AN INTRODUCTION TO WORKFLOWS WITH DAGMAN

Presented by Lauren Michael
Covered In This Tutorial

• Why Create a Workflow?
• Describing workflows as directed acyclic graphs (DAGs)
• Workflow execution via DAGMan (DAG Manager)
• Node-level options in a DAG
• Modular organization of DAG components
• DAG-level control
• Additional DAGMan Features
Automation!

- Objective: Submit jobs in a particular order, **automatically**.
- Especially if: Need to reproduce the same workflow multiple times.
DAG = "directed acyclic graph"

- topological ordering of vertices ("nodes") is established by directional connections ("edges")
- "acyclic" aspect requires a start and end, with no looped repetition
  - can contain cyclic subcomponents, covered in later slides for workflows

wikipedia.org/wiki/Directed_acyclic_graph
HTCondor has a **DAG Manager (DAGMan)!**


- DAGMan Applications
  - DAGMan Terminology
  - The DAG Input File: Basic Commands
  - Command Order
  - Node Job Submit File Contents
  - DAG Submission
  - File Paths in DAGs
  - DAG Monitoring and DAG Removal
  - Suspending a Running DAG
  - Advanced Features of DAGMan
  - The Rescue DAG
  - DAG Recovery
  - Visualizing DAGs with *dot*
  - Capturing the Status of Nodes in a File
  - A Machine-Readable Event History, the *jobstate.log* File
  - Status Information for the DAG in a ClassAd
  - Utilizing the Power of DAGMan for Large Numbers of Jobs
  - Workflow Metrics
  - DAGMan and Accounting Groups

- Virtual Machine Applications
  - The Submit Description File
  - Checkpoints
  - Disk Images
An Example HTC Workflow

- User must communicate the “nodes” and directional “edges” of the DAG
Simple Example for this Tutorial

- The DAG input file communicates the “nodes” and directional “edges” of the DAG.
Basic DAG input file:
JOB nodes, PARENT-CHILD edges

my.dag

JOB A A.sub
JOB B1 B1.sub
JOB B2 B2.sub
JOB B3 B3.sub
JOB C C.sub
PARENT A CHILD B1 B2 B3
PARENT B1 B2 B3 CHILD C

- Node names are used by various DAG features to modify their execution by DAG Manager.
Basic DAG input file: JOB nodes, PARENT-CHILD edges

```plaintext
my.dag

JOB A A.sub
JOB B1 B1.sub
JOB B2 B2.sub
JOB B3 B3.sub
JOB C C.sub
PARENT A CHILD B1 B2 B3
PARENT B1 B2 B3 CHILD C
```

(dag_dir)/

```
A.sub   B1.sub
B2.sub   B3.sub
C.sub   my.dag
(other job files)
```

• Node names and filenames can be anything.
• Node name and submit filename do not have to match.
Endless Workflow Possibilities

[Diagram showing workflow possibilities with nodes labeled A to I and processes like fastQSplit, filterContams, sol2sanger, etc.]

[Links to Wikimedia Commons and Confluence page]

https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator
DAGs are also useful for non-sequential work

‘bag’ of HTC jobs

disjointed workflows
Basic DAG input file: JOB nodes, PARENT-CHILD edges

```
my.dag

JOB A A.sub
JOB B1 B1.sub
JOB B2 B2.sub
JOB B3 B3.sub
JOB C C.sub
PARENT A CHILD B1 B2 B3
PARENT B1 B2 B3 CHILD C
```

Diagram:

```
A

B1 --> C
B2 --> C
B3 --> C
...
BN --> C
```

Reference:
HTCondor Manual: DAGMan Applications > DAG Input File
Submitting and Monitoring a DAGMan Workflow
Submitting a DAG to the queue

• Submission command:

`condor_submit_dag dag_file`

```
$ condor_submit_dag my.dag

File for submitting this DAG to HTCondor : my.dag.condor.sub
Log of DAGMan debugging messages : my.dag.dagman.out
Log of HTCondor library output : my.dag.lib.out
Log of HTCondor library error messages : my.dag.lib.err
Log of the life of condor_dagman itself : my.dag.dagman.log

Submitting job(s).
1 job(s) submitted to cluster 87274940.
```
A submitted DAG creates a DAGMan job process in the queue

