CENTER FOR HIGH THROUGHPUT COMPUTING

HTCondor

HTCondor and Workflows: Advanced Tutorial

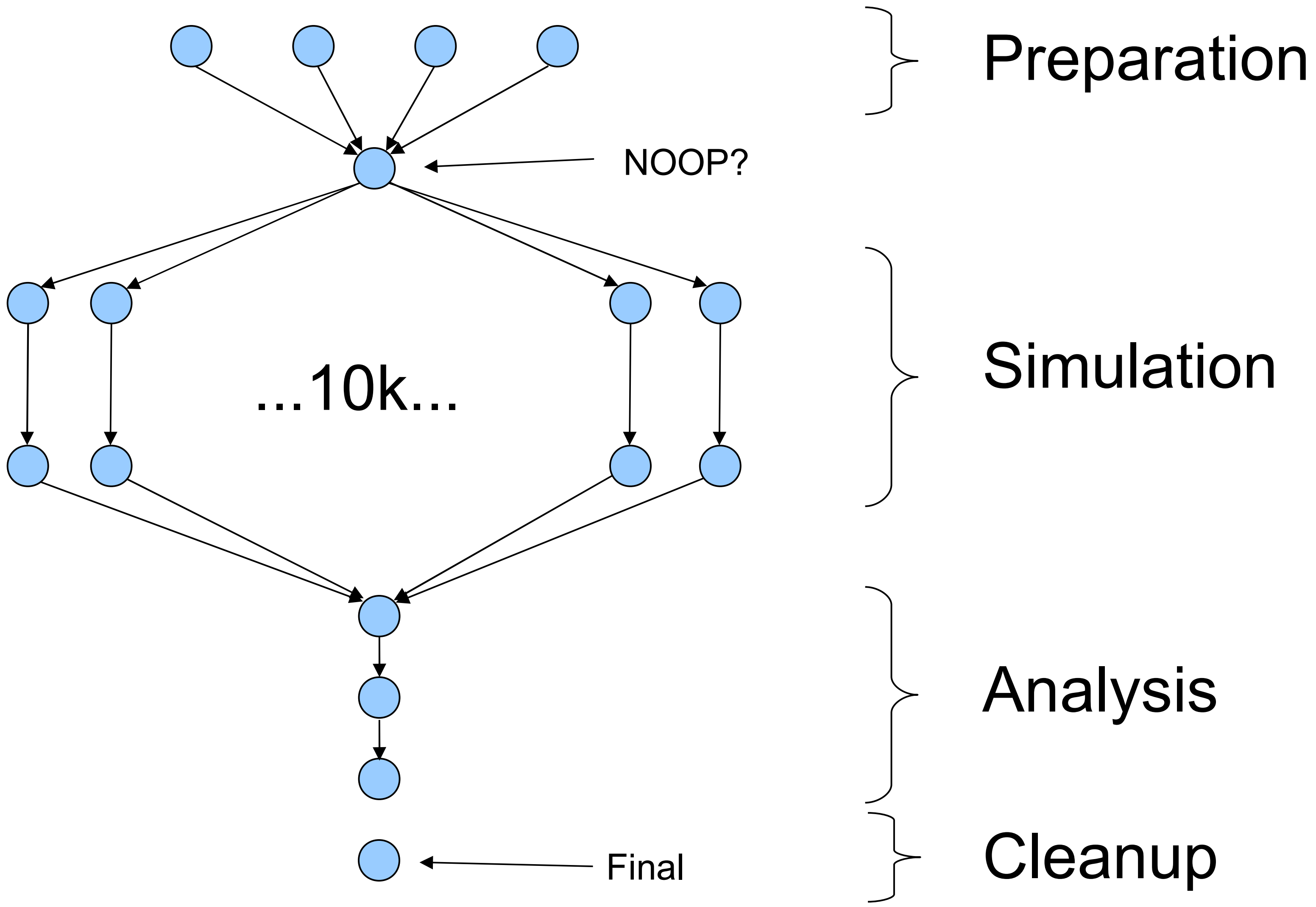
HTCondor Week 2014

Kent Wenger

Workflows in HTCondor

- This talk: techniques & features
- Please ask questions!

Example workflow



How big?

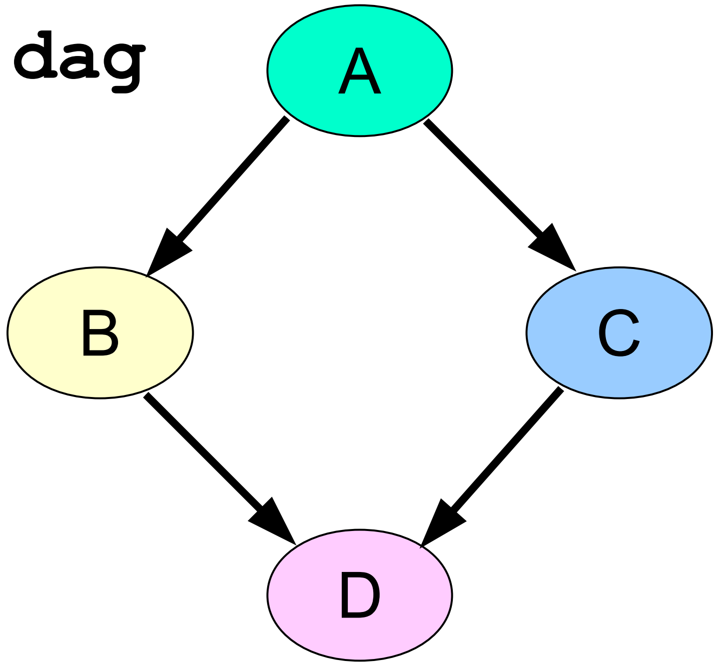
- We have users running 500k-job workflows in production (user hit 2.2 GB DAG file bug)
- Depends on resources on submit machine (memory, max. open files)
- “Tricks” can decrease resource requirements (talk to me)

Pretty big!

