Condor and Workflows: An Introduction

Condor Week 2011

Kent Wenger

Condor Project
Computer Sciences Department
University of Wisconsin-Madison

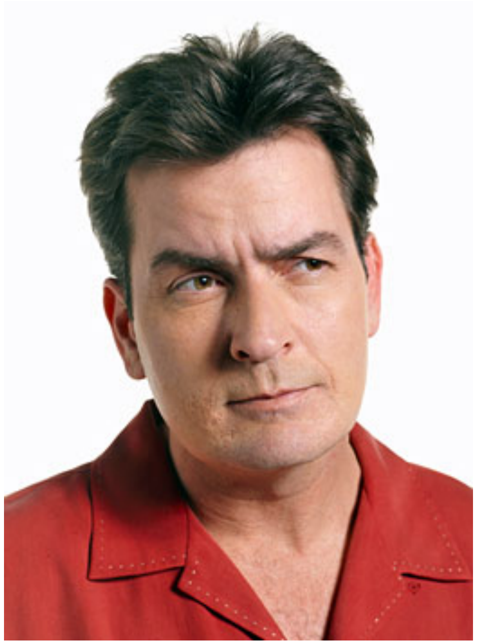
Outline

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

My jobs have dependencies…

Can Condor help solve my dependency problems?

Yes!

