Goals for this Session

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

- Objective: Submit jobs in a particular order, automatically.
- Especially if: Need to replicate the same workflow multiple times in the future.
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 DAG workflows
DESCRIBING WORKFLOWS WITH DAGMAN
DAGMan in the HTCondor Manual

- 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

- MPI Applications Within HTCondor’s Vanilla Universe

An Example HTC Workflow

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

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

my.dag

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

- Node names will be used by various DAG features to modify their execution by DAGMan.
Basic DAG input file:

**JOB** nodes, **PARENT-CHILD** edges

- Node names and filenames are your choice.
- Node name and submit filename do not have to match.

```plaintext
my.dag

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

(Other job files)

HTCondor Manual: DAGMan Applications > DAG Input File
Endless Workflow Possibilities
DAGs are also useful for non-sequential work

‘bag’ of HTC jobs

B1  B2  B3  ...  BN

disjointed workflows

A  B

C  D  E

F  G  H

I

A  B

C  D  E

F  G  H

I
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
```

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 : mydag.dag.condor.sub
Log of DAGMan debugging messages : mydag.dag.dagman.out
Log of HTCondor library output : mydag.dag.lib.out
Log of HTCondor library error messages : mydag.dag.lib.err
Log of the life of condor_dagman itself : mydag.dag.dagman.log

Submitting job(s).
1 job(s) submitted to cluster 128.
A submitted DAG creates a **DAGMan job** in the queue

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

```
$ 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  -  -   -     0.0
1 jobs; 0 completed, 0 removed, 0 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+00:00:06 R  0   0.3 condor_dagman
1 jobs; 0 completed, 0 removed, 0 idle, 1 running, 0 held, 0 suspended
```
Status files are created at the time of DAG submission

(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></td>
<td></td>
</tr>
</tbody>
</table>

*.condor.sub and *.dagman.log describe the queued DAGMan job process, as for any other jobs

*.dagman.out has DAGMan-specific 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
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

```
$ 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>...

<table>
<thead>
<tr>
<th>ID</th>
<th>OWNER</th>
<th>SUBMITTED</th>
<th>RUN_TIME</th>
<th>ST</th>
<th>PRI</th>
<th>SIZE</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>128.0</td>
<td>alice</td>
<td>4/30 18:08</td>
<td>0+00:20:36</td>
<td>R</td>
<td>0</td>
<td>0.3</td>
<td>condor_dagman</td>
</tr>
<tr>
<td>130.0</td>
<td>alice</td>
<td>4/30 18:18</td>
<td>0+00:00:00</td>
<td>I</td>
<td>0</td>
<td>0.3</td>
<td>B_run.sh</td>
</tr>
<tr>
<td>131.0</td>
<td>alice</td>
<td>4/30 18:18</td>
<td>0+00:00:00</td>
<td>I</td>
<td>0</td>
<td>0.3</td>
<td>B_run.sh</td>
</tr>
<tr>
<td>132.0</td>
<td>alice</td>
<td>4/30 18:18</td>
<td>0+00:00:00</td>
<td>I</td>
<td>0</td>
<td>0.3</td>
<td>B_run.sh</td>
</tr>
</tbody>
</table>
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
```
DAG Completion

*(dag_dir)/

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.sub</td>
<td></td>
</tr>
<tr>
<td>B1.sub</td>
<td></td>
</tr>
<tr>
<td>B2.sub</td>
<td></td>
</tr>
<tr>
<td>B3.sub</td>
<td></td>
</tr>
<tr>
<td>C.sub</td>
<td></td>
</tr>
<tr>
<td>(other job files)</td>
<td></td>
</tr>
<tr>
<td>my.dag</td>
<td></td>
</tr>
<tr>
<td>my.dag.condor.sub</td>
<td></td>
</tr>
<tr>
<td>my.dag.dagman.log</td>
<td></td>
</tr>
<tr>
<td>my.dag.dagman.out</td>
<td></td>
</tr>
<tr>
<td>my.dag.lib.err</td>
<td></td>
</tr>
<tr>
<td>my.dag.lib.out</td>
<td></td>
</tr>
<tr>
<td>my.dag.nodes.log</td>
<td></td>
</tr>
<tr>
<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 (look to first for errors)
STOPPING, RESTARTING, AND TROUBLESHOOTING
Removing a DAG from the queue

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

- Creates a rescue file so that only incomplete or unsuccessful NODES are repeated upon resubmission

```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 8:08 4 _ 1 6 129.0...133.0
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
```
Removal of a DAG creates a *rescue file*

\[(\text{dag\_dir})/\]

\begin{tabular}{lllll}
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 \\
\end{tabular}

- 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

DAGMan > DAG Monitoring and DAG Removal
DAGMan > The Rescue DAG
Rescue Files
For Resuming a Failed DAG

• A rescue file is created when:
  – a node fails, and after DAGMan advances through any other possible nodes
  – the DAG is removed from the queue (or aborted, see manual)
  – the DAG is halted and not unhalted (see manual)

• Resubmission uses the rescue file (if it exists) when the original DAG file is resubmitted
  – override: condor_submit_dag dag_file -f
Node Failures
Result in DAG Failure

- 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 queued process in a JOB node results in failure of the **entire node** and immediate removal of all other processes in the node.
  - RETRY of a JOB node retries 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 (remember the automatic rescue DAG file!)

