HTCondor and Workflows: An Introduction

HTCondor Week 2013

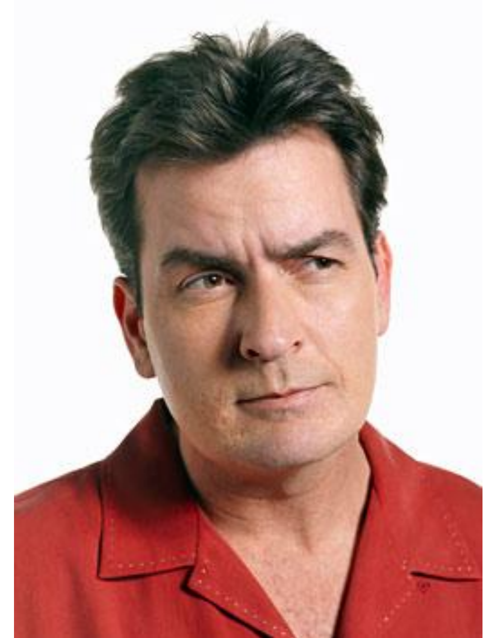
Kent Wenger

Outline

- Introduction/motivation
- Basic DAG concepts
- Running and monitoring a DAG
- Configuration
- Rescue DAGs
- Advanced DAGMan features

My jobs have dependencies…

Can HTCondor help solve my dependency problems?



Yes!

Workflows are the answer

What are workflows?

- General: a sequence of connected steps
- Our case
  - Steps are HTCondor jobs
  - Sequence defined at higher level
  - Controlled by a Workflow Management System (WMS), *not just a script*

Kent's summer 2012 workflow

Get permission from Miron to take leave

↓

Fly to Turkey

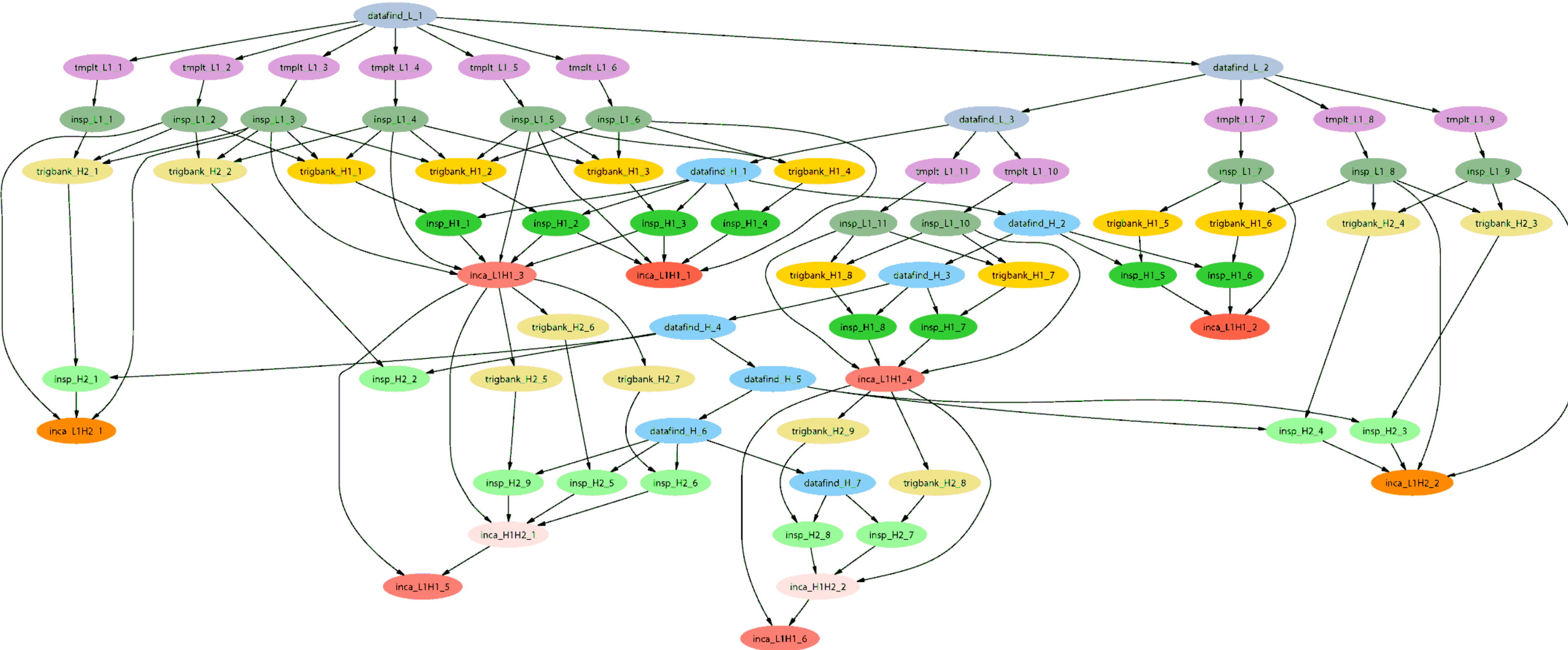
↓

Bike to Croatia

↓

Fly home

LIGO inspiral search application



Inspiral workflow application is the work of Duncan Brown, Caltech, Scott Koranda, UW Milwaukee, and the LSC Inspiral group

Workflows – launch and forget

- A workflow can take days, weeks or even months
- Automates tasks user *could* perform manually…
  - But WMS takes care of automatically
- Enforces inter-job dependencies
- Includes features such as retries in the case of failures – avoids the need for user intervention
- The workflow itself can include error checking
- The result: one user action can utilize many resources while maintaining complex job inter-dependencies and data flows

How big?