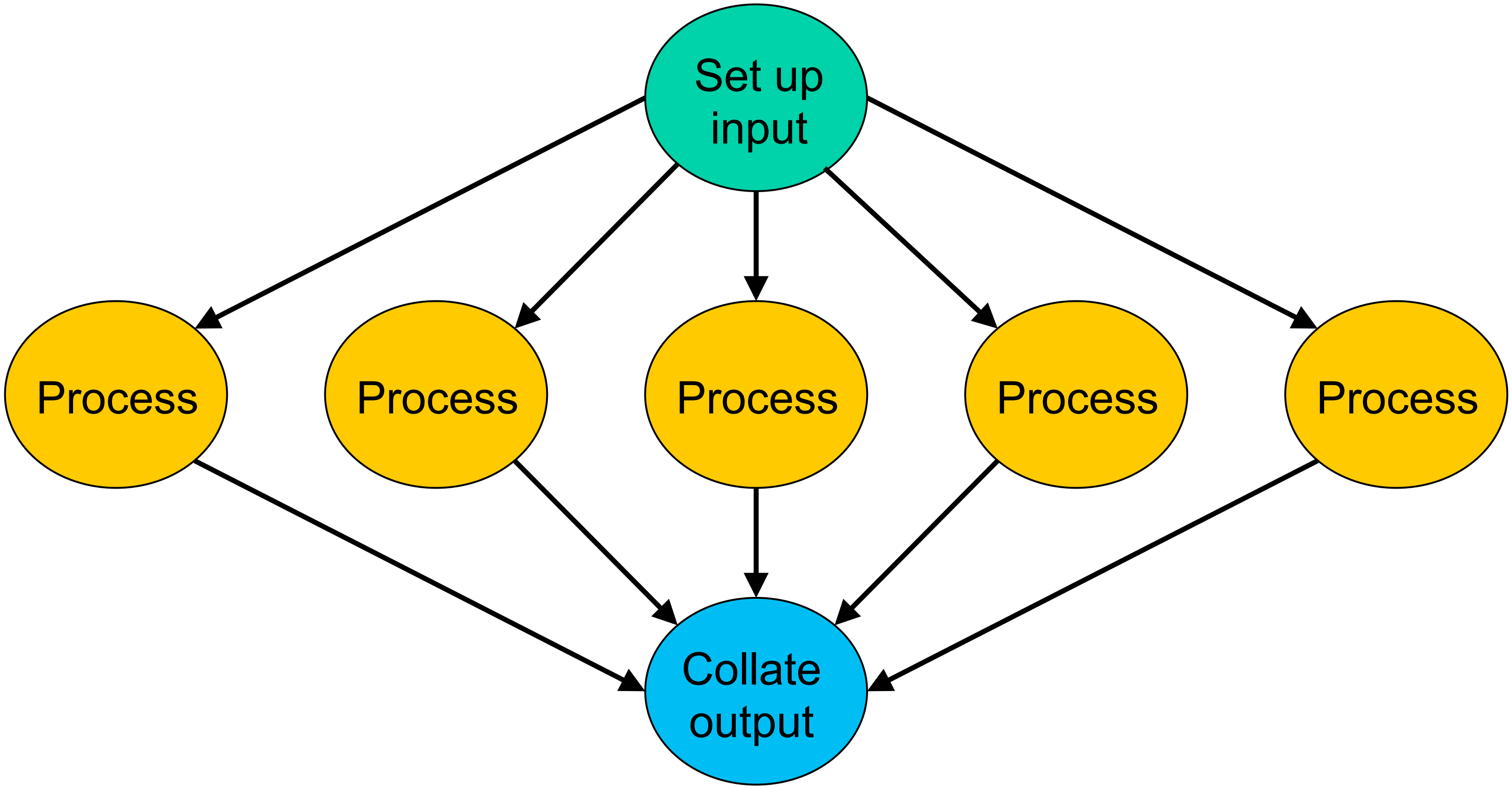


Workflows are the answer

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



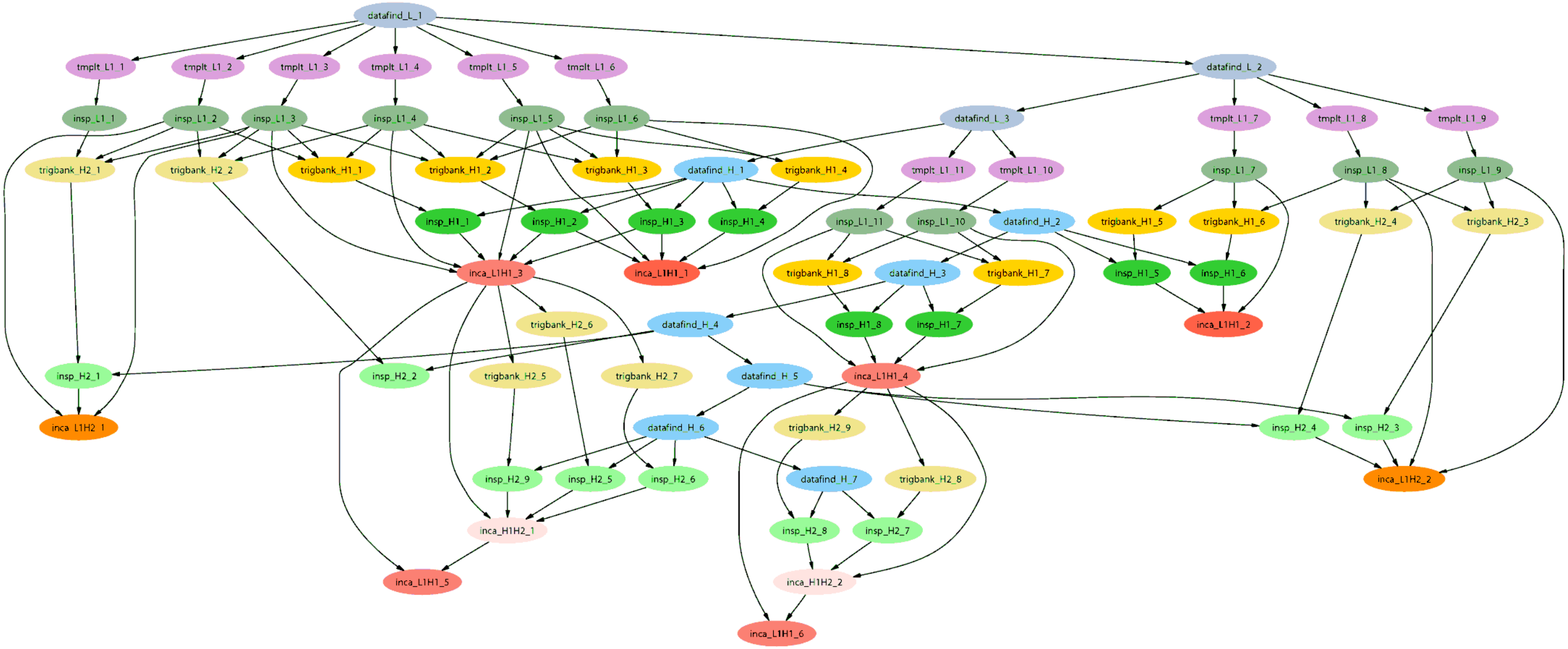
Workflows – launch and forget

- 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



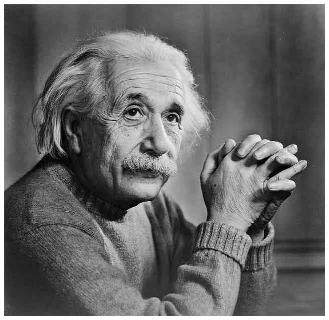
Inspiral workflow application is the work of Duncan Brown, Caltech, Scott Koranda, UW Milwaukee, and the LSC Inspiral group

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
- Pegasus

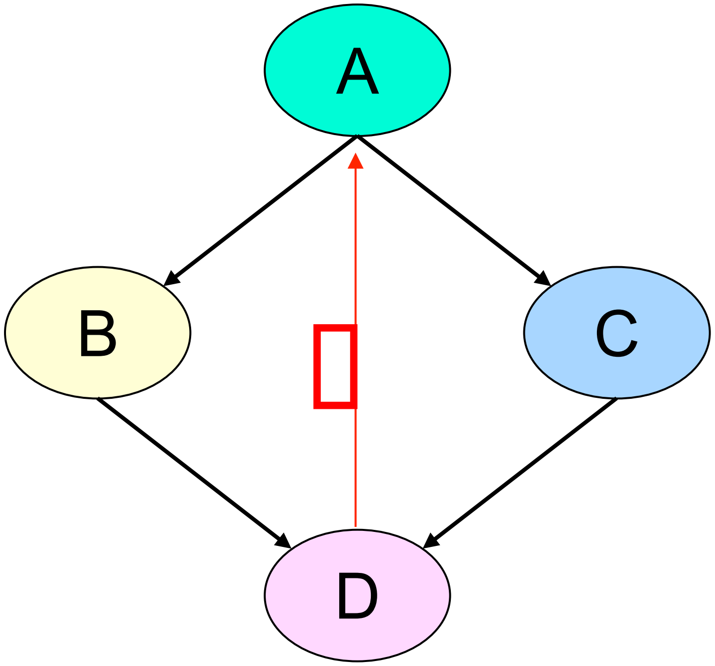


Albert learns DAGMan

- **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

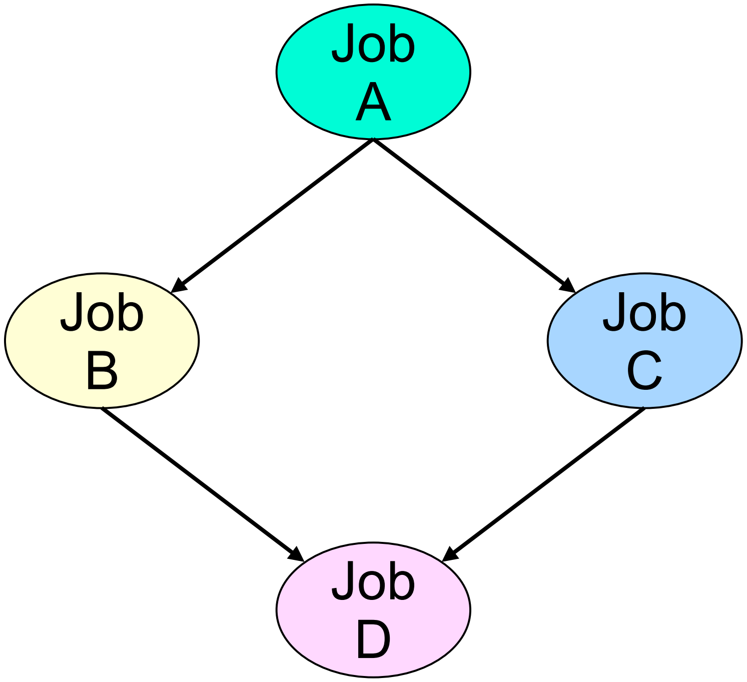
DAG definitions

- 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

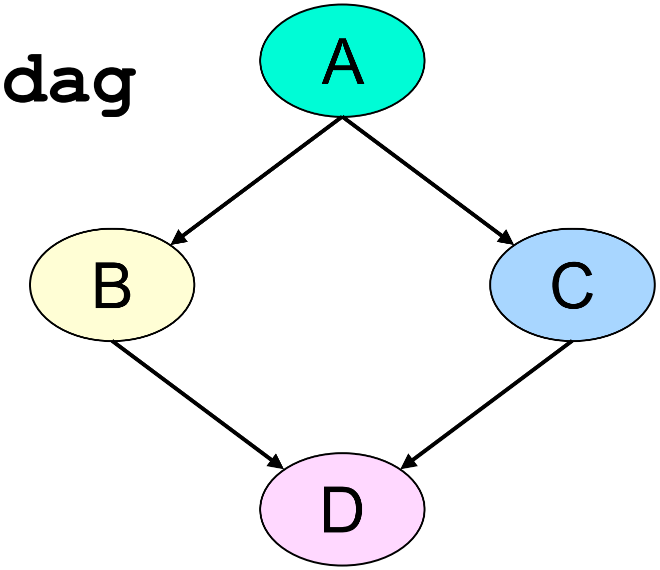
- Each node represents a Condor job (or cluster)
- Dependencies define the possible order of job execution