- DAGMan runs on the submit server, as a job in the queue
- **At first:**

```
$ condor_q
-- Schedd: submit-3.cltc.wisc.edu : <128.104.100.44:9618>
OWNER BATCH_NAME SUBMITTED DONE RUN IDLE TOTAL JOB_IDS
alice my.dag+128 4/30 18:08 _ _ _ _ 0.0
1 jobs; 0 completed, 0 removed, 0 idle, 1 running, 0 held, 0 suspended

$ condor_q -nobatch
-- Schedd: submit-3.cltc.wisc.edu : <128.104.100.44:9618>
ID OWNER SUBMITTED RUN_TIME ST PRI SIZE CMD
128.0 alice 4/30 18:08 0+00:00:06 R 0 0.3 condor_dagman
1 jobs; 0 completed, 0 removed, 0 idle, 1 running, 0 held, 0 suspended
```
Jobs are automatically submitted by the DAGMan job

- Seconds later, node A is submitted:

```bash
$ condor_q
-- Schedd: submit-3.chtc.wisc.edu : <128.104.100.44:9618>?

OWNER BATCH_NAME SUBMITTED DONE RUN IDLE TOTAL JOB_IDS
alice my.dag+128 4/30 18:08 _ _ 1 5 129.0
2 jobs; 0 completed, 0 removed, 1 idle, 1 running, 0 held, 0 suspended

$ condor_q -nobatch
-- Schedd: submit-3.chtc.wisc.edu : <128.104.100.44:9618>?

ID OWNER SUBMITTED RUN_TIME ST PRI SIZE CMD
128.0 alice 4/30 18:08 0+00:00:36 R 0 0.3 condor_dagman
129.0 alice 4/30 18:08 0+00:00:00 I 0 0.3 A_split.sh
2 jobs; 0 completed, 0 removed, 1 idle, 1 running, 0 held, 0 suspended
```
Jobs are automatically submitted by the DAGMan job

- After A completes, B1-3 are submitted

```bash
$ condor_q
-- Schedd: submit-3.chtc.wisc.edu : <128.104.100.44:9618?... 
OWNER   BATCH_NAME SUBMITTED  DONE  RUN  IDLE  TOTAL  JOB_ID
alice   my.dag+128 4/30 18:08 1  _  3  5  130.0 ... 132.0
4 jobs; 0 completed, 0 removed, 3 idle, 1 running, 0 held, 0 suspended

$ condor_q -nobatch
-- Schedd: submit-3.chtc.wisc.edu : <128.104.100.44:9618?... 
ID   OWNER SUBMITTED  RUN_TIME  ST  PRI  SIZE  CMD
128.0 alice  4/30 18:08 0+00:20:36  R  0  0.3 condor_dagman
130.0 alice  4/30 18:28 0+00:00:00  I  0  0.3 B_run.sh
131.0 alice  4/30 18:28 0+00:00:00  I  0  0.3 B_run.sh
132.0 alice  4/30 18:28 0+00:00:00  I  0  0.3 B_run.sh
4 jobs; 0 completed, 0 removed, 3 idle, 1 running, 0 held, 0 suspended
```
Jobs are automatically submitted by the DAGMan job

- After **B1-3** complete, node **C** is submitted

```
$ condor_q
-- Schedd: submit-3.chtc.wisc.edu : <128.104.100.44:9618>?...
OWNER      BATCH_NAME  SUBMITTED  DONE  RUN  IDLE  TOTAL  JOB_IDS
alice      my.dag+128  4/30 18:08  4    _   1     5   133.0
2 jobs; 0 completed, 0 removed, 1 idle, 1 running, 0 held, 0 suspended

$ condor_q -nobatch
-- Schedd: submit-3.chtc.wisc.edu : <128.104.100.44:9618>?...
ID        OWNER   SUBMITTED  RUN_TIME  ST  PRI  SIZE  CMD
128.0     alice   4/30 18:08  0:00:46:36 R  0    0.3  condor_dagman
133.0     alice   4/30 18:54  0:00:00:00 I  0    0.3  C_combine.sh
2 jobs; 0 completed, 0 removed, 1 idle, 1 running, 0 held, 0 suspended
```
Status files are Created at the time of DAG submission