- We have users running 500k-job workflows in production
- Depends on resources on submit machine (memory, max. open files)
- "Tricks" can decrease resource requirements (talk to me or Nathan Panike)

Workflow tools

- DAGMan: HTCondor's workflow tool
- Pegasus: a layer on top of DAGMan that is grid-aware and data-aware
- Makeflow: not covered in this talk
- Others…
- This talk will focus mainly on DAGMan

Pegasus WMS

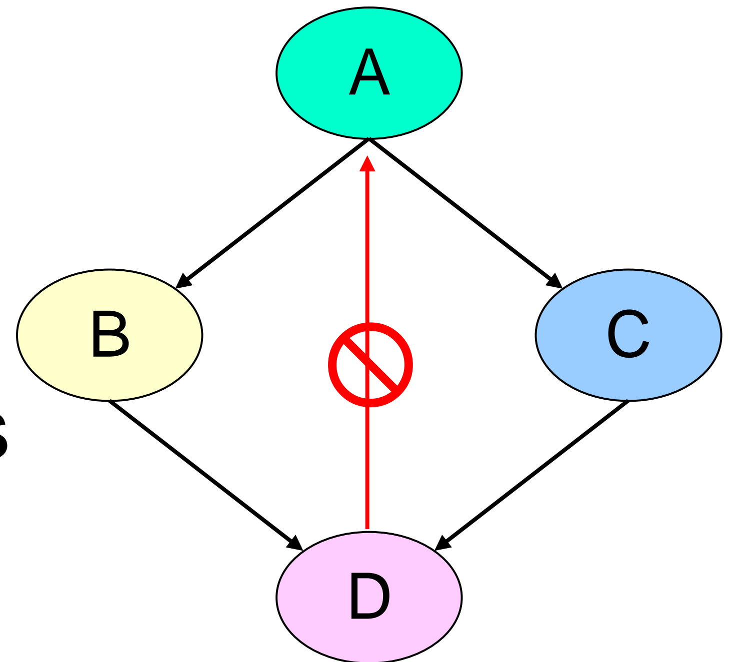
- A higher level on top of DAGMan
- User creates an abstract workflow
- Pegasus maps abstract workflow to executable workflow
- DAGMan runs executable workflow
- Doesn’t need full Condor (schedd only)
- A talk tomorrow with more details

Outline

- Introduction/motivation
- Basic DAG concepts
- Running and monitoring a DAG
- Configuration
- Rescue DAGs
- Advanced DAGMan features

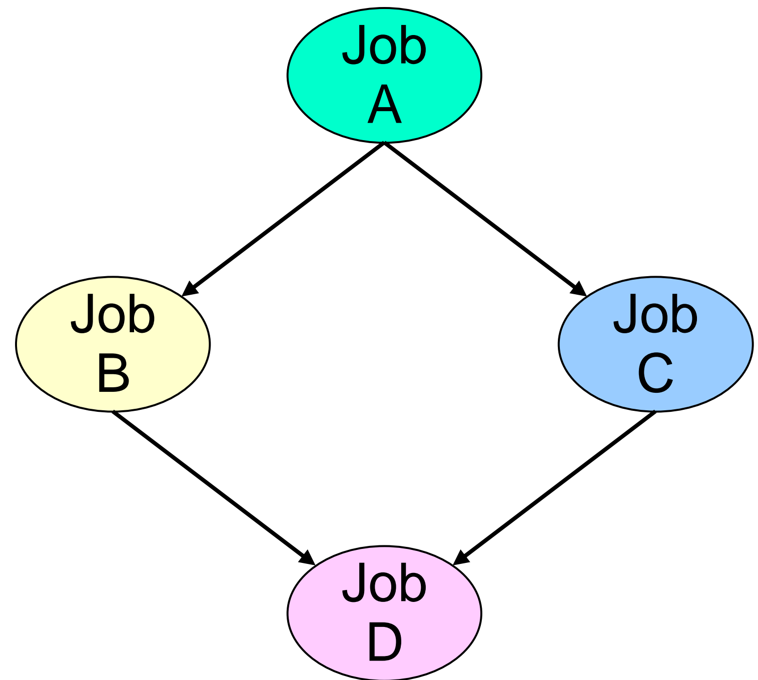
DAG definitions

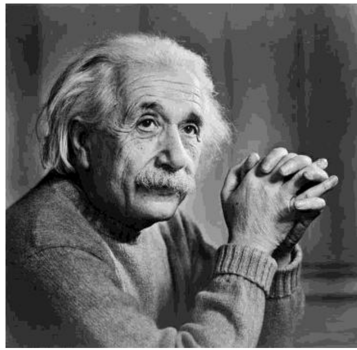
- DAGs have one or more nodes (or vertices)
- Dependencies are represented by arcs (or edges). These are arrows that go from parent to child)
- No cycles!



HTCondor and DAGs

- Each node represents a HTCondor job (or cluster)
- Dependencies define possible orders of job execution





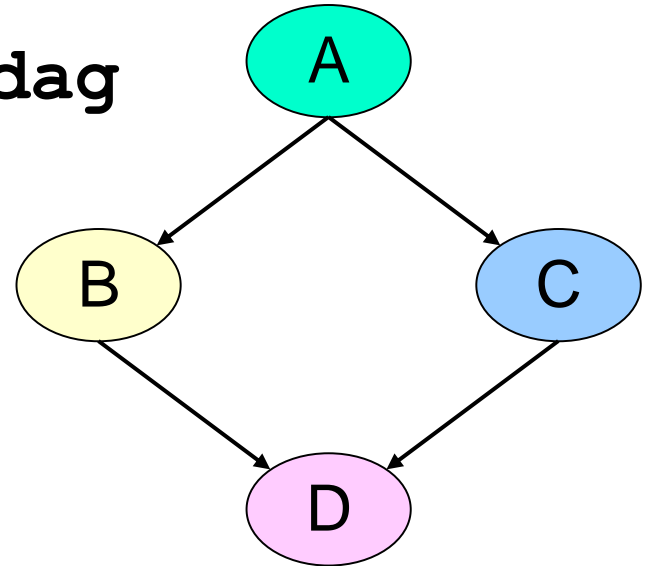
Albert learns DAGMan

- **Directed Acyclic Graph Manager**
- DAGMan allows Albert to specify the dependencies between his HTCondor jobs, so DAGMan manages the jobs automatically
- Dependency example: do not run job B until job A has completed successfully