Defining a DAG to Condor

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

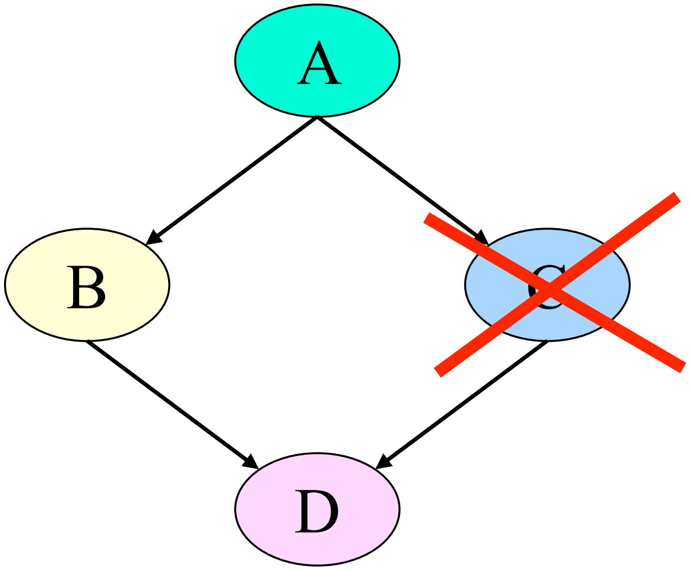
- Submit description files used in a DAG can create multiple jobs, but they must all be in a single cluster
- The failure of any job means the entire cluster fails. Other jobs are removed.

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



Outline

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

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

Controlling running DAGs

- `condor_rm`
  - Removes all queued node jobs, kills PRE/POST scripts (removes *entire* workflow)
  - Creates rescue DAG
- `condor_hold` and `condor_release`
  - Node jobs continue when DAG is held
  - No new node jobs submitted
  - DAGMan "catches up" when released

Monitoring a DAG run

- `condor_q -dag`
- `dagman.out` file
- Node status file
- `jobstate.log` file
- Dot file

condor_q -dag

- The **-dag** option associates DAG node jobs with the parent DAGMan job.
  - Only works for one level of DAG. *Nested DAGs do not work.*
- Shows current workflow state

condor_q -dag example

```
% condor_q -dag
-- Submitter: wenger@tonic.cs.wisc.edu : <128.105.121.53:59972> :
   tonic.cs.wisc.edu
 ID      OWNER/NODENAME   SUBMITTED     RUN_TIME ST PRI SIZE CMD
  82.0   wenger          4/15 11:48   0+00:01:02 R  0   19.5 condor_dagman -f
  84.0    |-B1           4/15 11:49   0+00:00:02 R  0   0.0  job_dagman_node
  85.0    |-B2           4/15 11:49   0+00:00:00 I  0   0.0  job_dagman_node
  86.0    |-B3           4/15 11:49   0+00:00:00 I  0   0.0  job_dagman_node
  87.0    |-B4           4/15 11:49   0+00:00:00 I  0   0.0  job_dagman_node
  88.0    |-B5           4/15 11:49   0+00:00:00 I  0   0.0  job_dagman_node
```

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
...
```

new and improved

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
- New in 7.5.4

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)
```

new and improved

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
- New in 7.5.5

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:

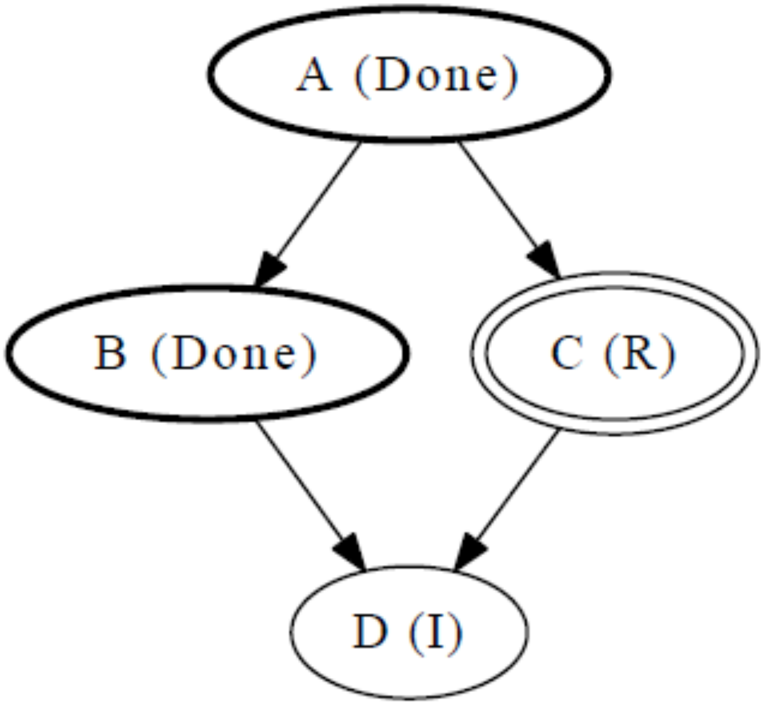
  ```
  DOT DotFile [UPDATE] [DONT-OVERWRITE]
  ```