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
```



Organization of files and directories

Files and directories

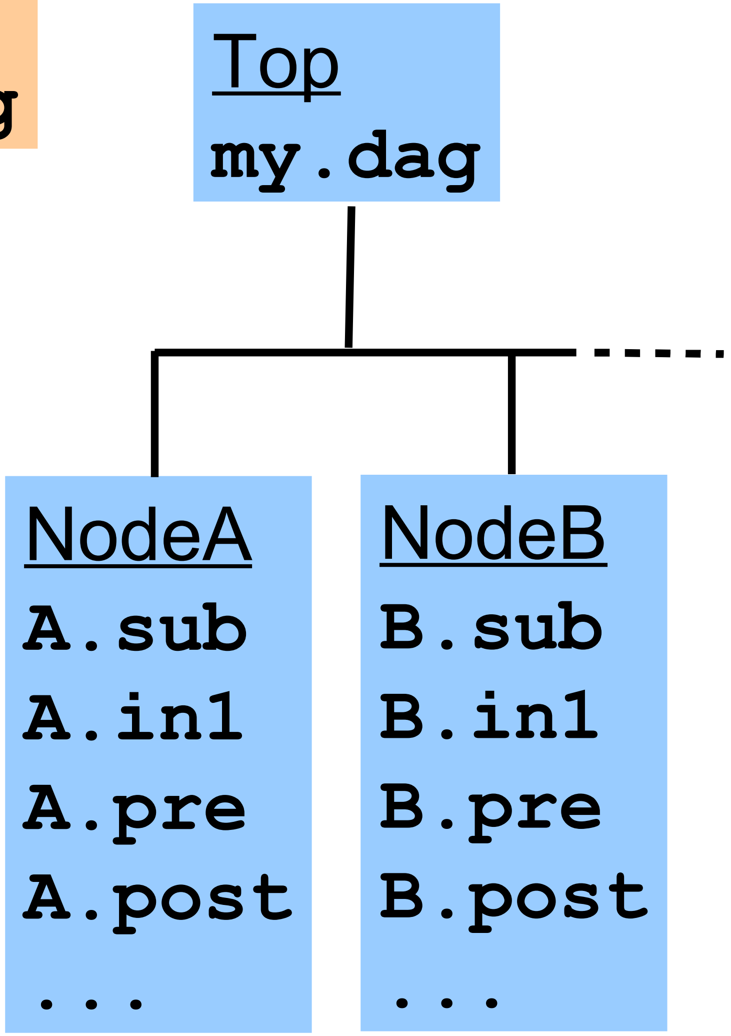
- By default, all paths in a DAG input file *and the associated submit files* are relative to the current working directory when **`condor_submit_dag`** is run.
- Modified by **`DIR`** directive on **`JOB`** command
- Also by **`-usedagdir`** on **`condor_submit_dag`** command line

Nodes in subdirectories

```
# in Top:
condor_submit_dag my.dag
```

```
# my.dag:
Job A A.sub Dir NodeA
Script Pre A A.pre
Job B B.sub Dir NodeB
...
```

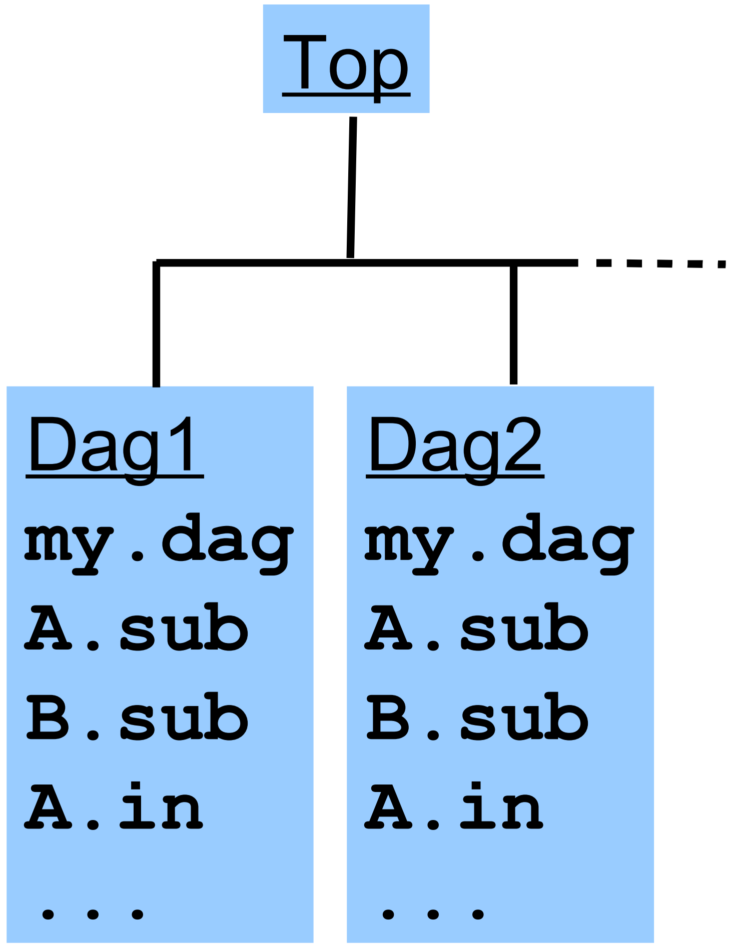
```
# A.sub
...
input = A.in1
...
```



DAGs in subdirectories