Defining a DAG to DAGMan

A DAG input file defines a DAG:

```
# file name: diamond.dag
Job A a.submit
Job B b.submit
Job C c.submit
Job D d.submit
Parent A Child B C
Parent B C Child D
```



Submit description files

For node B:

```
# file name:
#     b.submit
universe     = vanilla
executable = B
input        = B.in
output       = B.out
error        = B.err
log          = B.log
queue
```

For node C:

```
# file name:
#     c.submit
universe     = standard
executable = C
input        = C.in
output       = C.out
error        = C.err
log          = C.log
queue
```

Jobs/clusters

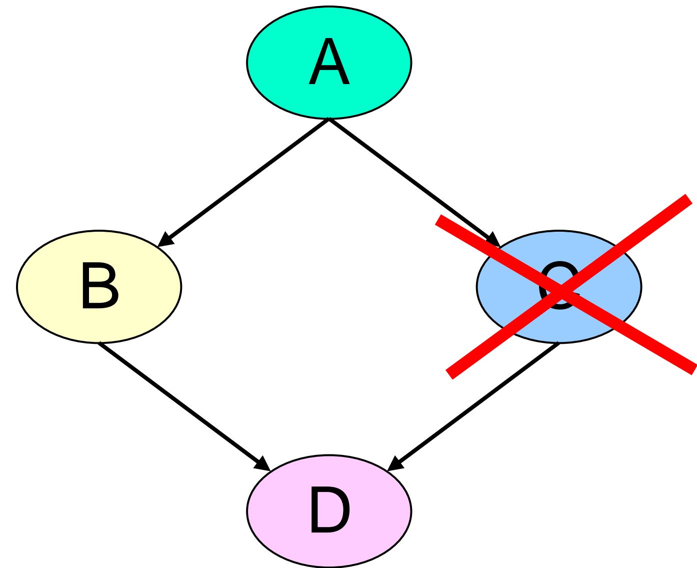
- Submit description files used in a DAG can create multiple jobs,
  but they must all be in a single cluster.
- The failure of any job means the entire cluster fails. Other jobs in the cluster are removed.

Node success or failure

- A node either succeeds or fails
- Based on the return value of the job(s)

  0 ⇨ success

  not 0 ⇨ failure
- This example: C fails
- Failed nodes block execution; DAG fails



PRE and POST scripts

- Optionally associated with nodes
- Run before (PRE) and after (POST) the actual HTCondor node job
- More details later…

Outline

- Introduction/motivation
- Basic DAG concepts
- Running and monitoring a DAG
- Configuration
- Rescue DAGs
- Advanced DAGMan features

Submitting the DAG to HTCondor

- To submit the entire DAG, run

  `condor_submit_dag` ***`DagFile`***

- condor_submit_dag creates a submit description file for DAGMan, and DAGMan itself is submitted as an HTCondor job (in the scheduler universe)
- **`-f(orce)`** option forces overwriting of existing files

Controlling running DAGs

- **`condor_rm` *`dagman_id`***
  - Removes *entire* workflow
    - Removes all queued node jobs
    - Kills PRE/POST scripts
  - Creates rescue DAG (more on this later)
  - Work done by partially-completed node jobs is lost

Controlling running DAGs (cont)

› `condor_hold` and `condor_release`
  - "Pauses" the DAG
  - Node jobs continue when DAG is held
  - No new node jobs submitted
  - DAGMan "catches up" when released

Controlling running DAGS: the halt file

- “Pauses” the DAG
- Create a file named ***DagFile.halt*** in the same directory as your DAG file.
- Remove halt file to resume normal operation
- Should be noticed w/in 5 sec (**DAGMAN_USER_LOG_SCAN_INTERVAL**)
- New in HTCondor version 7.7.5.

The halt file (cont)

- Jobs that are running will continue to run.
- POST scripts are run as jobs finish.
- No new jobs will be submitted and no PRE scripts will be run.
- When all submitted jobs complete, DAGMan creates a rescue DAG and exits.

- Good if load on submit machine is very high
- Avoids hold/release problem of possible duplicate PRE/POST script instances

Monitoring a DAG: condor_q -dag

- The `-dag` option associates DAG node jobs with the parent DAGMan job.
- Shows current workflow state
- New in 7.7.5: Shows nested DAGs properly.

condor_q –dag example

```
> condor_q -dag
-- Submitter: nwp@llunet.cs.wisc.edu : <128.105.14.28:51264> : llunet.cs.wisc.edu
    ID      OWNER/NODENAME   SUBMITTED     RUN_TIME ST PRI SIZE CMD
    392.0   nwp             4/25 13:27   0+00:00:50 R  0   1.7  condor_dagman -f –
    393.0    |-1            4/25 13:27   0+00:00:23 R  0   0.0  1281.sh 393
    395.0    |-0            4/25 13:27   0+00:00:30 R  0   1.7  condor_dagman -f –
    399.0     |-A           4/25 13:28   0+00:00:03 R  0   0.0  1281.sh 399
4 jobs; 0 completed, 0 removed, 0 idle, 4 running, 0 held, 0 suspended
```

Status in DAGMan's ClassAd

```
> condor_q -l 59 | grep DAG_
DAG_Status = 0
DAG_InRecovery = 0
DAG_NodesUnready = 1
DAG_NodesReady = 4
DAG_NodesPrerun = 2
DAG_NodesQueued = 1
DAG_NodesPostrun = 1
DAG_NodesDone = 3
DAG_NodesFailed = 0
DAG_NodesTotal = 12
```

- Sub-DAGs count as one node
- New in 7.9.5

Dagman.out file

- ***DagFile.dagman.out***
- Logs detailed workflow history
- Mostly for debugging
- Verbosity controlled by the **DAGMAN_VERBOSITY** configuration macro and **-debug *n*** on the **condor_submit_dag** command line
  - 0: least verbose
  - 7: most verbose
- Don't decrease verbosity unless you really have to.