- To create an image

  ```
  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
- Pegasus

DAGMan configuration

- 39 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
- Conflicting configuration specs → error
- 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
- Pegasus

Rescue DAGs

- 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
- Automatically run when you re-run the original DAG (unless -f) (since 7.1.0)

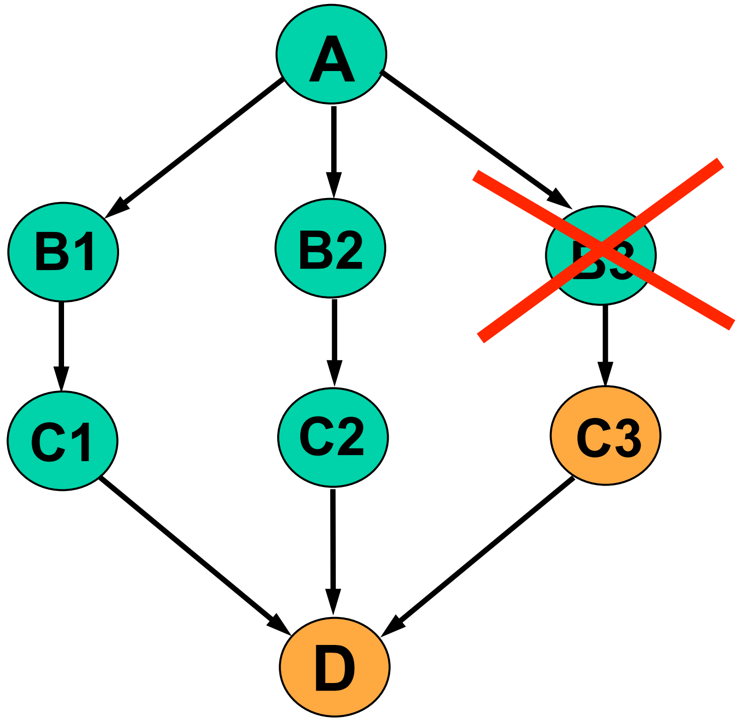
Rescue DAG naming

- **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

Rescue DAGs, cont.

🟢 Run

🟠 Not run



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
- Pegasus

PRE and POST scripts

- DAGMan allows PRE and/or POST scripts
  - Not necessarily a script: any executable
  - Run before (PRE) or after (POST) job
  - Run on the submit 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 (dynamically)
- Force jobs to run on same machine

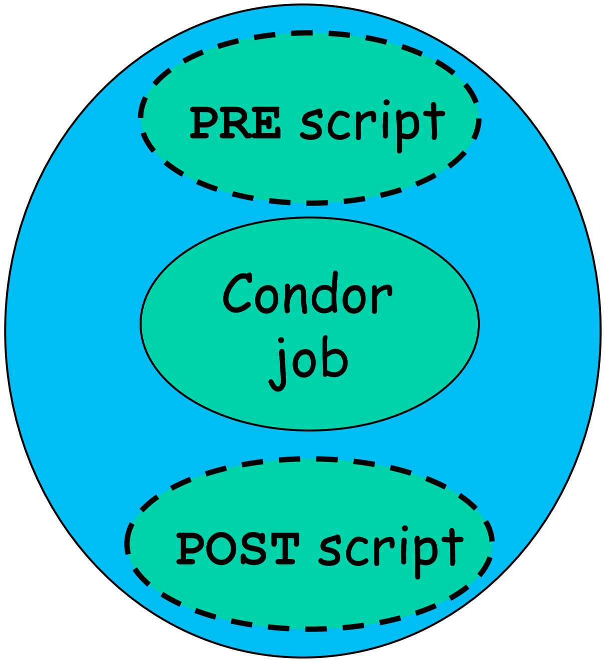
new and improved

Script argument variables

- **$JOB**: node name
- **$JOBID**: Condor ID (cluster.proc)
- **$RETRY**: current retry
- **$MAX_RETRIES**: max # of retries (new in 7.5.6)
- **$RETURN**: exit code of Condor/Stork job (POST only)

DAG node with scripts

- PRE script, Job, or POST script determines node success or failure (table in manual gives details)
- If PRE script fails, job and POST script are not run



new and improved

Default node job log

- Node job submit description files are no longer required to specify a log file (since 7.3.2)
- Default is **`DagFile.nodes.log`**
- Default log may be preferable (especially for submit file re-use)

new and improved

Lazy submit file reading

- Submit description files are now read lazily (since 7.3.2)
- Therefore, a PRE script can now write the submit description file of its own node job
- Also applies to nested DAGs, which allows some dynamic workflow modification

Node retries

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

