times to retry given node
- Affects entire node, not just its job
- Especially useful if job is sensitive to environment

```
JOB Analyzel analysis.sub
RETRY Analyzel 3 UNLESS-EXIT 99
```
Node Directories

**JOB name submit-file DIR directory**

- Use `directory` for all files for this node
- Submit file, executable, inputs, outputs, *everything*
- Effectively:
  ```
  cd directory
  condor_submit submit-file
  ```
- In submit, reference common files as, e.g., `../foo`

```
JOB Wibble wibble.sub DIR wibble
%
ls wibble
go-wibble.py input-1.txt wibble.sub
```
Node Priorities

**PRIORITY** *name* *value*

- Sets *DAGMan* priority for the given node
- Determines when *DAGMan* submits job to queue
- Hence, different than job priority (set in submit file)
- Useful when throttling jobs (*-maxjobs*, *-maxidle*)
- Integer (+/–), defaults to 0, higher is better

```plaintext
JOB Analyze1 analysis.sub
PRIORITY Analyze1 10

JOB Analyze2 analysis.sub
PRIORITY Analyze2 5
```
Skipping Nodes

**PRE_SKIP** name *exit-status*

- If node’s Pre-Script exits with the given exit status, skip rest of node
- Node is marked as successful

**JOB** Foo foo.sub
**SCRIPT** PRE Foo set-up-foo.py
**PRE_SKIP** Foo 1
Node Variables

VARS name macroname="value" ...

- Define *macro(s)* (= variable(s)) for submit file
- *macro name* is \w+, cannot start with *queue*
- Multiple macros for node on same line, or separate
- In value, $(JOB)$ expands to node *name*

```bash
JOB Foo foo.sub
VARS Foo arg1="hello" arg2="42"
VARS Foo arg3="$(JOB)"
```
Using Node Variables

- In submit file, reference macro as $(macroname)

### Example

```
JOB Foo foo.sub
VARS Foo arg1="hello" arg2="42"
VARS Foo arg3="$(JOB)"

executable = /bin/echo
universe = local
output = test.out
error = test.err
log = test.log
arguments = "A1=$(arg1) A2=$(arg2) ..."
queue
```
Node Variables Can Simplify Submit Files

- Move data from *many* submit files to 1 DAGMan file
- Use VARS, $(cluster)$, and/or $(process)$

```shell
JOB Analysis1 analysis.sub
VARS Analysis1 jobname="$(JOB)" arg="ABW"

JOB Analysis2 analysis.sub
VARS Analysis2 jobname="$(JOB)" arg="ADO"

output = analysis.$(jobname).out
error = analysis.$(jobname).err
log = analysis.log
arguments = "$(arg)"
queue
```
Scripting Simple DAGs
Designing DAGs for Scripting

• Mostly, focus on wide, parallel parts
• Consider pros and cons of each choice

• **VARS** and 1 submit file, or 1 submit file per node?
  – Often easier to script one complex DAG submit file
  – Submit file can specify subdirectories (**initialdir**)

• Use sub-directories?
  – Same considerations as without DAG
  – More useful with distinct inputs or lots of output files
  – Put common files in **../** or **../common/**

• Consider using DAGMan for independent jobs
def psub(text): ... # add text to submit file
psub(dag_submit_header)

n = 0
for t in product(parameter_1, parameter_2):
    n += 1
    psub('JOB N%d node.sub DIR node-%d' % (n, n))
    psub('RETRY N%d 3 UNLESS-EXIT 1' % (n))
    if t[0] < 1.0: psub('PRIORITY N%d 10' % (n))
    args = '%d %s' % (n, t[1])
    psub('SCRIPT PRE N%d pre.py %s' % (n, args))
    psub('PARENT Start CHILD N%d' % (n))
write_node_dir(sources, n, t)

psub(dag_submit_footer)
Setting Up Node Directories

- Much like before, but need to include submit file

```python
# sources: dict from filename to contents

def prepare_node_dir(sources, node, params):
    node_dir = 'node-%d' % (node)
    os.mkdir(node_dir)

    # write node submit file, incl. job arguments
    node_sub = os.path.join(node_dir, 'node.sub')
    write_node_submit(node_sub, params)

    for filename in sources:
        text = sources[filename]
        target = os.path.join(dirname, filename)
        write_template(text, target, params)
```
Splices
Understanding Splices

- Reusable DAG fragment, *inserted into* larger DAG
- Like a function, if you think about it
- Common use: write outer DAG once, replace insides
Splice Syntax