(dag_dir)/

| A.sub         | B1.sub       | B2.sub       |
| B3.sub        | C.sub        | *(other job files)* |
| **my.dag**    | **my.dag.condor.sub** | **my.dag.dagman.log** |
| **my.dag.dagman.out** | **my.dag.lib.err** | **my.dag.lib.out** |
| **my.dag.nodes.log** |

* `.condor.sub` and `.dagman.log` describe the queued DAGMan job process

* `.dagman.out` has detailed logging (look to first for errors)

* `.lib.err/out` contain std err/out for the DAGMan job process

* `.nodes.log` is a combined log of all jobs within the DAG
DAG Completion

(dag_dir)/

<table>
<thead>
<tr>
<th>A.sub</th>
<th>B1.sub</th>
<th>B2.sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>B3.sub</td>
<td>C.sub</td>
<td>(other job files)</td>
</tr>
<tr>
<td>my.dag</td>
<td>my.dag.condor.sub</td>
<td>my.dag.dagman.log</td>
</tr>
<tr>
<td>my.dag.dagman.out</td>
<td>my.dag.lib.err</td>
<td>my.dag.lib.out</td>
</tr>
<tr>
<td>my.dag.nodes.log</td>
<td>my.dag.dagman.metrics</td>
<td></td>
</tr>
</tbody>
</table>

*.dagman.metrics is a summary of events and outcomes

*.dagman.log will note the completion of the DAGMan job

*.dagman.out has detailed logging for all jobs (look to first for errors)
Removing a DAG from the queue

- Remove the DAGMan job in order to stop and remove the entire DAG:
  
  ```
  condor_rm dagman_jobID
  ```

$ condor_q

-- Schedd: submit-3.chtc.wisc.edu : <128.104.100.44:9618>...

<table>
<thead>
<tr>
<th>OWNER</th>
<th>BATCH_NAME</th>
<th>SUBMITTED</th>
<th>DONE</th>
<th>RUN</th>
<th>IDLE</th>
<th>TOTAL</th>
<th>JOB_IDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice</td>
<td>my.dag+128</td>
<td>4/30 18:08</td>
<td>4</td>
<td>_</td>
<td>1</td>
<td>6</td>
<td>133.0</td>
</tr>
</tbody>
</table>

2 jobs; 0 completed, 0 removed, 1 idle, 1 running, 0 held, 0 suspended

$ condor_rm 128

All jobs in cluster 128 have been marked for removal

- Creates a rescue file so that only incomplete or unsuccessful NODES are repeated upon resubmission.
Node Failures Result in DAG Failure and Removal

• If a node JOB fails (non-zero exit code)
  – DAGMan continues to run other JOB nodes until it can no longer make progress

• Example at right:
  – B2 fails
  – Other B* jobs continue
  – DAG fails and exits after B* and before node C
Best Control Achieved with One Process per JOB Node

- While submit files can ‘queue’ many processes, a **single process per submit** file is usually best for DAG JOBs
  - Failure of any process in a JOB node results in failure of the entire node and immediate removal of other processes in the node.
  - RETRY of a JOB node resubmits the entire submit file.
Resolving held node jobs

- Look at the hold reason (in the job log, or with `condor_q -hold`)
- Fix the issue and release the jobs (`condor_release`)
- OR - remove the entire DAG, resolve, then resubmit the DAG
Beyond the Basic DAG: Node-level Modifiers
By default, JOB files are relative to the DAG submission directory

```
my.dag
```

<table>
<thead>
<tr>
<th>JOB</th>
<th>sub file</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A.sub</td>
</tr>
<tr>
<td>B1</td>
<td>B1.sub</td>
</tr>
<tr>
<td>B2</td>
<td>B2.sub</td>
</tr>
<tr>
<td>B3</td>
<td>B3.sub</td>
</tr>
<tr>
<td>C</td>
<td>C.sub</td>
</tr>
</tbody>
</table>