```
Retry C 4
```

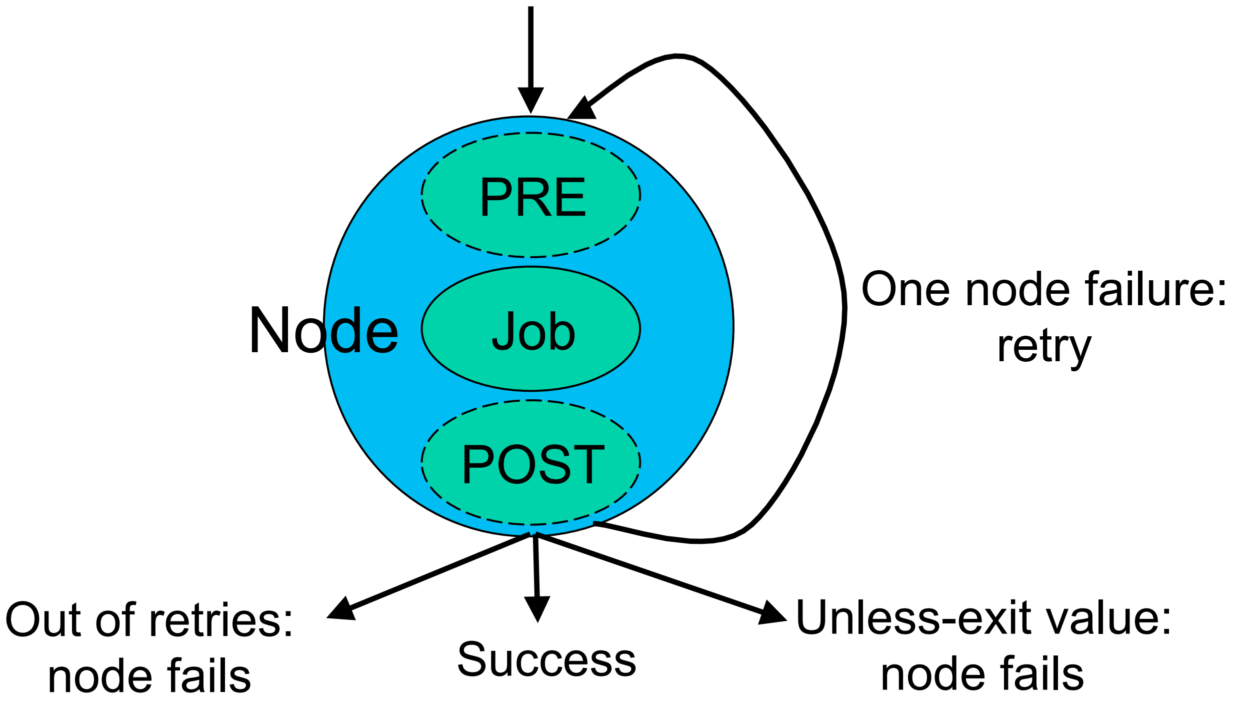
(to retry node C four times before calling the node failed)

  - 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



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
- Value cannot contain single quotes; double quotes must be escaped

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

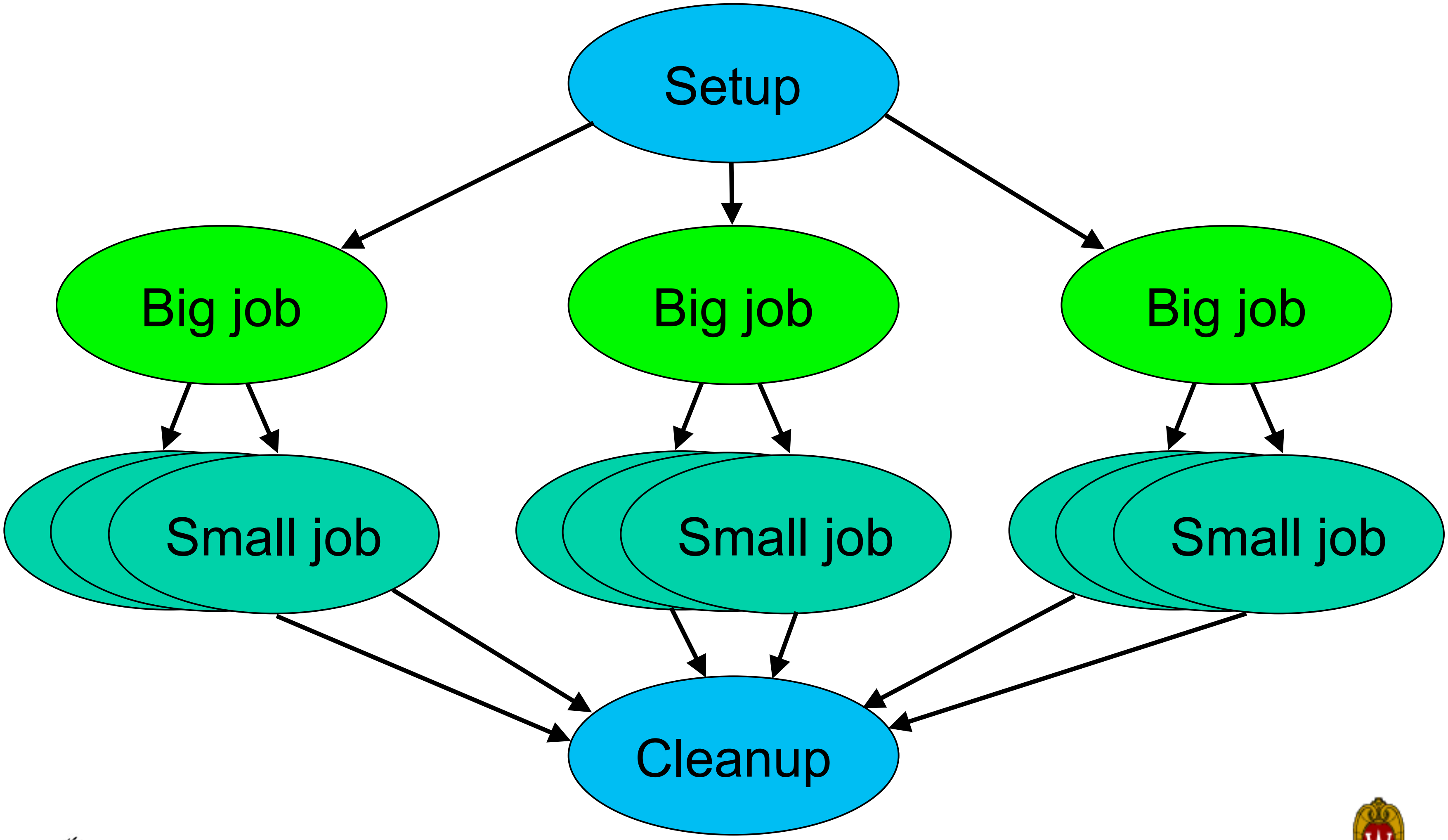
Node category throttles

- 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