Dagman.out contents

```
...
04/17/11 13:11:26 Submitting Condor Node A job(s)...
04/17/11 13:11:26 submitting: condor_submit -a dag_node_name' '=' 'A -a +DAGManJobId' '='
    '180223 -a DAGManJobId' '=' '180223 -a submit_event_notes' '=' 'DAG' 'Node:' 'A -a
    +DAGParentNodeNames' '=' '"" dag_files/A2.submit
04/17/11 13:11:27 From submit: Submitting job(s).
04/17/11 13:11:27 From submit: 1 job(s) submitted to cluster 180224.
04/17/11 13:11:27       assigned Condor ID (180224.0.0)
04/17/11 13:11:27 Just submitted 1 job this cycle...
04/17/11 13:11:27 Currently monitoring 1 Condor log file(s)
04/17/11 13:11:27 Event: ULOG_SUBMIT for Condor Node A (180224.0.0)
04/17/11 13:11:27 Number of idle job procs: 1
04/17/11 13:11:27 Of 4 nodes total:
04/17/11 13:11:27  Done     Pre   Queued    Post   Ready   Un-Ready   Failed
04/17/11 13:11:27   ===     ===      ===     ===     ===        ===      ===
04/17/11 13:11:27     0       0        1       0       0          3        0
04/17/11 13:11:27 0 job proc(s) currently held
...
```

This is a small excerpt of the dagman.out file.

Node status file

- Shows a snapshot of workflow state
  - Overwritten as the workflow runs
  - Updated atomically
- In the DAG input file:
  NODE_STATUS_FILE *statusFileName* [*minimumUpdateTime*]
- Not enabled by default

Node status file contents

```
BEGIN 1302885255 (Fri Apr 15 11:34:15 2011)
Status of nodes of DAG(s): job_dagman_node_status.dag

JOB A STATUS_DONE       ()
JOB B1 STATUS_SUBMITTED (not_idle)
JOB B2 STATUS_SUBMITTED (idle)
...
Nodes total: 12
Nodes done: 8
...
DAG status: STATUS_SUBMITTED ()
Next scheduled update: 1302885258 (Fri Apr 15
  11:34:18 2011)
END 1302885255 (Fri Apr 15 11:34:15 2011)
```

Jobstate.log file

- Shows workflow history
- Meant to be machine-readable (for Pegasus)
- Basically a subset of the **`dagman.out`** file
- In the DAG input file:
  `JOBSTATE_LOG` ***`JobstateLogFileName`***
- Not enabled by default

Jobstate.log contents

```
1302884424 INTERNAL *** DAGMAN_STARTED 48.0
  ***
1302884436 NodeA PRE_SCRIPT_STARTED - local -
  1
1302884436 NodeA PRE_SCRIPT_SUCCESS - local -
  1
1302884438 NodeA SUBMIT 49.0 local - 1
1302884438 NodeA SUBMIT 49.1 local - 1
1302884438 NodeA EXECUTE 49.0 local - 1
1302884438 NodeA EXECUTE 49.1 local - 1
...
```

Dot file

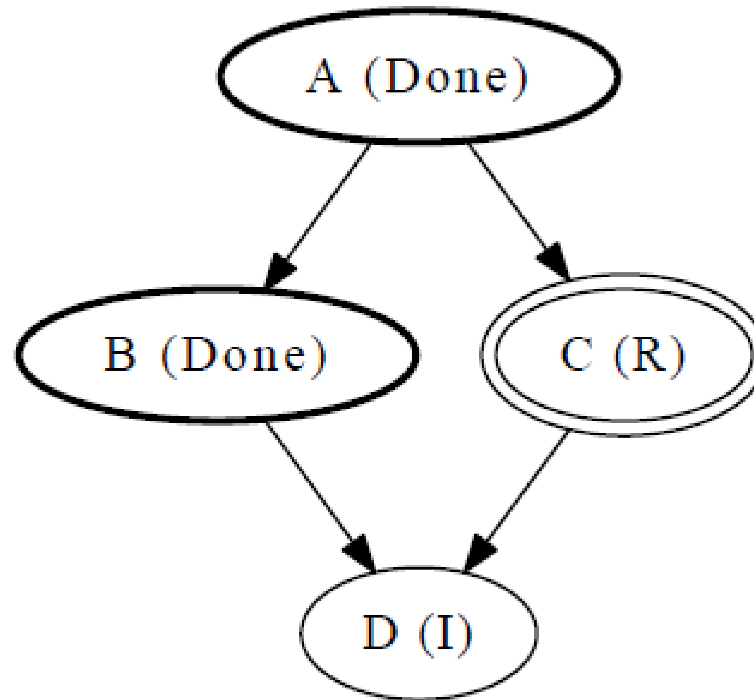
- Shows a snapshot of workflow state
- Updated atomically
- For input to the dot visualization tool
- In the DAG input file:

  DOT *DotFile* [UPDATE] [DONT-OVERWRITE]

- To create an image

  dot -Tps *DotFile* -o *PostScriptFile*

Dot file example



Outline

› Introduction/motivation
› Basic DAG concepts
› Running and monitoring a DAG
› Configuration
› Rescue DAGs
› Advanced DAGMan features

DAGMan configuration

- A few dozen DAGMan-specific configuration macros (see the manual…)
- From lowest to highest precedence
  - HTCondor configuration files
  - User's environment variables:
    - `_CONDOR_`*`macroname`*
  - DAG-specific configuration file (preferable)
  - **`condor_submit_dag`** command line

Per-DAG configuration

- In DAG input file:
  `CONFIG` ***`ConfigFileName`***
  or
  `condor_submit_dag -config` ***`ConfigFileName`*** **`...`**
- Generally prefer **`CONFIG`** in DAG file over **`condor_submit_dag -config`** or individual arguments
- Specifying more than one configuration file is an error.