```
# in Top:
condor_submit_dag
   -usedagdir
   Dag1/my.dag
   Dag2/my.dag ...
```

```
# Dag1/my.dag:
Job A A.sub
Job B B.sub
...
```

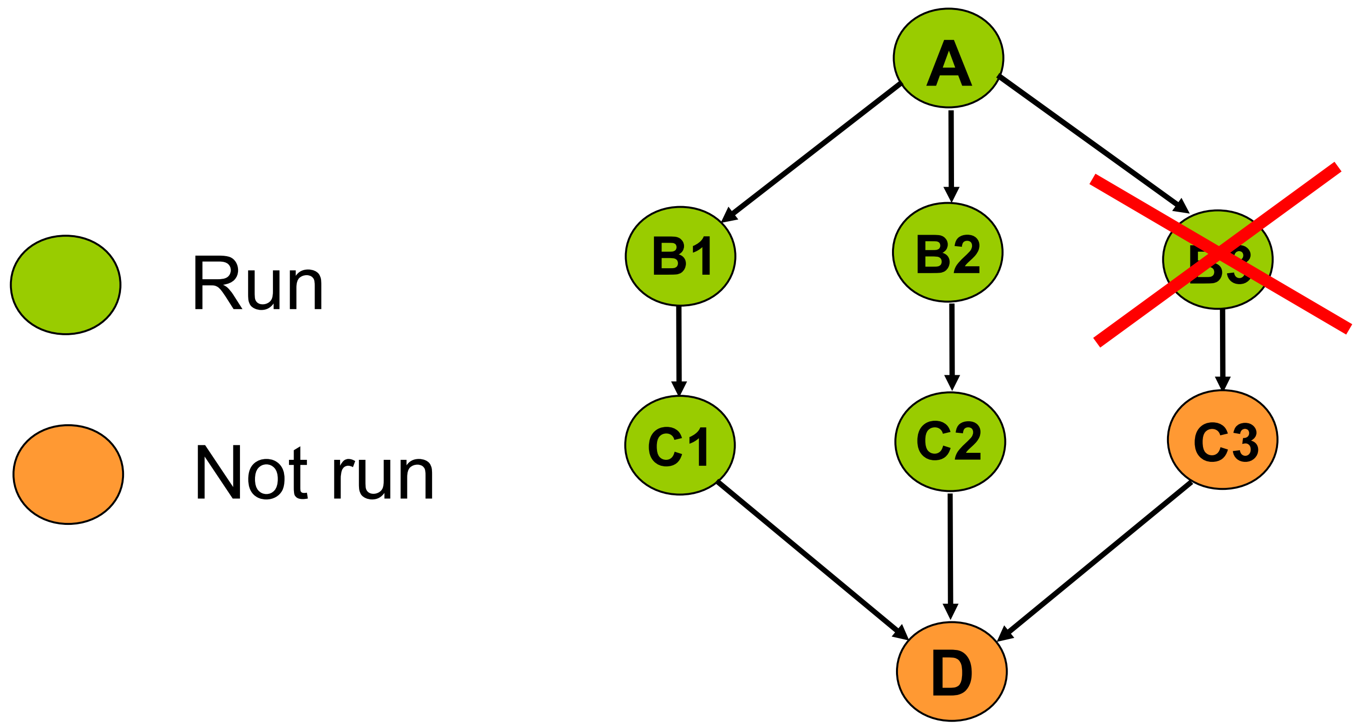


Multiple independent jobs

- Why use DAGMan?:
  - Throttling
  - Retry of failed jobs
  - Rescue DAG
  - PRE/POST scripts
  - Submit file re-use
- DAG file has JOB commands but not PARENT...CHILD

Rescue DAGs

Rescue DAGs (cont)



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`** or when DAG is halted and all queued jobs finish
  - 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`**)

CENTER FOR HIGH THROUGHPUT COMPUTING

HTCondor

Rescue DAGs (cont)

- 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.

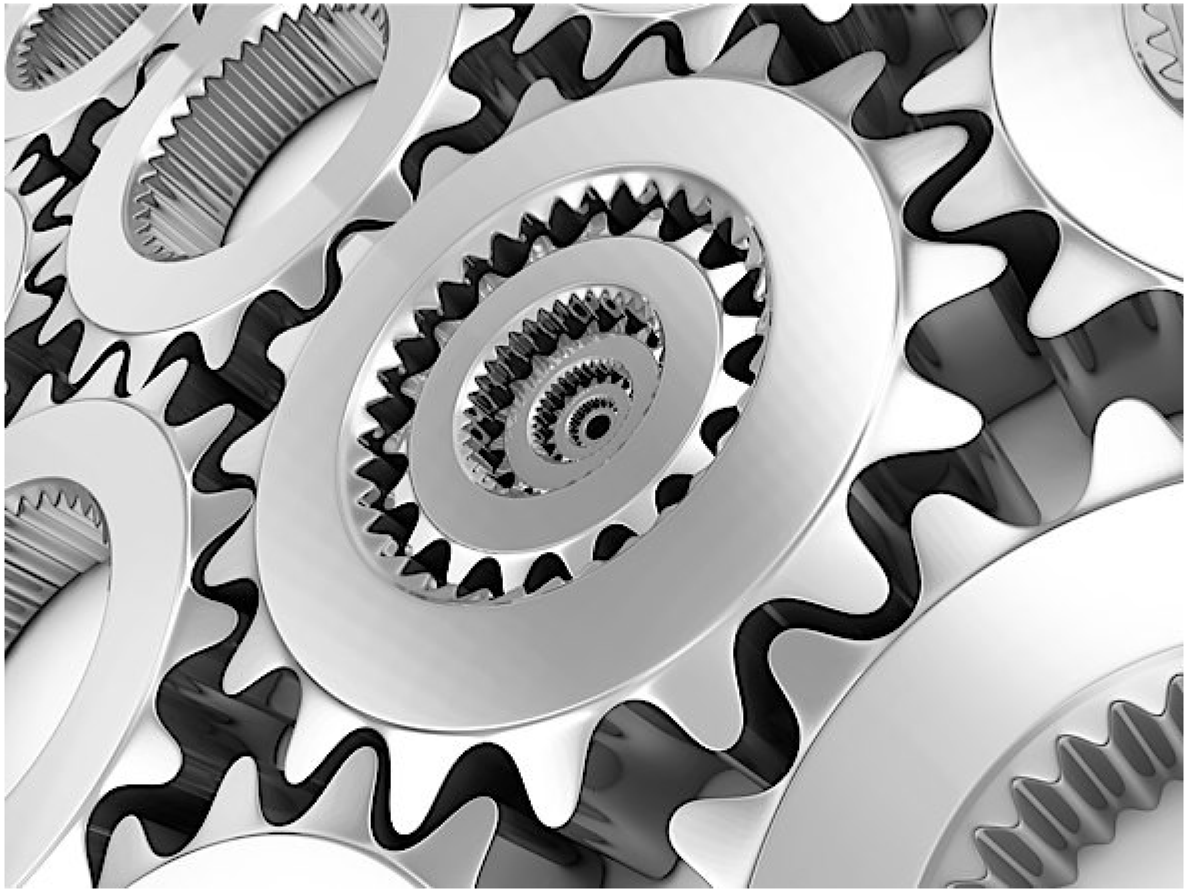
Rescue DAGs (cont)

- 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

DAGMan configuration



DAGMan configuration (cont)

- 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 command line:
  `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
```

