Outline

> Introduction/motivation
> Basic DAG concepts
> Running and monitoring a DAG
> Configuration
> Rescue DAGs and recovery
> Advanced DAGMan features
My jobs have dependencies...

Can Condor help solve my dependency problems?

Yes!

Workflows are the answer
What are workflows?

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

Set up input

Process
Process
Process
Process
Process

Collate output
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
Workflow tools

- **DAGMan**: Condor’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**
LIGO inspiral search application

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

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

- **Directed Acyclic Graph Manager**

- DAGMan allows Albert to specify the dependencies between his Condor jobs, so DAGMan manages the jobs automatically.

- Dependency example: do not run job B until job A has completed successfully.
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!
Condor and DAGs

- Each node represents a Condor job (or cluster)
- Dependencies define the possible order of job execution
A DAG input file defines a DAG:

```bash
# 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 are removed.
- No macros in “log” submit entries (for now)
Node success or failure

- A node either succeeds or fails
- Based on the return value of the job(s)
  - 0 \Rightarrow success
  - not 0 \Rightarrow failure
- This example: C fails
- Failed nodes block execution; DAG fails
Outline

▶ Introduction/motivation
▶ Basic DAG concepts
▶ Running and monitoring a DAG
▶ Configuration
▶ Rescue DAGs and recovery
▶ Advanced DAGMan features
Submitting the DAG to Condor

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 a Condor job (in the scheduler universe)
- `-f(orce)` option forces overwriting of existing files
Vocabulary

- Rescue DAGs save the state of a partially-completed DAG, and are created when a node fails or the `condor_dagman` job is removed with `condor_rm`.
- PRE And POST scripts are code associated with a job that run on the submit host.
- Nested DAGs are jobs that are themselves DAGs.
Controlling running DAGs

> condor_rm
  - Removes all queued node jobs, kills PRE/POST scripts (removes *entire* workflow)
  - Kills PRE/POST scripts
  - Removes entire workflow
  - Creates rescue DAG
Controlling running DAGs (cont)

condor_hold and condor_release

- Node jobs continue when DAG is held
- No new node jobs submitted
- DAGMan “catches up” when released
Controlling running DAGS: the halt file

• New in Condor version 7.7.5.
• Create a file named \textit{DAGfile.halt} in the same directory as your DAG file.
• Jobs that are running will continue to run.
• No new jobs will be submitted and no PRE scripts will be run.
The halt file (cont)

- When all submitted jobs complete, DAGman creates a rescue dag and exits.
- When jobs finish, POST scripts will be run.
- When all submitted jobs complete, DAGman creates a rescue dag and exits.
The halt file (cont)

- If the halt file is removed, DAGman returns to normal operation.
condor_q -dag

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