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

<table>
<thead>
<tr>
<th>ID</th>
<th>OWNER</th>
<th>SUBMITTED</th>
<th>RUN_TIME</th>
<th>ST</th>
<th>PRI</th>
<th>SIZE</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>128.0</td>
<td>alice</td>
<td>4/30 18:08</td>
<td>0+00:20:36</td>
<td>R</td>
<td>0</td>
<td>0.3</td>
<td>condor_dagman</td>
</tr>
<tr>
<td>130.0</td>
<td>alice</td>
<td>4/30 18:18</td>
<td>0+00:00:00</td>
<td>H</td>
<td>0</td>
<td>0.3</td>
<td>B_run.sh</td>
</tr>
<tr>
<td>131.0</td>
<td>alice</td>
<td>4/30 18:18</td>
<td>0+00:00:00</td>
<td>H</td>
<td>0</td>
<td>0.3</td>
<td>B_run.sh</td>
</tr>
<tr>
<td>132.0</td>
<td>alice</td>
<td>4/30 18:18</td>
<td>0+00:00:00</td>
<td>H</td>
<td>0</td>
<td>0.3</td>
<td>B_run.sh</td>
</tr>
</tbody>
</table>

4 jobs; 0 completed, 0 removed, 0 idle, 1 running, 3 held, 0 suspended
```
BEYOND THE BASIC DAG: NODE-LEVEL MODIFIERS
### Default File Organization

- **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)

- **What if you want to organize files into other directories?**
Node-specific File Organization with \textit{DIR}

- \textit{DIR} sets the submission directory of the node

\begin{itemize}
  \item \texttt{my.dag}
  \item (\texttt{dag_dir})/
\end{itemize}

\begin{verbatim}
JOB A A.sub  DIR A
JOB B1 B1.sub  DIR B
JOB B2 B2.sub  DIR B
JOB B3 B3.sub  DIR B
JOB C C.sub  DIR C
PARENT A  CHILD B1 B2 B3
PARENT B1 B2 B3  CHILD C
\end{verbatim}

\begin{verbatim}
my.dag
A/  A.sub  (A job files)
B/  B1.sub  B2.sub  B3.sub  (B job files)
C/  C.sub  (C job files)
\end{verbatim}
**PRE and POST scripts** run on the submit server, as part of the node.

```bash
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 lightweight work; otherwise include work in node jobs.
RETRY failed nodes to overcome transient errors

- Retry a node up to $N$ times if the 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
  ```

- **Note:** Unnecessary for nodes (jobs) that can use `max_retries` in the submit file.

- See also: retry except for a particular exit code (`UNLESS-EXIT`), or retry scripts (`DEFER`).
**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, perhaps by examining JOB output

Example:
```
SCRIPT PRE A download.sh
JOB A A.sub
SCRIPT POST A checkA.sh
RETRY A 5
```
MODULAR ORGANIZATION OF DAG COMPONENTS
Submit File Templates via VARS

- **VARS** line defines node-specific values that are passed into submit file variables
  
  \[
  \text{VARS node\_name var1=\textit{value} [var2=\textit{value}]}\]

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

```plaintext
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"

B.sub

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

```
my.dag
JOB A A.sub
SPLICE B B.spl
JOB C C.sub
PARENT A CHILD B
PARENT B CHILD C

B.spl
JOB B1 B1.sub
JOB B2 B2.sub
...
JOB BN BN.sub
```
Repeating DAG Components!!

https://confluence.pegasus.isi.edu/display/pegasus/LIGO+IHOPE
What if some DAG components can’t be known at submit time?

If $N$ can only be determined as part of the work of $A$ …
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
Use a **SUBDAG** to achieve a Cyclic Component 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

```plaintext
my.dag

JOB A A.sub
SUBDAG EXTERNAL B B.dag
SCRIPT POST B iterateB.sh
RETRY B 1000
JOB C C.sub
PARENT A CHILD B
PARENT B CHILD C
```
More in the **HTCondor Manual** and the **HTCondor Week DAGMan Tutorial!!!**
YOUR TURN!
DAGMan Exercises!

• Essential: Exercises 1-4
• Ask questions! ‘See you in Slack!'