PARENT A CHILD B1 B2 B3
PARENT B1 B2 B3 CHILD C

```
(dag_dir)/
```

<table>
<thead>
<tr>
<th>sub file</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.sub</td>
</tr>
<tr>
<td>B1.sub</td>
</tr>
<tr>
<td>B2.sub</td>
</tr>
<tr>
<td>B3.sub</td>
</tr>
<tr>
<td>C.sub</td>
</tr>
<tr>
<td>my.dag</td>
</tr>
</tbody>
</table>

(other job files)

• What if you want to organize different JOB node files in different directories?
Designate different submission directories with DIR

- combine DIR with submit file contents (file paths) to achieve your desired organization

**my.dag**

<table>
<thead>
<tr>
<th>JOB</th>
<th>A</th>
<th>A.sub</th>
<th>DIR A</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOB</td>
<td>B1</td>
<td>B1.sub</td>
<td>DIR B</td>
</tr>
<tr>
<td>JOB</td>
<td>B2</td>
<td>B2.sub</td>
<td>DIR B</td>
</tr>
<tr>
<td>JOB</td>
<td>B3</td>
<td>B3.sub</td>
<td>DIR B</td>
</tr>
<tr>
<td>JOB</td>
<td>C</td>
<td>C.sub</td>
<td>DIR C</td>
</tr>
</tbody>
</table>

PARENT A CHILD B1 B2 B3
PARENT B1 B2 B3 CHILD C

(dag_dir)/

<table>
<thead>
<tr>
<th>my.dag</th>
</tr>
</thead>
<tbody>
<tr>
<td>A/ A.sub (A job files)</td>
</tr>
<tr>
<td>B/ B1.sub B2.sub B3.sub (B job files)</td>
</tr>
<tr>
<td>C/ C.sub (C job files)</td>
</tr>
</tbody>
</table>
PRE and POST scripts run on the submit server, as part of the node

my.dag

JOB A A.sub
SCRIPT POST A sort.sh
JOB B1 B1.sub
JOB B2 B2.sub
JOB B3 B3.sub
JOB C C.sub
SCRIPT PRE C tar_it.sh
PARENT A CHILD B1 B2 B3
PARENT B1 B2 B3 CHILD C

• Use sparingly for light work; otherwise include work in submitted jobs
RETRY failed nodes to overcome transient errors

- Retry a node up to $N$ times if it fails (the job exit code is non-zero):
  
  **RETRY node_name N**

  Example:

  ```
  JOB A A.sub
  RETRY A 5
  JOB B B.sub
  PARENT A CHILD B
  ```

- See also: retry except for a particular exit code (UNLESS–EXIT)

- **Note:** $\text{max\_retries}$ in the submit file are preferable for simple cases
Modular Organization and Control of DAG Components
Repeating DAG Components!!

https://confluence.pegasus.isi.edu/display/pegasus/LIGO+IHOPE
Submit File Templates via VARS

- **VARS** line defines node-specific values that are passed into submit file variables
  
  ```
  VARS node_name var1="value" [var2="value"]
  ```

- Allows a single submit file shared by all B jobs, rather than one submit file for each JOB.

```dagscript
my.dag

JOB B1 B.sub
VARS B1 data="B1" opt="10"

JOB B2 B.sub
VARS B2 data="B2" opt="12"

JOB B3 B.sub
VARS B3 data="B3" opt="14"
```

```bash
B.sub

... InitialDir = $(data)
arguments = $(data).csv $(opt)
...
queue
```
SPLICE subsets of the DAG to simplify lengthy DAG files

my.dag

<table>
<thead>
<tr>
<th>JOB A</th>
<th>A.sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPLICE B</td>
<td>B.spl</td>
</tr>
<tr>
<td>JOB C</td>
<td>C.sub</td>
</tr>
<tr>
<td>PARENT A</td>
<td>CHILD B</td>
</tr>
<tr>
<td>PARENT B</td>
<td>CHILD C</td>
</tr>
</tbody>
</table>

B.spl

<table>
<thead>
<tr>
<th>JOB B1</th>
<th>B1.sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOB B2</td>
<td>B2.sub</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>JOB BN</td>
<td>BN.sub</td>
</tr>
</tbody>
</table>
What if some DAG components can’t be known ahead of time?