Node categories example



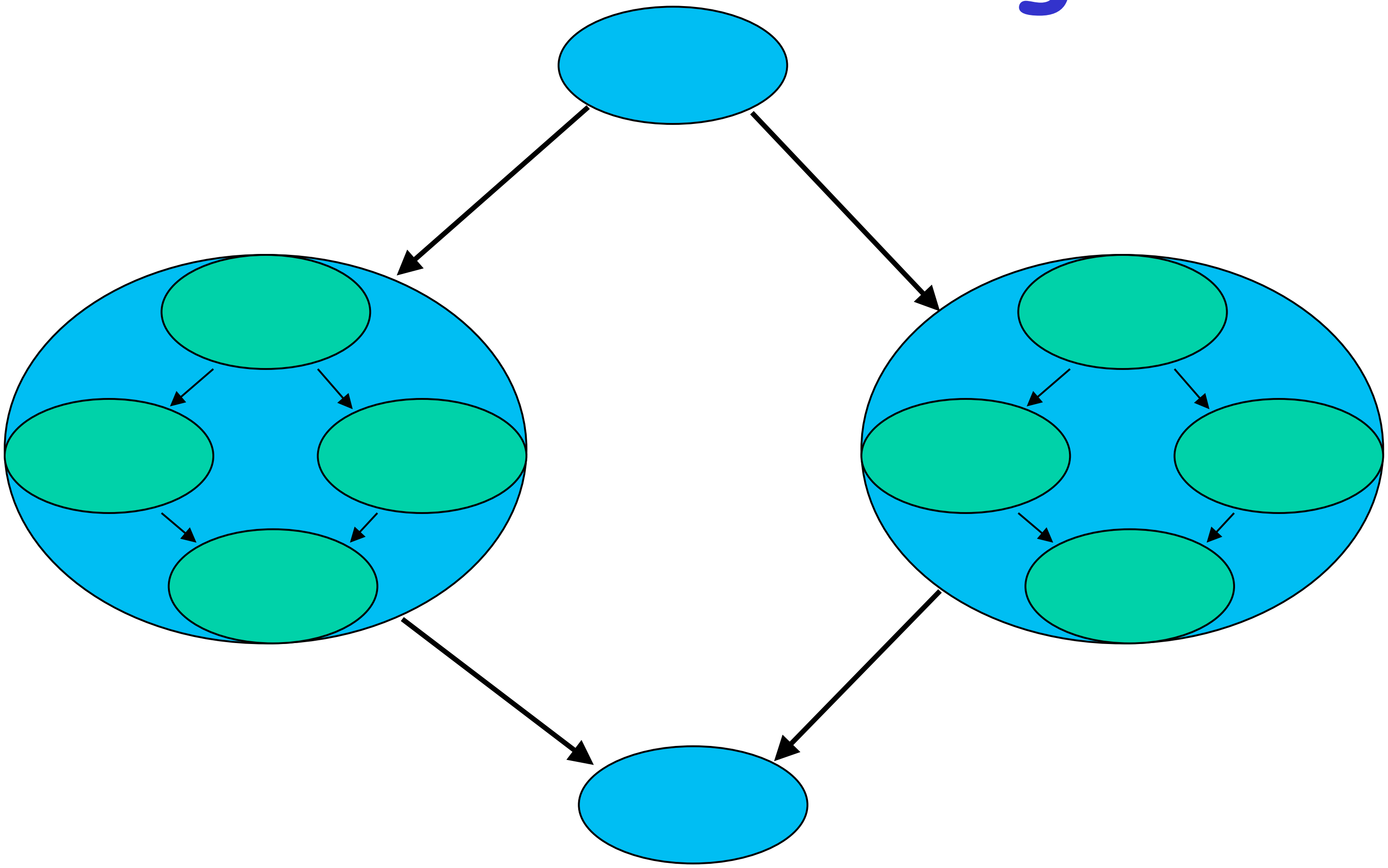
Nested DAGs

- 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
- 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

Nested DAGs diagram



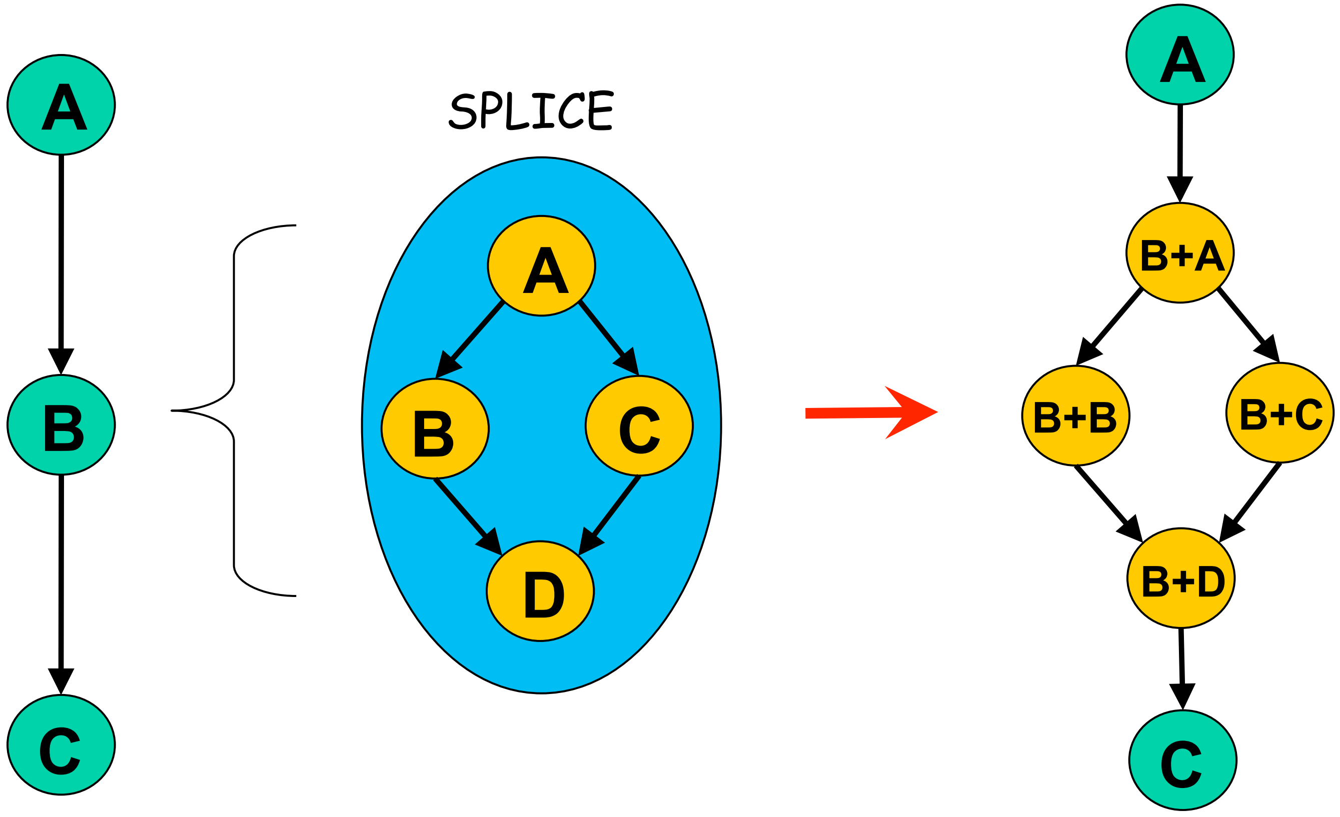
Splices

- Directly includes splice'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
- Since 7.1

Why splices?

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

Splice diagram



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:

```
ABORT-DAG-ON JobName AbortExitValue
    [RETURN DagReturnValue]
```

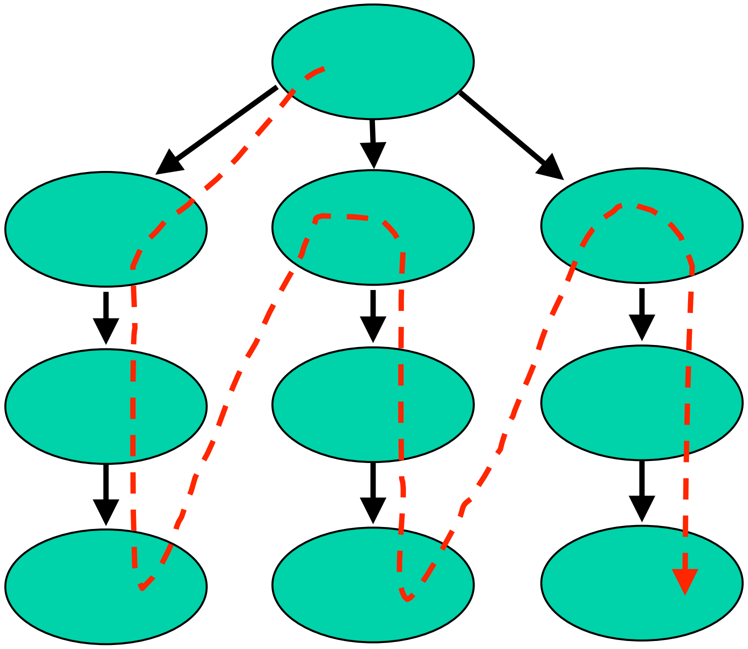
- If node value is **`AbortExitValue`**, the entire DAG is aborted, implying that jobs are removed, and a rescue DAG is created.
- Can be used for conditionally skipping nodes (especially with sub-DAGs)

Node priorities

- In the DAG input file:
  `PRIORITY JobName PriorityValue`
- Determines order of submission of ready nodes
- Does *not* violate or change DAG semantics
- Mostly useful when DAG is throttled
- Higher numerical value equals “better” priority

Depth-first DAG traversal

- Get some results more quickly
- Possibly clean up intermediate files more quickly
- **DAGMAN_SUBMIT_DEPTH_FIRST=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

new and improved

Cross-splice node categories

- Prefix category name with "+"

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

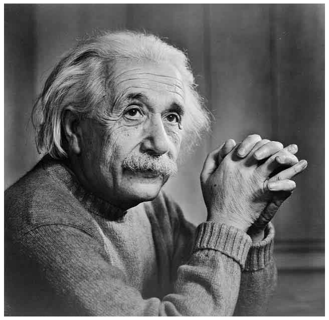
- See the Splice section in the manual for details
- New in 7.5.3

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
- Pegasus



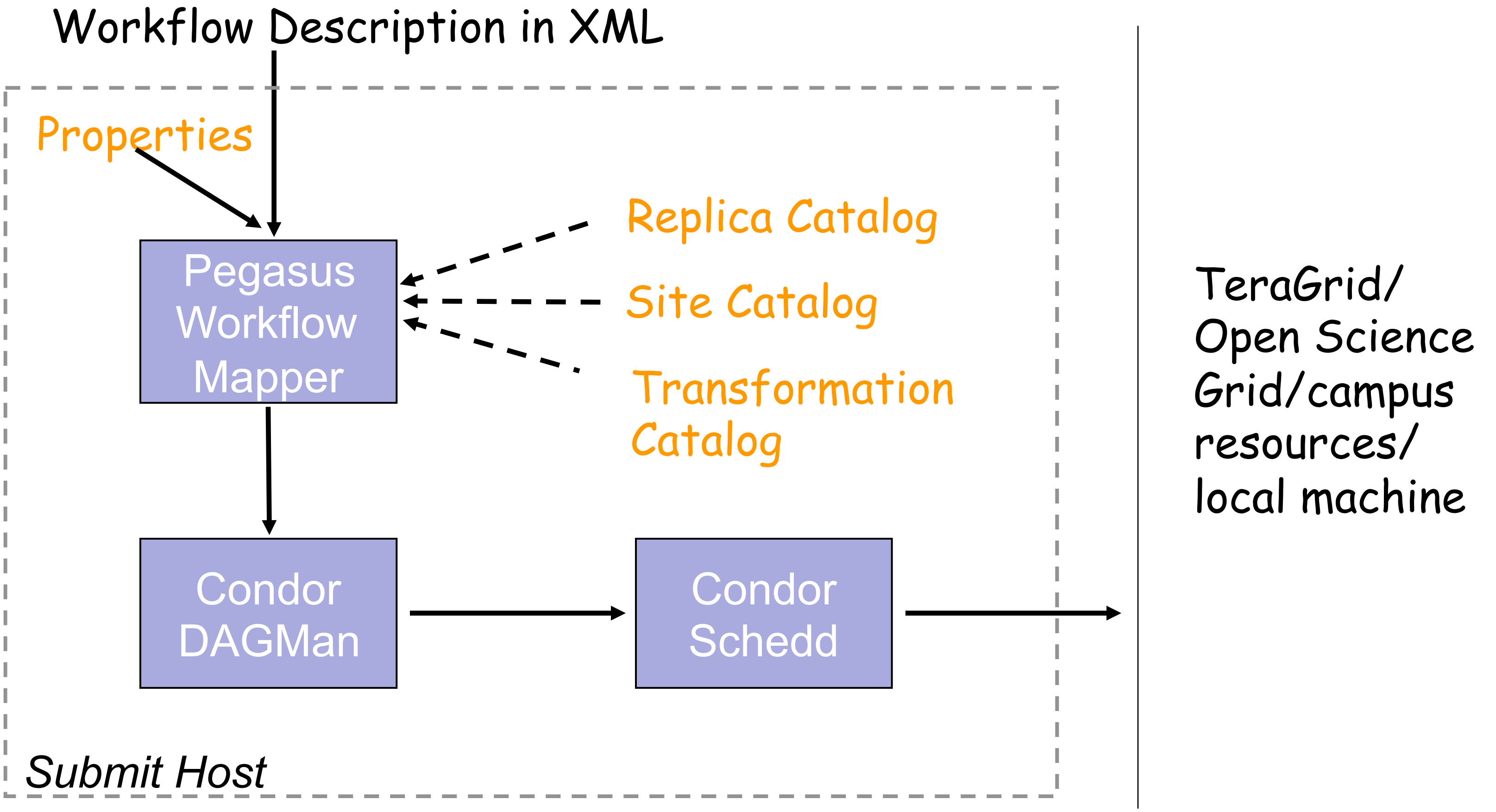
Albert meets Pegasus-WMS

- What if I want to define workflows that can flexibly take advantage of different grid resources?
- What if I want to register data products in a way that makes them available to others?
- What if I want to use the grid without a full Condor installation?

Pegasus Workflow Management System

- 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 (DAGMan/ schedd only)

Pegasus WMS



Pegasus WMS restructures and optimizes the workflow, moves data, provides reliability

Pegasus features

- Workflow has inter-job dependencies (similar to DAGMan)
- Pegasus can map jobs to grid sites
- Pegasus handles discovery and registration of data products
- Pegasus handles data transfer to/ from grid sites

Abstract workflow (DAX)

- Pegasus workflow description—DAX
  - Workflow "high-level language"
  - Devoid of resource descriptions
  - Devoid of data locations
  - Refers to codes as logical transformations
  - Refers to data as logical files

DAX example