Per-DAG configuration (cont)

- Configuration entries not related to DAGMan are ignored
- Syntax like any other HTCondor config file

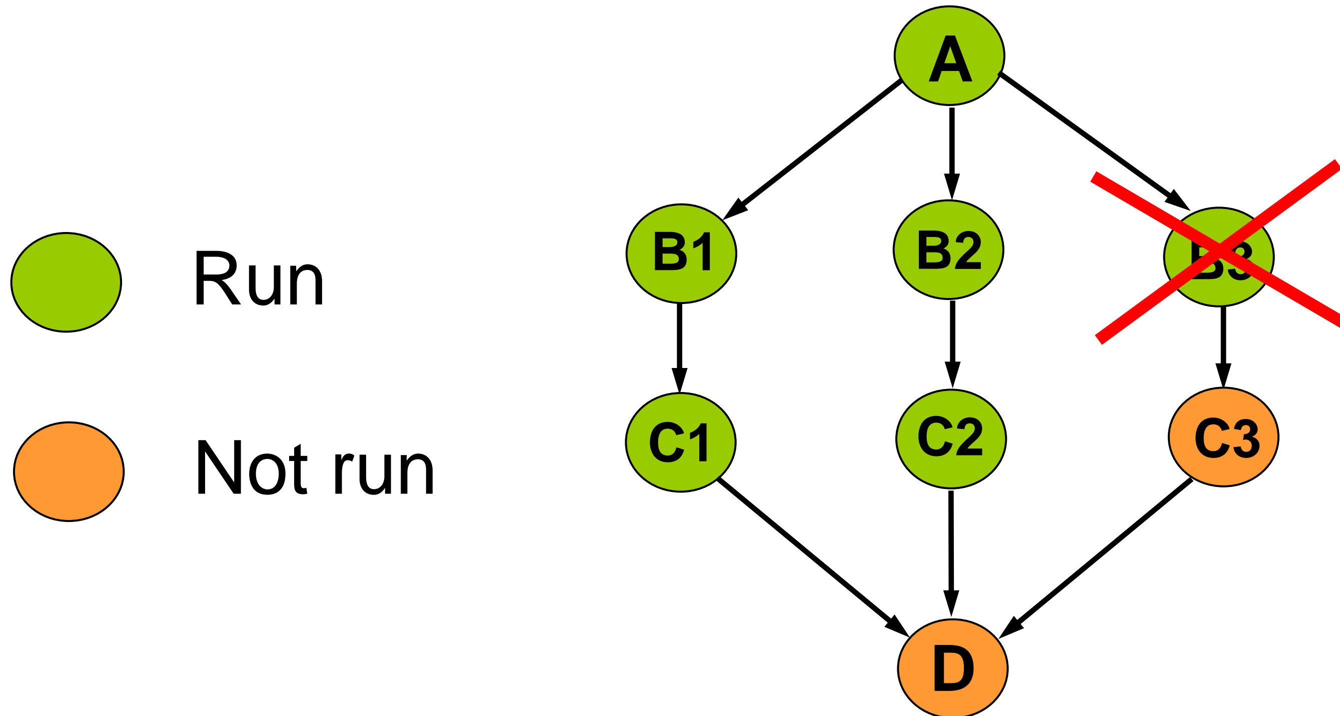
```
# file name: bar.dag
CONFIG bar.config

# file name: bar.config
DAGMAN_ALWAYS_RUN_POST = False
DAGMAN_MAX_SUBMIT_ATTEMPTS = 2
```

Outline

- Introduction/motivation
- Basic DAG concepts
- Running and monitoring a DAG
- Configuration
- Rescue DAGs
- Advanced DAGMan features

Rescue DAGs



Rescue DAGs (cont)

- Save the state of a partially-completed DAG
- Created when a node fails or the **`condor_dagman`** job is removed with **`condor_rm`**
  - DAGMan makes as much progress as possible in the face of failed nodes
- DAGMan immediately exits after writing a rescue DAG file
- Automatically run when you re-run the original DAG (unless **`-f`** is passed to **`condor_submit_dag`**)

Rescue DAGs (cont)

- New in HTCondor version 7.7.2, the Rescue DAG file, by default, is only a partial DAG file
- A partial Rescue DAG file contains only information about which nodes are done, and the number of retries remaining for nodes with retries.
- Does not contain information such as the actual DAG structure and the specification of the submit file for each node job.
- Partial Rescue DAGs are automatically parsed in combination with the original DAG file, which contains information such as the DAG structure.

Rescue DAGs (cont)

- If you change something in the original DAG file, such as changing the submit file for a node job, that change will take effect when running a partial Rescue DAG.

Rescue DAG naming

- ***`DagFile.rescue001`***, ***`DagFile.rescue002`***, etc.
- Up to 100 by default (last is overwritten once you hit the limit)
- Newest is run automatically when you re-submit the original ***`DagFile`***
- **`condor_submit_dag -dorescuefrom`** ***`number`*** to run specific rescue DAG
  - Newer rescue DAGs are renamed

Outline

- Introduction/motivation
- Basic DAG concepts
- Running and monitoring a DAG
- Configuration
- Rescue DAGs
- Advanced DAGMan features

PRE and POST scripts

- DAGMan allows PRE and/or POST scripts
  - Not necessarily a script: any executable
  - Run before (PRE) or after (POST) job
  - Scripts run on submit machine (not execute machine)
- In the DAG input file:

```
Job A a.submit
Script PRE A before-script arguments
Script POST A after-script arguments
```

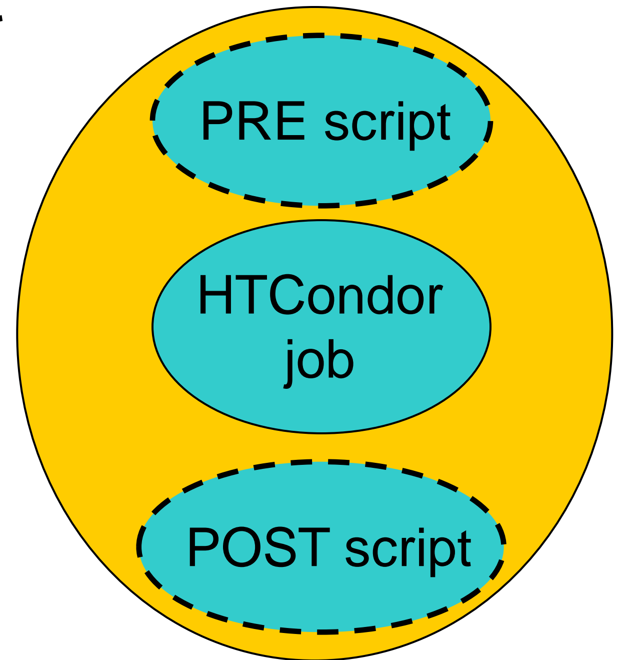
- No spaces in script name or arguments

Why PRE/POST scripts?