B1  B2  B3  ...  BN

A

C

e.g. If the value of N can only be determined as part of the work of the prior node (A) ...

HTCondor Week 2020
A SUBDAG within a DAG

my.dag

JOB  A    A.sub
SUBDAG EXTERNAL  B    B.dag
JOB  C    C.sub
PARENT  A    CHILD  B
PARENT  B    CHILD  C

B.dag (written by A)

JOB  B1   B1.sub
JOB  B2   B2.sub
...
JOB  BN   BN.sub

A SUBDAG is not submitted (so contents do not have to exist) until prior nodes in the outer DAG have completed.
Use a SUBDAG to achieve Cyclic Components within a DAG

- POST script determines whether another iteration is necessary; if so, exits non-zero
- RETRY applies to entire SUBDAG, which may include multiple, sequential nodes

```
my.dag

JOB A A.sub
SUBDAG EXTERNAL B B.dag
SCRIPT POST B iterateB.sh
RETRY B 100
JOB C C.sub
PARENT A CHILD B
PARENT B CHILD C
```
More at the end of this presentation and in the HTCondor Manual!!!

Covered in Later Slides

- Why Create a Workflow?
- Describing workflows as *directed acyclic graphs* (DAGs)
- Workflow execution via DAGMan (DAG Manager)
- Node-level options in a DAG (cont...)
- Modular organization of DAG components (...)
- DAG-level control (...)
- Additional DAGMan Features
QUESTIONS?

htcondor-users@cs.wisc.edu
lmichael@wisc.edu
Beyond the Basic DAG: Node-level Modifiers
RETRY applies to whole node, including PRE/POST scripts

- PRE and POST scripts are included in retries
- RETRY of a node with a POST script uses the exit code from the POST script (not from the job)
  - POST script can do more to determine node success (or need for iteration)

Example:

```bash
SCRIPT PRE A download.sh
JOB A A.sub
SCRIPT POST A checkA.sh
RETRY A 5
```
SCRIPT Arguments and Argument Variables

$JOB: node name

$JOBID: cluster.proc

$RETURN: exit code of the node

$PRE_SCRIPT_RETURN: exit code of PRE script

$RETRY: current retry (‘iteration’) count

(more variables described in the manual)
Other Node-Level Controls

• Set the **PRIORITY** of JOB nodes with:

  ```
  PRIORITY node_name priority_value
  ```

• Use a **PRE_SKIP** to skip a node and mark it as successful, if the PRE script exits with a specific exit code:

  ```
  PRE_SKIP node_name exit_code
  ```
Modular Organization and Control of DAG Components
Use nested SPLICEs with DIR for repeating workflow components

**my.dag**

```plaintext
JOB A A.sub DIR A
SPLICE B B.spl DIR B
JOB C C.sub DIR C
PARENT A CHILD B
PARENT B CHILD C
```

**B.spl**

```plaintext
SPLICE B1 ../inner.spl DIR B1
SPLICE B2 ../inner.spl DIR B2
...
SPLICE BN ../inner.spl DIR BN
```

**inner.spl**

```plaintext
JOB 1 ../1.sub
JOB 2 ../2.sub
PARENT 1 CHILD 2
```
Use nested SPLICEs with DIR for repeating workflow components

**my.dag**

```plaintext
JOB A A.sub DIR A
SPLICE B B.spl DIR B
JOB C C.sub DIR C
PARENT A CHILD B
PARENT B CHILD C
```

**B.spl**

```plaintext
SPLICE B1 ../inner.spl DIR B1
SPLICE B2 ../inner.spl DIR B2
...
SPLICE BN ../inner.spl DIR BN
```

**inner.spl**

```plaintext
JOB 1 ../1.sub
JOB 2 ../2.sub
PARENT 1 CHILD 2
```

**my.dag**

```plaintext
(dag_dir)/

A/ A.sub (A job files)
B/ B.spl
  /inner.spl
    1.sub 2.sub
    B1/ (1-2 job files)
    B2/ (1-2 job files)
    ...
    BN/ (1-2 job files)
C/ C.sub (C job files)
```
More on SPLICE Behavior

• **HTCondor takes in a DAG and its SPLICEEs as a single, large DAG file.**
  – SPLICEEs simply allow the user to simplify and modularize the DAG expression using separate files
  – A single DAGMan job is queued with single set of status files.