Configuration: workflow log file

- DAGMan now uses a single log file for all node jobs
- Put workflow log file on local disk
- DAGMAN_DEFAULT_NODE_LOG
- In 8.1(/8.2): changes for global config
  - `@(DAG_DIR)/@(DAG_FILE).nodes.log`
  - `/localdisk/@(DAG_FILE).nodes.log`

Pre skip

DAG node with scripts: PRE_SKIP

- Allows PRE script to immediately declare node successful (job and POST script are not run)
- In the DAG input file:

```
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, 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)

No-op nodes

No-op nodes (cont)

- 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

No-op nodes (ex)

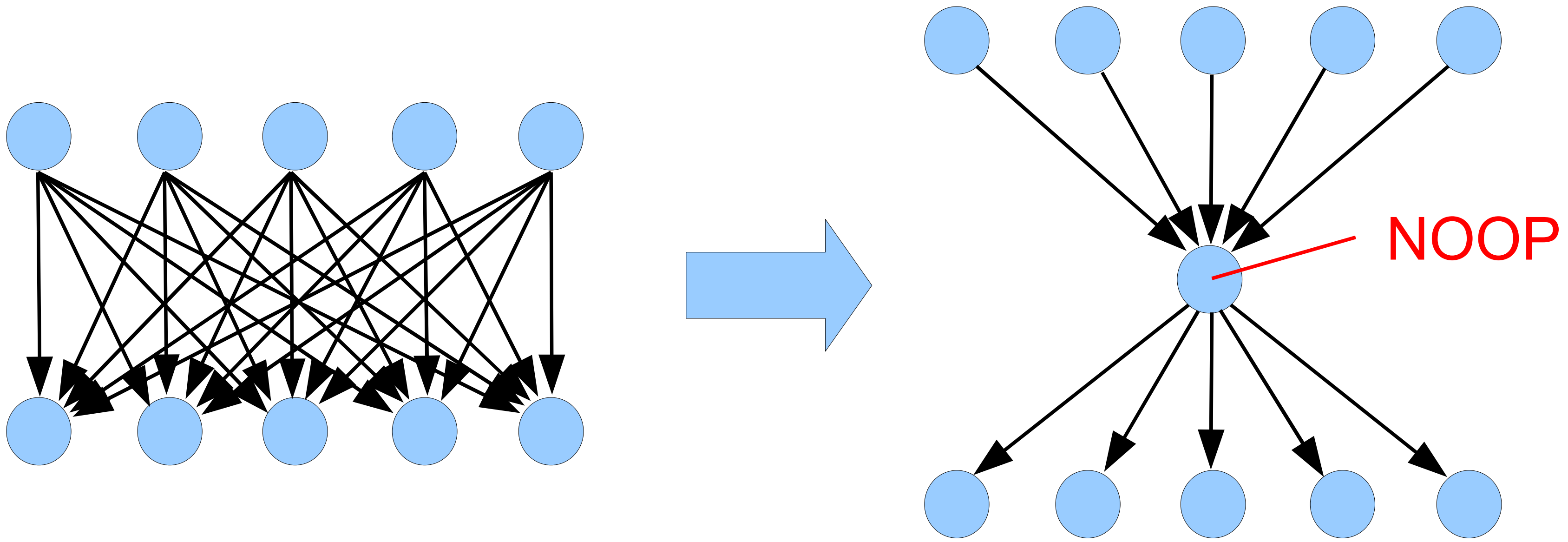
- Here is an example:

```
# 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.

No-op nodes (ex 2)

Simplify dag structure



Node retries

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

```
Retry C 4
```

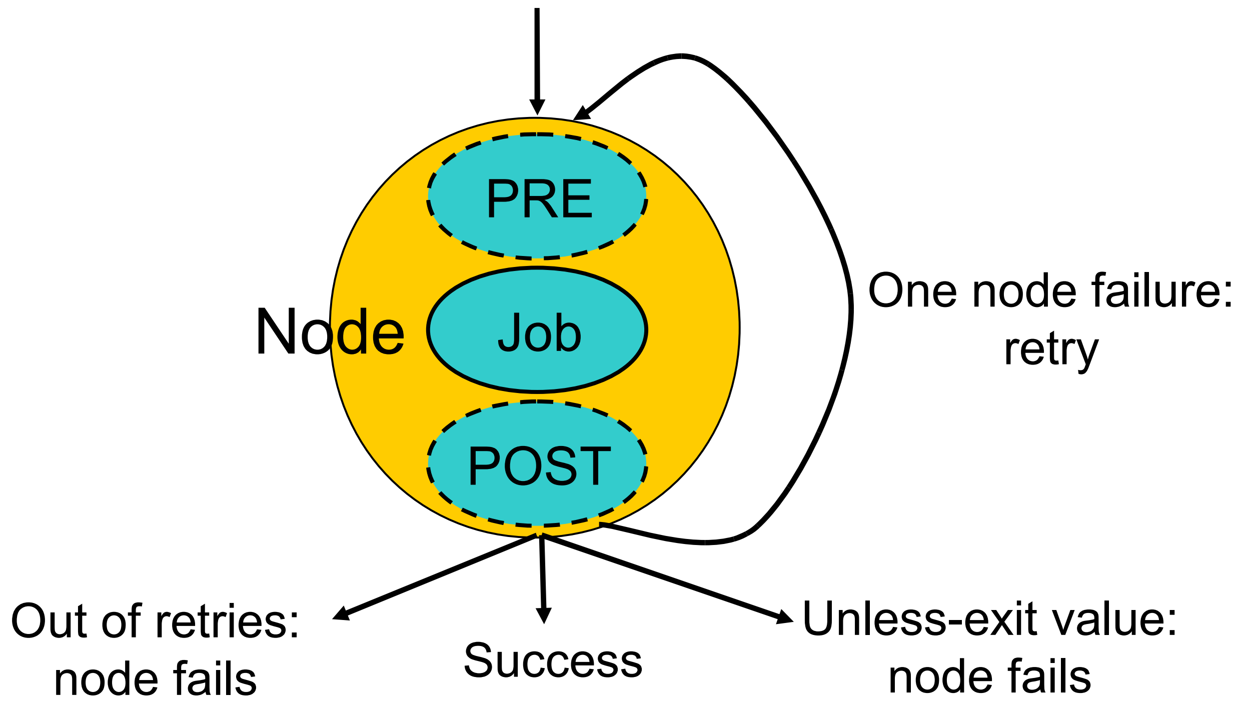
(to retry node `C` four times before calling the node failed)

  - 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

Node variables (cont)

- To re-use submit files
- In DAG input file:
  `VARS` ***`JobName varname="value"`*** **`[`*****`varname="value"`*****`... ]`**
- In submit description file:
  `$(`***`varname`*****`)`**
- ***`varname`*** can only contain alphanumeric characters and underscore
- ***`varname`*** cannot begin with "**`queue`**"
- ***`varname`*** is not case-sensitive
- ***`varname`*** beginning with "+" defines classad attribute (e.g., **`+State = "Wisconsin"`**)

CENTER FOR HIGH THROUGHPUT COMPUTING

HTCondor

Node variables (cont)

- ***`Value`*** cannot contain single quotes; double quotes must be escaped
- The variable **`$(JOB)`** contains the DAG node name
- **`$(RETRY)`** contains retry count
- Any number of VARS values per node
- DAGMan warns if a VAR name is defined more than once for a node