- Set up input
- Check output
- Create submit file or sub-DAG (dynamically)
- Probably lots of other reasons…

DAG node with scripts

- PRE script, Job, or POST script determines node success or failure (table in manual gives details)



DAG node with scripts (cont)

- If PRE script fails, job is not run. The POST script is run (new in 7.7.2).
  - Set `DAGMAN_ALWAYS_RUN_POST = False` to get old behavior

Script argument variables

- **$JOB**: node name
- **$JOBID**: Condor ID (*cluster.proc*) (POST only)
- **$RETRY**: current retry
- **$MAX_RETRIES**: max # of retries
- **$RETURN**: exit code of HTCondor/Stork job (POST only)
- **$PRE_SCRIPT_RETURN**: PRE script return value (POST only)
- **$DAG_STATUS**: A number indicating the state of DAGMan. See the manual for details.
- **$FAILED_COUNT**: the number of nodes that have failed in the DAG

DAG node with scripts: PRE_SKIP

- Here is the syntax:

```
JOB A A.cmd
SCRIPT PRE A A.pre
PRE_SKIP A non-zero integer
```

- If the PRE script of A exits with the indicated value, this is normally a failure.
- Instead, the node succeeds immediately, and the node job and POST script are skipped.
- If the PRE script fails with a different value, the node job is skipped, and the POST script runs (as if PRE_SKIP were not specified).

DAG node with scripts: PRE_SKIP (cont)

- When the POST script runs, the **$PRE_SCRIPT_RETURN** variable contains the return value from the PRE script. (See manual for specific cases)
- New in 7.7.2.

NOOP nodes

- Appending the keyword **`NOOP`** causes a job to not be run, without affecting the DAG structure.
- The PRE and POST scripts of NOOP nodes will be run. If this is not desired, comment them out.
- Can be used to test DAG structure

NOOP nodes (ex)

- Here is an example:

```
# file name: diamond.dag
Job A a.submit NOOP
Job B b.submit NOOP
Job C c.submit NOOP
Job D d.submit NOOP
Parent A Child B C
Parent B C Child D
```

- Submitting this to DAGMan will cause DAGMan to exercise the DAG, without actually running node jobs.

Node retries

- For possibly transient errors
- Before a node is marked as failed. . .
  - Retry N times. In the DAG file:

```
Retry C 4
```

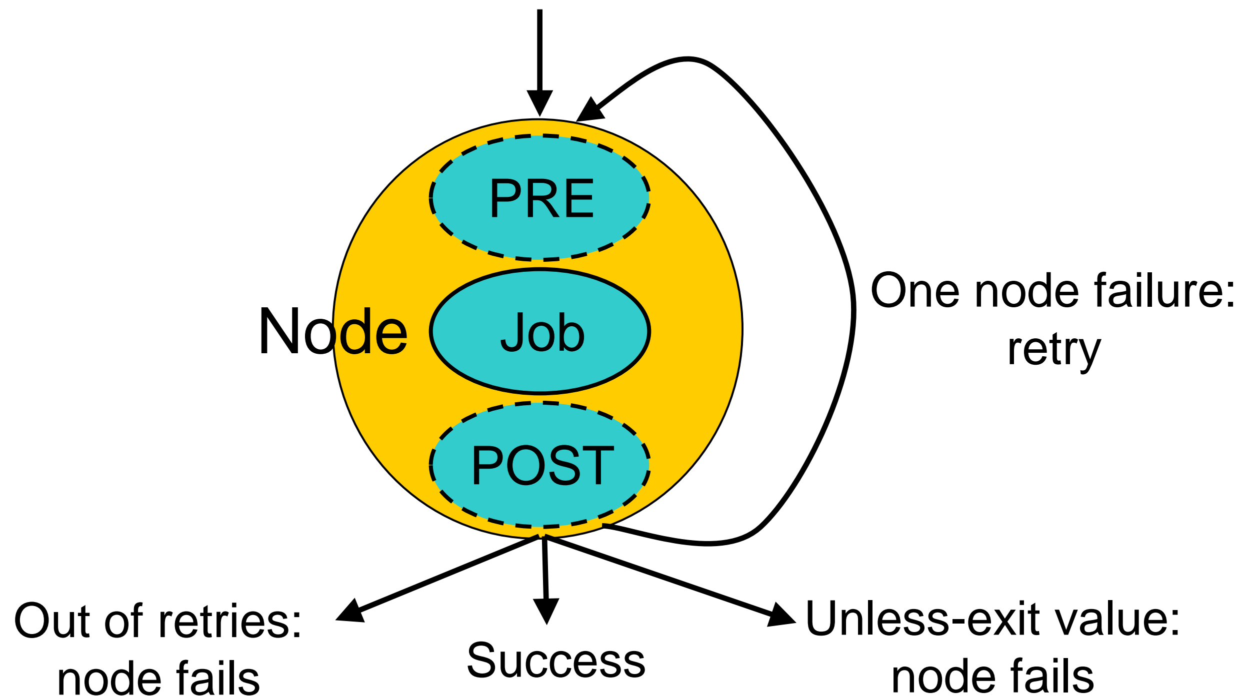
(to retry node **C** four times before calling the node failed)

  - Retry N times, unless a node returns specific exit code. In the DAG file:

```
Retry C 4 UNLESS-EXIT 2
```