**SPLICE** name *inner-dag-file* DIR directory

- Like the **JOB** statement, except it names a **DAG file**
- All nodes in splice become part of (outer) DAG
- Can create **PARENT / CHILD** relationships for splice, which affect all of its initial/final nodes

**JOB** Start start.sub
**JOB** End end.sub

**SPLICE** Diamond1 diamond.dag
**SPLICE** Diamond2 diamond.dag

**PARENT** Start **CHILD** Diamond1 Diamond2
# Splice

```plaintext
JOB A a.sub
VARS A x="$(JOB)"
JOB B b.sub
VARS B x="$(JOB)"
PARENT A CHILD B
```

# Outer

```plaintext
JOB X x.sub
SPLICE Y000 spl.dag
...
SPLICE Y999 spl.dag
JOB Z z.sub
PARENT X CHILD Y000
PARENT Y000 CHILD Z
```
Sub-DAGs
Understanding Sub-DAGs

- Reusable DAG fragment, *submitted by* larger DAG
- Also like a function, if you think about it
- Splices are better in most cases, except for one…
SUBDAG Syntax

**SUBDAG EXTERNAL** *name inner-dag* **DIR** *dir*

- Like the **JOB** statement, except it names *a DAG file*
- Nodes in sub-DAG **do not** become part of DAG
- DAGman submits *inner-dag* when job is run

```
JOB Start start.sub
JOB End end.sub
```

```
SUBDAG EXTERNAL Diamond1 diamond.dag
SUBDAG EXTERNAL Diamond2 diamond.dag
```

```
PARENT Start CHILD Diamond1 Diamond2
PARENT Diamond1 Diamond2 CHILD End
```
Running Nested DAGs

- DAGMan does `condor_submit_dag` on DAG file
  - Hence, another copy of DAGMan is running
  - If there are many copies, submit machine may suffer

- Sub-DAG not processed until needed
  - Allows for some cool tricks...
  - Errors not discovered until run-time!

- Rescue DAGs are complicated, but still work
Dynamic DAGs
The Need for Dynamic DAGs

- Suppose the exact number of parallel jobs depends on some initial (significant) input processing
  - ... or exact number of stages ...
  - ... or exact DAG shape ...

- **We could:**
  - Run one job to process input, then...
  - Manually run script to generate rest of DAG
  - But we want to automate!

- Dynamic DAG — build (part of) DAG *during* run
Dynamic DAGs

• How to implement:
  – In DAG, add one or more **SUBDAG EXTERNAL** nodes
  – (Re)Write their DAGMan submit files in earlier node (or, even in the node’s pre-script!)

• Again, errors not found until sub-DAG is submitted

• Outer DAG can be very simple and/or generic:
Dynamic DAG Example

- DAGMan submit file for simple, generic outer DAG:

```plaintext
JOB Start start.sub
SUBDAG EXTERNAL Innards dynamic.dag
JOB End end.sub
SCRIPT PRE Innards generate-dag.py
PARENT Start CHILD Innards
PARENT Innards CHILD End
```
Workflow Management Systems
makeflow

• Different way to describe workflow DAG
  – Uses syntax like `make`
  – Handles data transfers (so does Condor/DAGMan)
  – Highly fault tolerant (so is DAGMan)

• Works with several distributed computing systems
  – Condor
  – Sun Grid Engine (SGE)
  – Work Queue (also from CCL)

• From Doug Thain’s Cooperative Computing Lab
  http://nd.edu/~ccl/software/makeflow/
Pegasus WMS

- Supports higher-level workflow abstractions
- Compiles down to DAG
- Works with Condor, OSG, Amazon EC2, TeraGrid, …
- Used on a wide variety of complex science projects
- Lots of cool example applications online
- From *Information Sciences Institute*, USC
  
  http://pegasus.isi.edu/
SOAR

- System Of Automated Runs
- Automatically scans directories for jobs to run
- Each “job” can be a complete DAG in itself
- Puts jobs into DAG and manages workflow
- Also handles R and MATLAB jobs well
- Provides extra tracking and reporting tools
- From Bill Taylor, CHTC Team

http://submit.chtc.wisc.edu/SOAR/
Homework
Homework

• Script a workflow!

• Using the Mandelbrot generator again, but adding the stitching step at the end

• **Note:** Use a different universe (**scheduler**) for the **montage** node (**only**)!

• If you have an alternate workflow that you would like to work on instead, talk to me