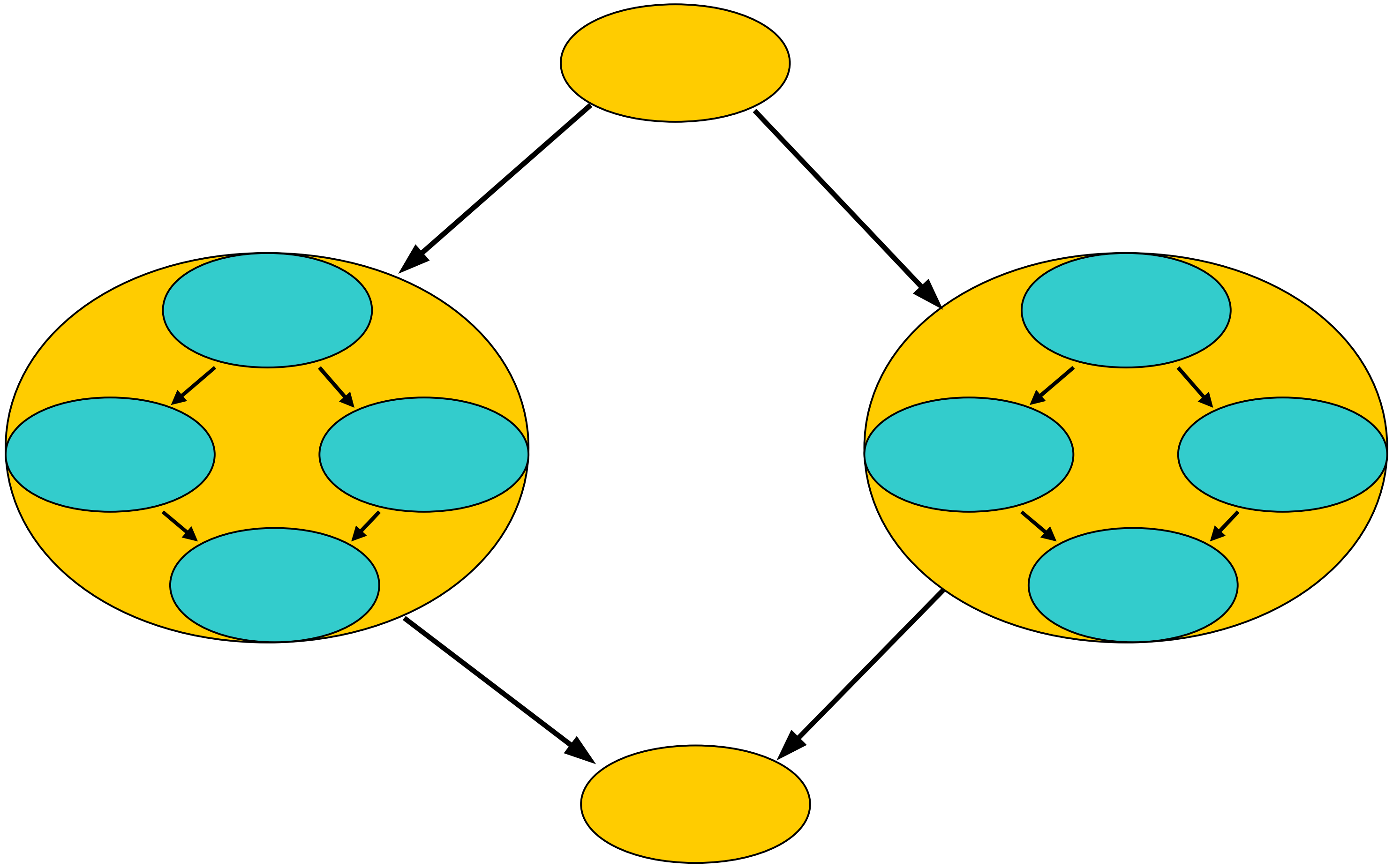
Node variables (ex)

```
# foo.dag
Job B10 B.sub
Vars B10 infile=”B_in.10”
Vars B10 +myattr=”4321”

# B.sub
input = $(infile)
arguments = $$([myattr])
```

Nested DAGs

Nested DAGs (cont)



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, retries, DIR, etc.)
- Each sub-DAG has its own DAGMan
  - Separate throttles for each sub-DAG
  - Separate rescue DAGs

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

Splices (cont)