```
<!-- part 1: list of all files used (may be empty) -->
  <filename file="f.input" link="input"/>
  . . .
<!-- part 2: definition of all jobs (at least one) -->
  <job id="ID000001" namespace=”pegasus" name="preprocess" version="1.0" >
                <argument>-a top -T 6  -i <filename file=”f.input"/>  -o <filename
file=”f.intermediate"/>
                </argument>
                <uses file=”f.input" link="input" register="false" transfer=”true"/>
                <uses file=”f.intermediate" link="output" register=”false" transfer=“false”>
                <!-- specify any extra executables the job needs . Optional  -->
                <uses file=“keg” link=“input” register=“false” transfer=“true”
type=“executable”>
  </job>
  . . .
<!-- part 3: list of control-flow dependencies (empty for single jobs) -->
 <child ref="ID000002">
    <parent ref="ID000001"/>
  </child>
```

(excerpted for display)

Basic workflow mapping

- Select where to run the computations
  - Change task nodes into nodes with executable descriptions
- Select which data to access
  - Add stage-in and stage-out nodes to move data
- Add nodes that register the newly-created data products
- Add nodes to create an execution directory on a remote site
- Write out the workflow in a form understandable by a workflow engine
  - Include provenance capture steps

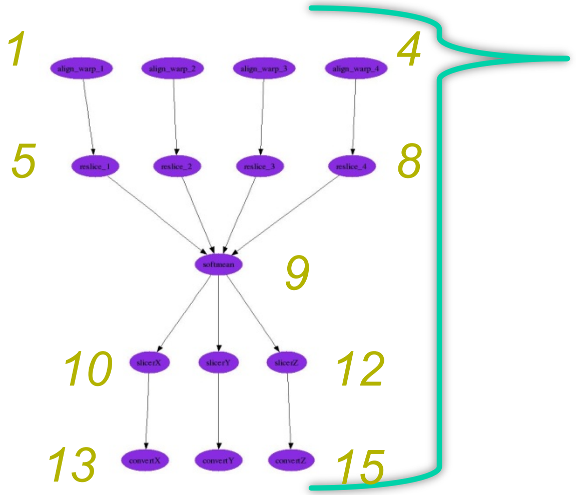
Mapping a workflow

- To map a workflow, use the ***pegasus-plan*** command:

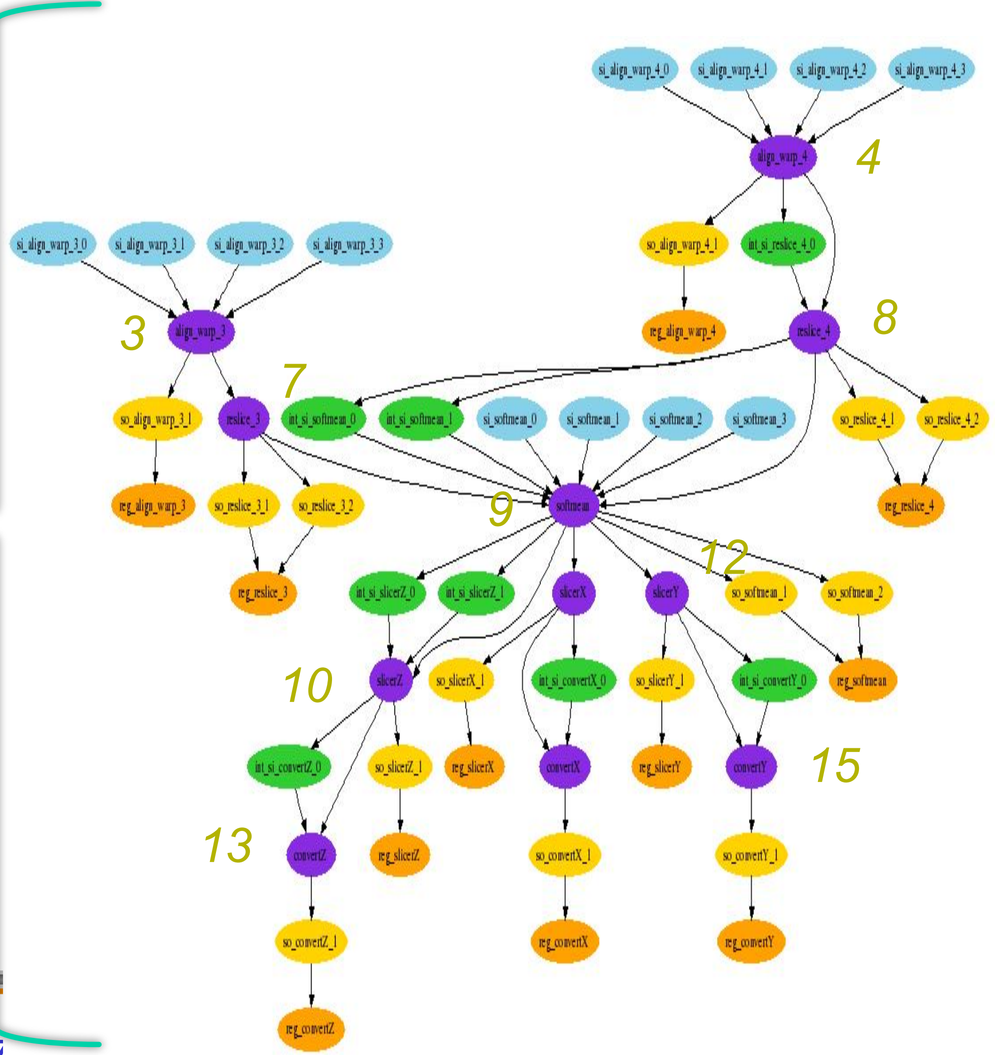
```
pegasus-plan
  -Dpegasus.user.properties=pegasus-
  wms/config/properties --dir dags --
  sites viz --output local --force --
  nocleanup --dax pegasus-wms/dax/
  montage.dax
```

- Creates executable workflow

Pegasus workflow mapping



Original workflow: 15 compute nodes devoid of resource assignment



Resulting workflow mapped onto 3 Grid sites:

11 compute nodes (4 reduced based on available intermediate data)

13 data stage-in nodes

8 inter-site data transfers

14 data stage-out nodes to long-term storage

14 data registration nodes (data cataloging)

Running a workflow

- To run a workflow, use the ***pegasus-run*** command:

  ```
  pegasus-run
    -Dpegasus.user.properties=pegasus-
    wms/dags/train01/pegasus/montage/
    run0001/pegasus.51773.properties
    pegasus-wms/dags/train01/pegasus/
    montage/run0001
  ```

- Runs *condor_submit_dag* and other tools
- Pegasus-plan gives you the pegasus-run command you need

There’s much more…

- We’ve only scratched the surface of Pegasus’s capabilities

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