```
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
```
dagman.out file

- DagFile.dagman.out
- Verbosity controlled by the `DAGMAN_VERBOSITY` configuration macro (new in 7.5.6) and `-debug` on the `condor_submit_dag` command line
- Directory specified by `-outfile_dir` directory
- Mostly for debugging
- Logs detailed workflow history
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

▶ In the DAG input file:
   `NODE_STATUS_FILE statusFileName [minimumUpdateTime]`
▶ Not enabled by default
▶ Shows a snapshot of workflow state
  • Overwritten as the workflow runs
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)
...
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

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

- In the DAG input file:
  \[\text{DOT } \text{DotFile} \ [\text{UPDATE}] \ [\text{DONT-OVERWRITE}]\]

- To create an image:
  \[\text{dot -Tps DotFile -o PostScriptFile}\]

- Shows a snapshot of workflow state
Dot file example

DAGMan Job status at Mon Apr 18 16:57:33 2011
Outline

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

- A few dozen DAGMan-specific configuration macros (see the manual...)
- From lowest to highest precedence
  - Condor 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 is an error.
Per-DAG configuration (cont)

- Configuration entries not related to DAGman are ignored by DAGman
- Syntax like any other Condor config file
Outline

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

Run

Not run

A

B1

B2

B3

C1

C2

C3

D
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
Rescue DAGs (cont)

➤ Automatically run when you re-run the original DAG (unless -f) (since 7.1.0)
➤ DAGman immediately exits after writing a rescue DAG file
Rescue DAGs (cont)

- New in condor version 7.7.2, the Rescue DAG file, by default, is only a partial DAG file

- `DAGMAN_WRITE_PARTIAL_RESCUE = False` turns this off.
Rescue DAGs (cont)

- 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.
Rescue DAGs (cont)

> 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
Recovery mode

- Happens automatically when DAGMan is held/released, or if DAGMan crashes and restarts
- Node jobs continue
- DAGMan recovers node job state
- DAGMan is robust in the face of failures
Outline

▶ Introduction/motivation
▶ Basic DAG concepts
▶ Running and monitoring a DAG
▶ Configuration
▶ Rescue DAGs and recovery
▶ 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

> 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 (dynamically)
- Force jobs to run on same machine
Script argument variables

- `$JOB`: node name
- `$JOBID`: Condor ID (cluster.proc)
- `$RETRY`: current retry
- `$MAX_RETRIES`: max # of retries
- `$RETURN`: exit code of Condor/Stork job (POST only)
Script argument variables (cont)

> `$PRE_SCRIPT_RETURN`: More on this below (POST only)

> `$DAG_STATUS`: A number indicating the state of DAGman. See the manual for details.

> `$FAILED_COUNT`: is simply the number of nodes that have failed in the DAG
NOOP nodes

- It is useful to have the ability to check your work.
- 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.
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.
DAG node with scripts: PRE_SKIP

- New feature in Condor version 7.7.2.
- Here is the syntax:
  
  ```
  JOB A A.cmd
  SCRIPT PRE A A.pre
  PRE SKIP A non-zero integer
  ```

- Here, the PRE script of A will run. If the script exits with the indicated value, this is normally a failure.
DAG node with scripts: PRE_SKIP (cont)

- DAGman instead recognizes this as an indication to succeed this node immediately, and skip the node job and POST script.
- If the PRE script fails with a different value, the node job is skipped, and the postscript runs.
DAG node with scripts: PRE_SKIP (cont)

> When the postscript runs, the $PRE_SCRIPT_RETURN variable contains the return value from the prescript. (See manual for specific cases)
NOOP nodes

- It is useful to have the ability to check your work.
- 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.
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 anything.
Node retries

> In case of 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

- One node failure: retry
- Out of retries: node fails
- Unless-exit value: node fails
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
- Cannot use variables in a log file name (for now)
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.
- DAGman warns if a VAR is defined more than once for a job.
Nested DAGs (cont)

- Runs the sub-DAG as a job within the top-level DAG
- In the DAG input file:
  \texttt{SUBDAG EXTERNAL JobName DagFileName}
- Any number of levels
- Sub-DAG nodes are like any other
- Each sub-DAG has its own DAGMan
  - Separate throttles for each sub-DAG
Why nested DAGs?

- Scalability
- Re-try more than one node
- Dynamic workflow modification
- DAG re-use
Throttling

- Limit load on submit machine and pool
  - `Maxjobs` limits jobs in queue/running
  - `Maxidle` submit jobs until idle limit is hit
  - `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

- Setup
  - Big job
    - Small job
  - Big job
    - Small job
  - Big job
    - Small job

- Cleanup
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
Splices
Splices (cont)

- Directly includes splice’s nodes within the top-level DAG
- In the DAG input file:
  \[ \text{SPLICE } \text{JobName } \text{DagFileName} \]
- Splices cannot have PRE and POST scripts (for now)
- No retries
- Splice DAGs must exist at submit time
Why splices?

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

Top level
# splice1.dag
Job A A.submit
Splice B splice2.dag
Job C C.submit
Parent A Child B
Parent B Child C

Splice
# splice2.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
DAG abort

- In DAG input file:
  \texttt{ABORT-DAG-ON JobName AbortExitValue [RETURN DagReturnValue]}

- If node value is \texttt{AbortExitValue}, the entire DAG is aborted, implying that jobs are removed, and a rescue DAG is created.

- Can be used for conditionally skipping nodes (especially with sub-DAGs)
FINAL Nodes

- Introduced in Condor version 7.7.5
- Use **FINAL** in place of **JOB** in DAG file.
- At most one FINAL node per DAGman.
- FINAL nodes cannot have parents or children.
FINAL Nodes (cont)

- The FINAL node is submitted after DAGman has made as much progress as possible.
- In case of a DAG failure, the FINAL node is run; some nodes may not be run, but the FINAL node will be run.
FINAL Nodes (cont)

- Success or failure of the FINAL node determines the success of the DAG run.
- It is envisioned that PRE and POST scripts of FINAL nodes will use $\text{DAG\_STATUS}$ and $\text{FAILED\_COUNT}$.
Node priorities

- In the DAG input file:
  PRIORITY JobName PriorityValue
- Determines order of submission of ready nodes
- Does not violate or change DAG semantics
- Higher numerical value equals “better” priority
Node priorities (cont)

- Child nodes get the largest priority of parents. This may or may not be useful. Let us know if you want a different policy.
- For subdags, pretend that the subdag is spliced in.
- DAGman priorities are copied to job priorities.
Depth-first DAG traversal

- Get some results more quickly
- Possibly clean up intermediate files more quickly
- `DAGMAN_SUBMITDEPTHFIRST=True`
Multiple DAGs

- On the command line:
  `condor_submit_dag dag1 dag2 ...`
- Runs multiple, independent DAGs
- Node names modified (by DAGMan) to avoid collisions
- Useful: throttles apply across DAGs
- Failure produces a single rescue DAG
Cross-splice node categories

- Prefix category name with “+”
  - MaxJobs +init 2
  - Category A +init

- See the Splice section in the manual for details
DAGMAN_HOLD_CLAIM_TIME

- An optimization introduced in Condor version 7.7.5 as a configuration option
- If a DAGman job has child nodes, it will instruct the condor schedd to hold the machine claim for the integer number of seconds that is the value of this option, which defaults to 20.
Thus, upon completion, the schedd will not go through a negotiation cycle before starting the job; it will simply start a new job with the old claim on the startd we have just finished using.
DAGMAN_USE STRICT

- New configuration option introduced in Condor version 7.7.0
- Think of it as `-Werror` for DAGman.
- If set to 0, no warnings become errors.
- If set to 3, all warnings become errors.
DAGMAN_USE_STRICT (ex)

One place where we check for warnings is the log file code: if we see strangeness, we print out a warning. If you are paranoid, you might want DAGman to write a rescue DAG and exit immediately, and set option = 3.
More information

There’s much more detail, as well as examples, in the DAGMan section of the online Condor manual.
Outline

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

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