Node retries, continued

› Node is retried as a whole



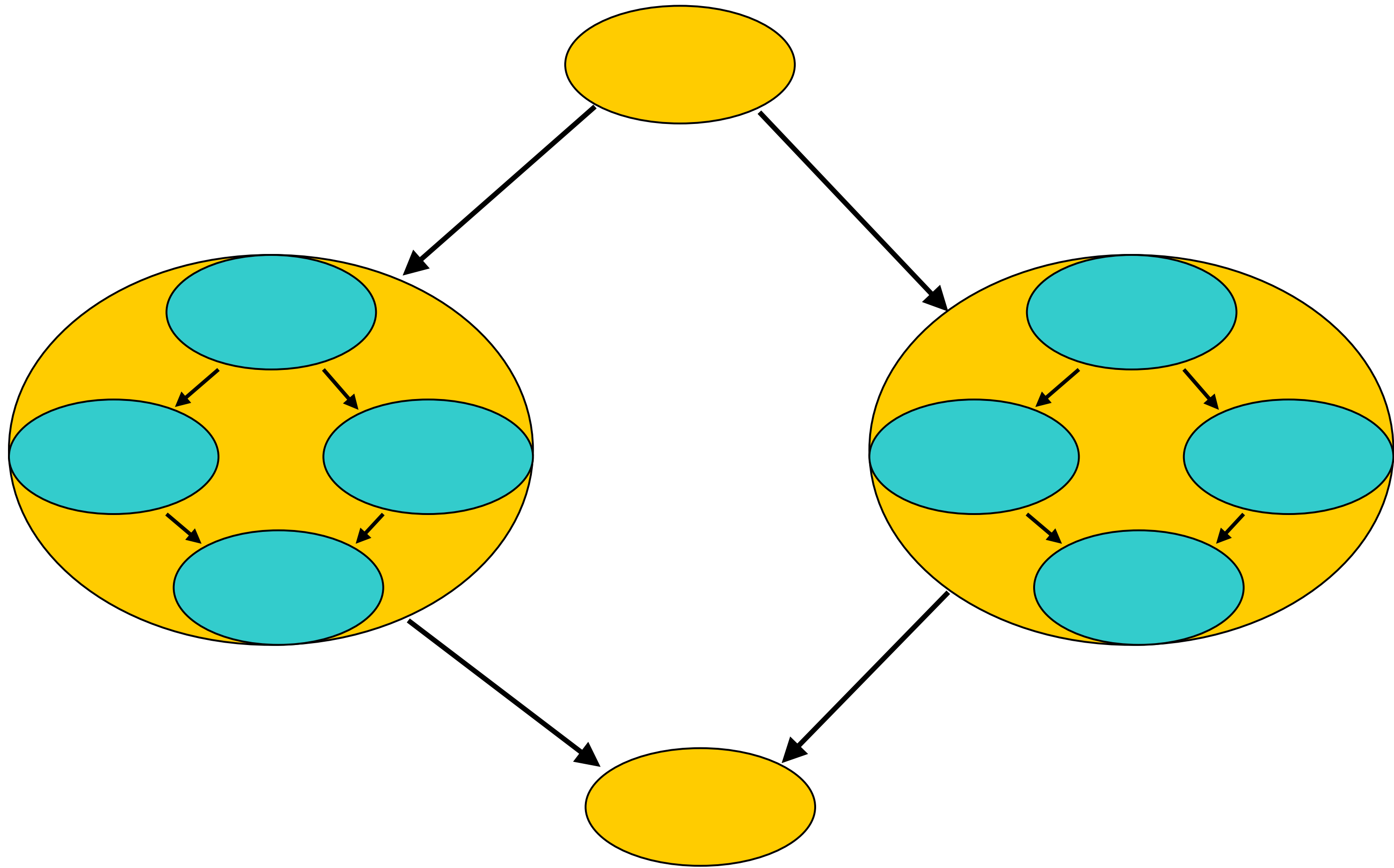
Node variables

- To re-use submit files
- In DAG input file
  VARS *JobName varname*="*string*" [*varname*="*string*"...]
- In submit description file
  $(*varname*)
- **varname** can only contain alphanumeric characters and underscore
- **varname** cannot begin with "**queue**"
- **varname** is not case-sensitive

Node variables (cont)

- Value cannot contain single quotes; double quotes must be escaped
- The variable **$(JOB)** contains the DAG node name of the job.
- More than one VARS line per job is allowed.
- DAGMan warns if a VAR is defined more than once for a job.

Nested DAGs



Nested DAGs (cont)

- Runs the sub-DAG as a job within the top-level DAG
- In the DAG input file:

  SUBDAG EXTERNAL *JobName DagFileName*
- Any number of levels
- Sub-DAG nodes are like any other (can have PRE/POST scripts)
- Each sub-DAG has its own DAGMan
  - Separate throttles for each sub-DAG

Why nested DAGs?

- DAG re-use
- Scalability
- Re-try more than one node
- Short-circuit parts of the workflow
- Dynamic workflow modification (sub-DAGs can be created "on the fly")

Splices

- Directly includes splice DAG's nodes within the top-level DAG
- In the DAG input file:
  SPLICE *JobName DagFileName*
- Splices cannot have PRE and POST scripts (for now)
- No retries
- Splice DAGs must exist at submit time
- Splices can be nested (and combined with sub-DAGs)