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

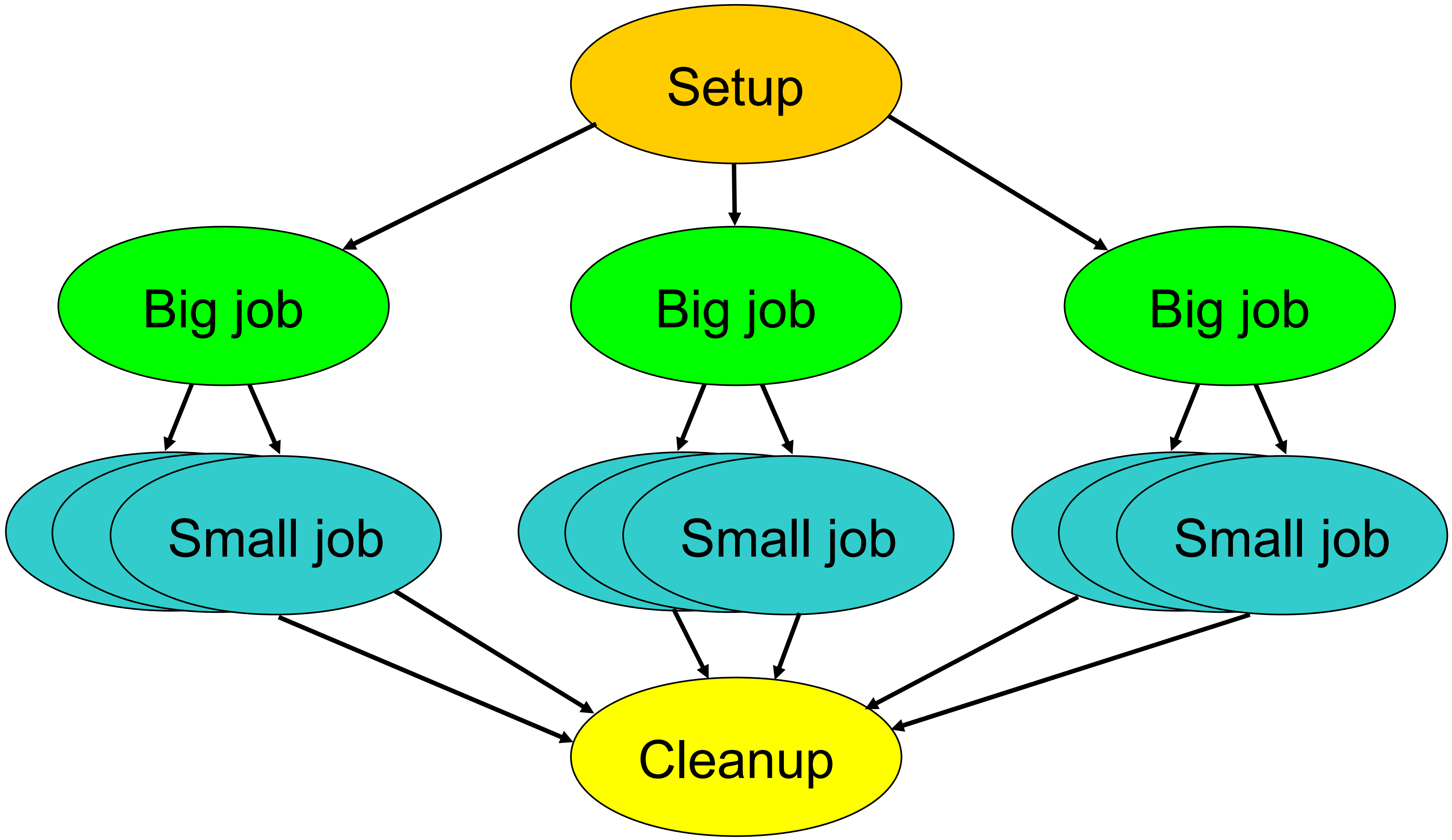
Throttling

Throttling (cont)

- Limit load on submit machine and pool
  - **`Maxjobs`** limits jobs in queue
  - **`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 categories (cont)



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

Node priorities

Node priorities (cont)

- 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!
- Effective node prio = max(explicit node prio, parents' effective prios, DAG prio)
- For sub-DAGs, pretend that the sub-DAG is spliced in.
- Overrides priority in node job submit file

Node priorities (upcoming changes)

- Priority change to DAGMan job “trickles down” to nodes
- Different “inheritance” policy:
  - Effective node prio = explicit node prio + DAG prio?
  - Effective node prio = average(explicit node prio, parents' effective prios, DAG prio)?

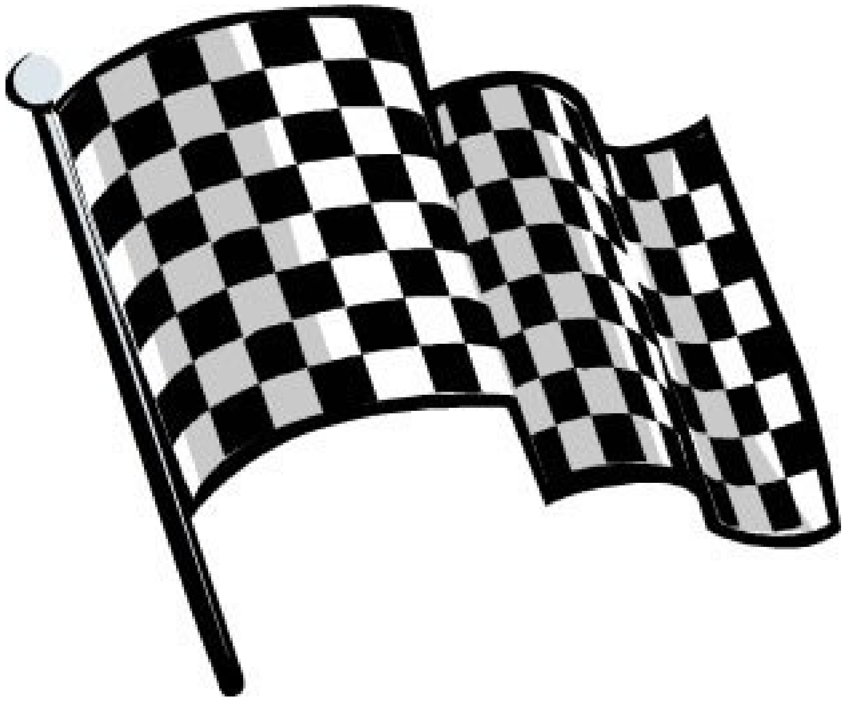
DAG abort

- 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 nodes (cont)