• Great for gradually testing and building up a large DAG (since a SPLICE file can be submitted by itself, without its outer DAG).

• SPLICE lines are not treated like nodes.
  – no PRE/POST scripts or RETRIES
More on SUBDAG Behavior

• Each SUBDAG EXTERNAL is a DAGMan job running in the queue, and too many can overwhelm the queue.
  – WARNING: SUBDAGs should only be used (rather than SPLICES) when absolutely necessary!

• SUBDAGs are nodes (can have PRE/POST scripts, retries, etc.)
Other Modular Controls

• Append **NOOP** to a JOB definition so that its JOB process isn’t run by DAGMan
  – Test DAG structure without running jobs (node-level)
  – Simplify combinatorial PARENT-CHILD statements (modular)

• Communicate DAG features separately with **INCLUDE**
  – e.g. separate files for JOB nodes and for VARS definitions, as part of the same DAG

• Define a **CATEGORY** of JOB nodes to throttle only a specific subset
DAG-level Control
Throttle job nodes of large DAGs via DAG-level configuration

• If a DAG has many (thousands or more) jobs, submit server and queue performance can be assured by limiting:
  – Number of jobs in the queue
  – Number of jobs idle (waiting to run)
  – Number of PRE or POST scripts running

• Limits can be specified in a DAG-specific CONFIG file (recommended) or as arguments to condor_submit_dag
DAG-specific throttling via a CONFIG file

my.dag

JOB A A.sub
SPLICE B B.dag
JOB C C.sub
PARENT A CHILD B
PARENT B CHILD C
CONFIG my.dag.config

my.dag.config

DAGMAN_MAX_JOBS_SUBMITTED = 5000
DAGMAN_MAX_JOBS_IDLE = 1000
DAGMAN_MAX_PRE_SCRIPTS = 4
DAGMAN_MAX_POST_SCRIPTS = 4
Removal of a DAG results in a rescue file

(dag_dir)/

A.sub  B1.sub  B2.sub  B3.sub  C.sub  (other job files)
my.dag       my.dag.condor.sub  my.dag.dagman.log
my.dag.dagman.out  my.dag.lib.err  my.dag.lib.out
my.dag.metrics  my.dag.nodes.log  my.dag.rescue001

- Named `dag_file.rescue001`
  - increments if more rescue DAG files are created
- Records which NODES have completed successfully
  - does not contain the actual DAG structure
Rescue Files For Resuming a Failed DAG

• A rescue file is created any time a DAG is removed from the queue by the user (condor_rmr) or automatically:
  – a node fails, and after DAGMan advances through any other possible nodes
  – the DAG is aborted (covered later)
  – the DAG is halted and not unhalted (covered later)

• The rescue file will be used (if it exists) when the original DAG file is resubmitted
  – override: `condor_submit_dag dag_file -f`
Pause (then resume) a DAG by holding it

• Hold the DAGMan job process:
  \texttt{condor\_hold dagman\_jobID}

• Pauses the DAG
  – No new node jobs submitted
  – Queued node jobs continue to run (including SUBDAGs), but no PRE/POST scripts
  – DAG resumes when released
    \texttt{(condor\_release dagman\_jobID)}
Cleanly quit a DAG with a halt file

• Create a file named `DAG_file.halt` in the same directory as the submitted DAG file
• Allows the DAG to complete nodes in-progress
  – No new node jobs submitted
  – Queued node jobs and SUBDAGs (including POST scripts) continue to run, but not PRE scripts
  – After all queued jobs have completed, the DAG creates a rescue DAG file and exits.
• If the DAG hasn’t yet exited and the file is deleted, then the DAG resumes
Other DAG-Level Controls

• Replace the `node_name` with `ALL_NODES` to apply a DAG feature to all nodes of the DAG

• Abort the entire DAG if a specific node exits with a specific exit code:

  \[
  \text{ABORT-DAG-ON node_name exit_code}
  \]

• Define a \textbf{FINAL} node that will always run, even in the event of DAG failure (to clean up, perhaps).

  \[
  \text{FINAL node_name submit_file}
  \]