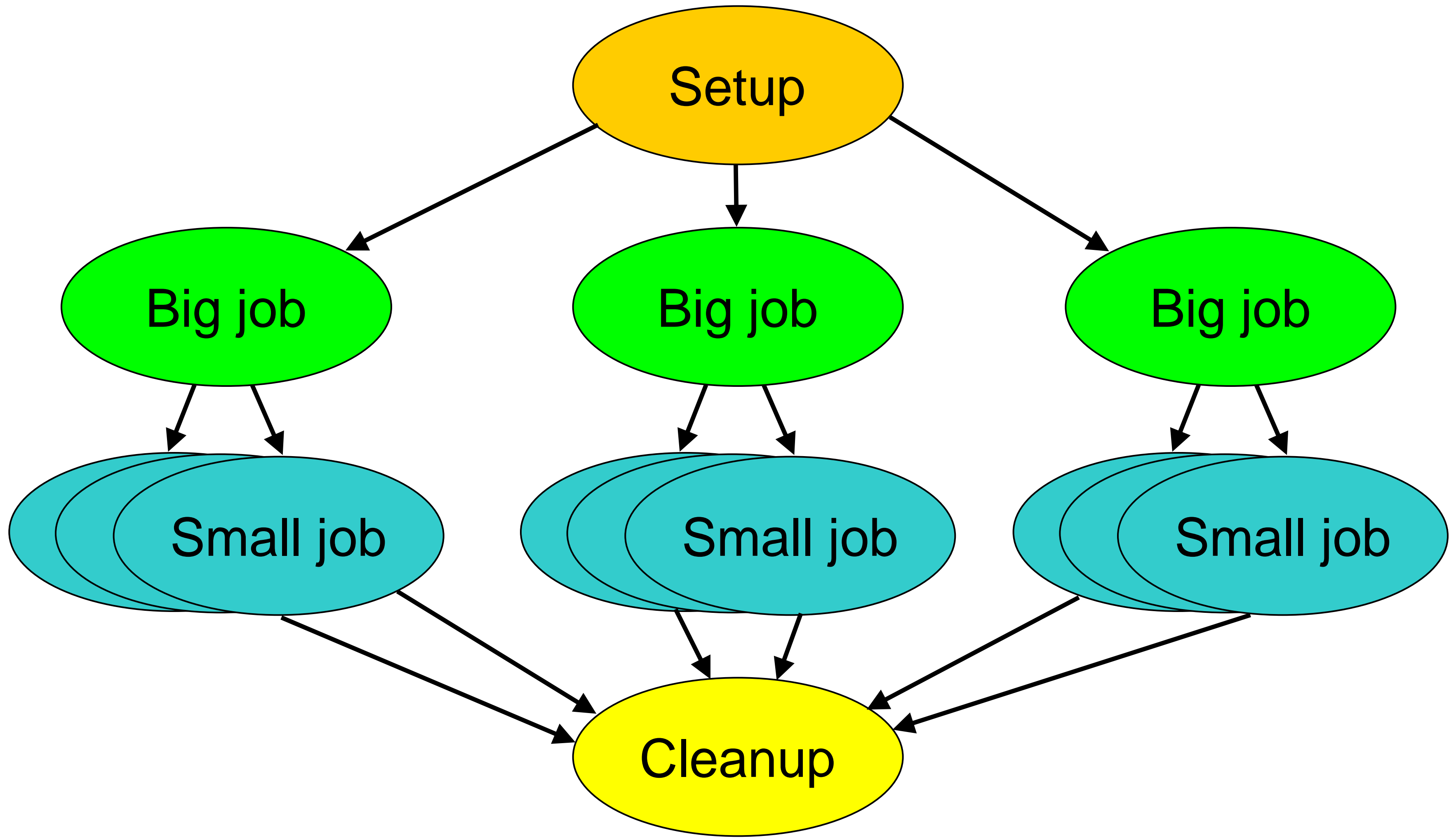
Why splices?

- DAG re-use
- Advantages of splices over sub-DAGs
  - Reduced overhead (single DAGMan instance)
  - Simplicity (e.g., single rescue DAG)
  - Throttles apply across entire workflow

Throttling

- Limit load on submit machine and pool
  - `Maxjobs` limits jobs in queue/running
  - `Maxidle` submit jobs until idle limit is hit
    - Can get more idle jobs if jobs are evicted
  - `Maxpre` limits PRE scripts
  - `Maxpost` limits POST scripts
- All limits are *per DAGMan*, not global for the pool or submit machine
- Limits can be specified as arguments to **`condor_submit_dag`** or in configuration

Node categories



Node category throttles

- Useful with different types of jobs that cause different loads
- In the DAG input file:
  CATEGORY *JobName CategoryName*
  MAXJOBS *CategoryName MaxJobsValue*
- Applies the ***MaxJobsValue*** setting to only jobs assigned to the given category
- Global throttles still apply

Cross-splice node categories

- Prefix category name with “+”

```
MaxJobs +init 2
Category A +init
```

- See the Splice section in the manual for details

DAG abort

- In DAG input file:
  ABORT-DAG-ON *JobName AbortExitValue*
  [RETURN *DagReturnValue*]
- If node value is ***AbortExitValue***, the entire DAG is aborted, implying that queued node jobs are removed, and a rescue DAG is created.
- Can be used for conditionally skipping nodes (especially with sub-DAGs)

FINAL Nodes

- FINAL node *always* runs at end of DAG (even on failure)
- Use **`FINAL`** in place of **`JOB`** in DAG file
- At most one FINAL node per DAG
- FINAL nodes cannot have parents or children (but can have PRE/POST scripts)
- New in 7.7.5

FINAL Nodes (cont)

- Success or failure of the FINAL node determines the success of the entire DAG
- PRE and POST scripts of FINAL nodes can use **`$DAG_STATUS`** and **`$FAILED_COUNT`** to determine the state of the workflow

Node priorities

- In the DAG input file:
  `PRIORITY JobName PriorityValue`
- Determines order of submission of ready nodes
- DAG node priorities are copied to job priorities (including sub-DAGs)
- Does *not* violate or change DAG semantics
- Higher numerical value equals "better" priority

Node priorities (cont)

- Better priority nodes are not guaranteed to run first!
- Child nodes get the largest priority of its own and all of its parents. Let us know if you want a different policy.
- For sub-DAGs, pretend that the sub-DAG is spliced in.

DAGMAN_HOLD_CLAIM_TIME

- An optimization introduced in HTCondor version 7.7.5 as a configuration option
- If a DAGMan job has child nodes, it will instruct the HTCondor schedd to hold the machine claim for the integer number of seconds that is the value of this option, which defaults to 20.
- Next job starts w/o negotiation cycle, using existing claim on startd

More information

- There's much more detail, as well as examples, in the DAGMan section of the online HTCondor manual.

Relevant Links

- DAGMan: http://research.cs.wisc.edu/htcondor/dagman/dagman.html
- Pegasus: http://pegasus.isi.edu/
- Makeflow: http://nd.edu/~ccl/software/makeflow/
- For more questions: htcondor-admin@cs.wisc.edu