- 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)

FINAL nodes (cont)

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

Advanced workflow monitoring

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

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
- As of 8.1.6, *in ClassAd format* (a set of ClassAds)

Node status file contents

```
[
  Type = "DagStatus";
  DagFiles = {
    "job_dagman_node_status.dag"
  };
  Timestamp = 1397683160; /* "Wed Apr 16 16:19:20
  2014" */
  DagStatus = 3; /* "STATUS_SUBMITTED ()" */
  NodesTotal = 12;
  NodesDone = 0;
  NodesPre = 0;
  NodesQueued = 1;
  ...
```

Node status file contents (cont)

```
[
  Type = "NodeStatus";
  Node = "C";
  NodeStatus = 6; /* "STATUS_ERROR" */
  StatusDetails = "Job proc (1980.0.0)
 failed with status 5";
  RetryCount = 2;
  JobProcsQueued = 0;
  JobProcsHeld = 0;
]
```

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
...
```

DAGMan metrics

- *Anonymous* workflow metrics (for Pegasus)
- Metrics file (JSON format) generated at end of run (***`dagfile.metrics`***)
- Reported by default (can be disabled)
- Dagman.out tells whether metrics were reported

DAGMan metrics example

```
{
    "client":"condor_dagman",
    "version":"8.1.6",
    ...
    "start_time":1396448008.138,
    "end_time":1396448047.596,
    "duration":39.458,
    "exitcode":0,
    ...
    "total_jobs":3,
    "total_jobs_run":3,
    "total_job_time":0.000,
    "dag_status":0
}
```

More information

- There's much more detail, as well as examples, in the DAGMan section of the online HTCondor manual.
- DAGMan: http://research.cs.wisc.edu/htcondor/dagman/dagman.html
- For more questions: htcondor-admin@cs.wisc.edu, htcondor-users@cs.wisc.edu

Extra slides

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

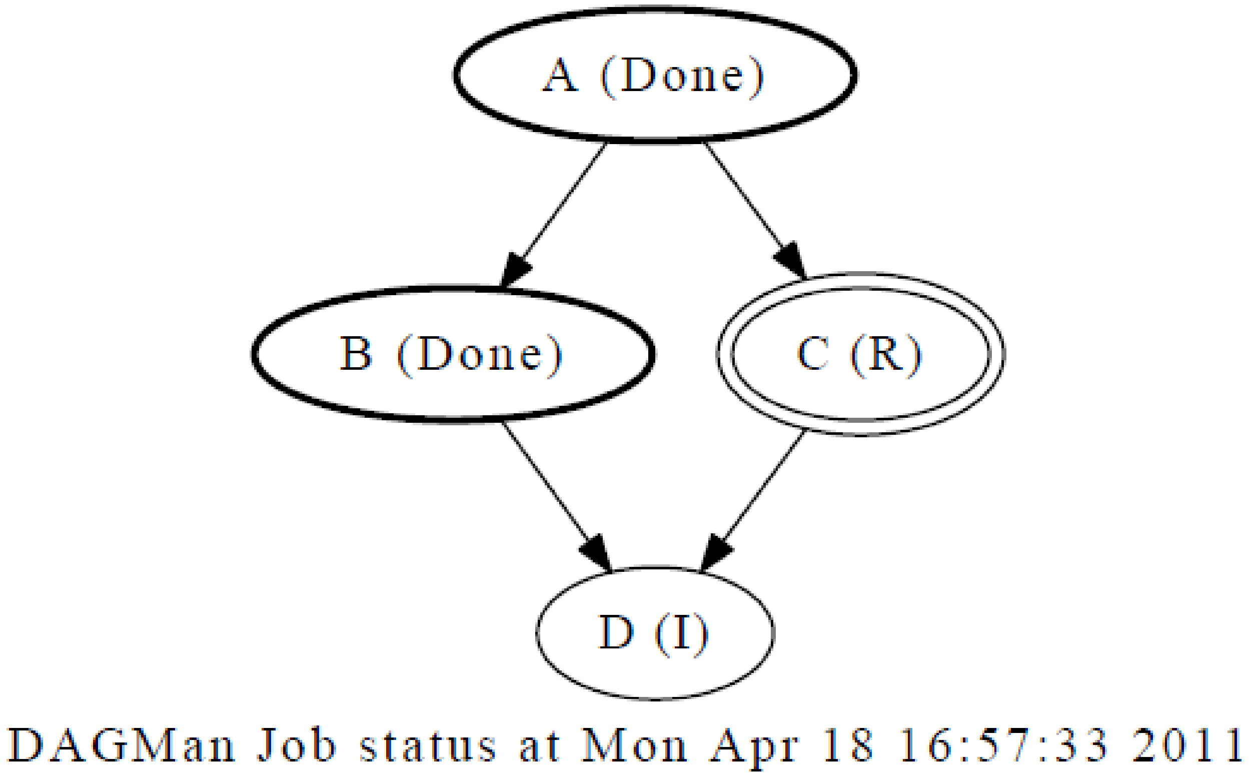
Dot file

- 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


A (Done)
B (Done)
C (R)
D (I)
DAGMan Job status at Mon Apr 18 16:57:33 2011