Workflows with HTCondor's DAGMan

Thursday, Lecture 4

Lauren Michael

Questions so far?

Open Science Grid

Goals for this Session

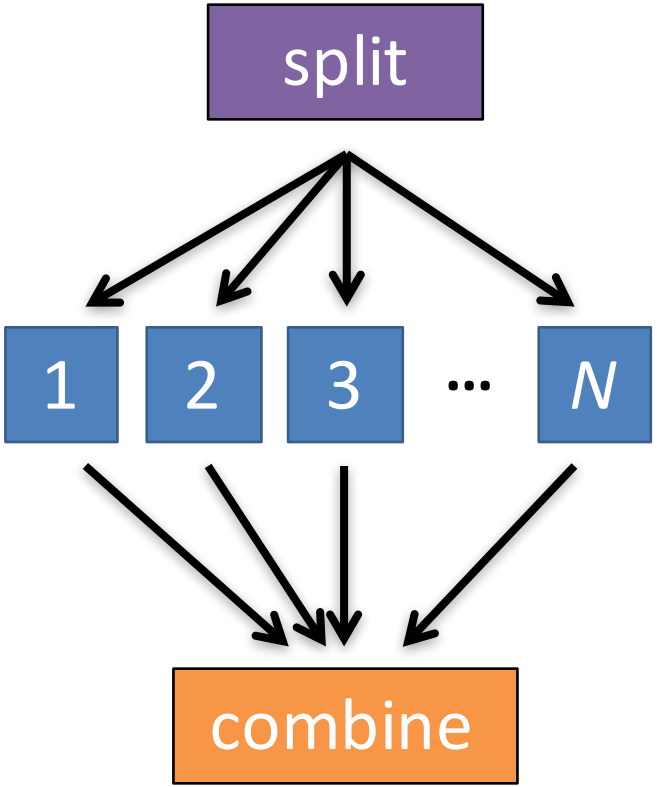
- Describing workflows as *directed acyclic graphs* (DAGs)
- Workflow execution via DAGMan (DAG Manager)
- Node-level options in a DAG
- Modular organization of DAG components
- Additional DAGMan Features

WHY WORKFLOWS?
WHY DAGS?

Automation!

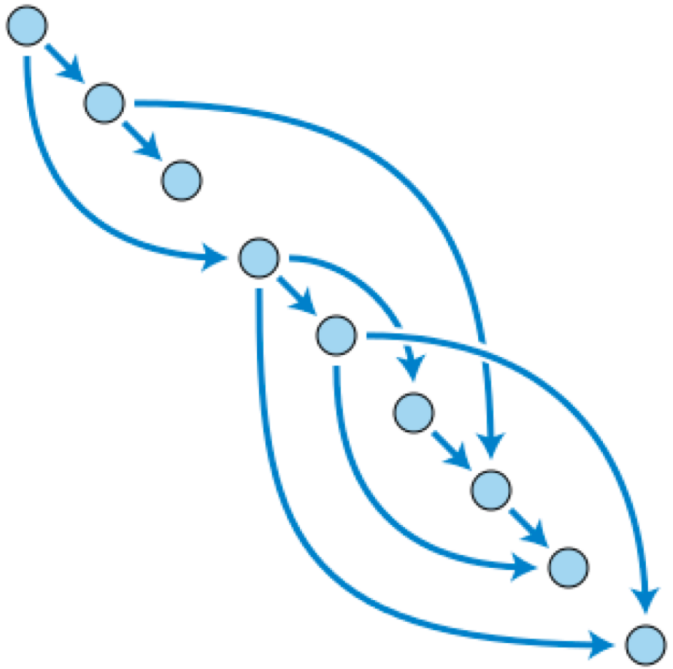
- Objective: Submit jobs **in a particular order**, *automatically*.

- 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



Wikimedia Commons

wikipedia.org/wiki/Directed_acyclic_graph

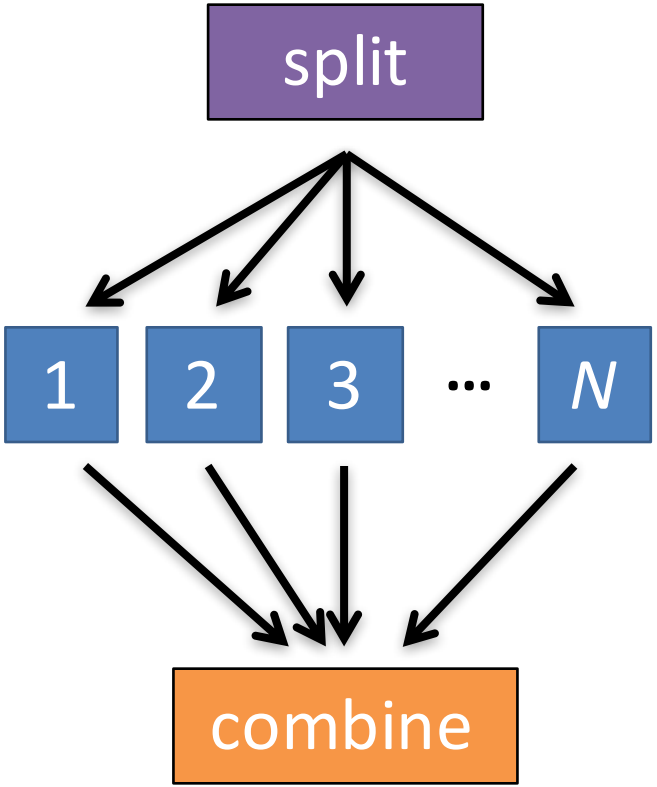
DESCRIBING WORKFLOWS WITH DAGMAN

DAGMan in the HTCondor Manual

Secure https://research.cs.wisc.edu/htcondor/manual/current/2_Users_Manual.html

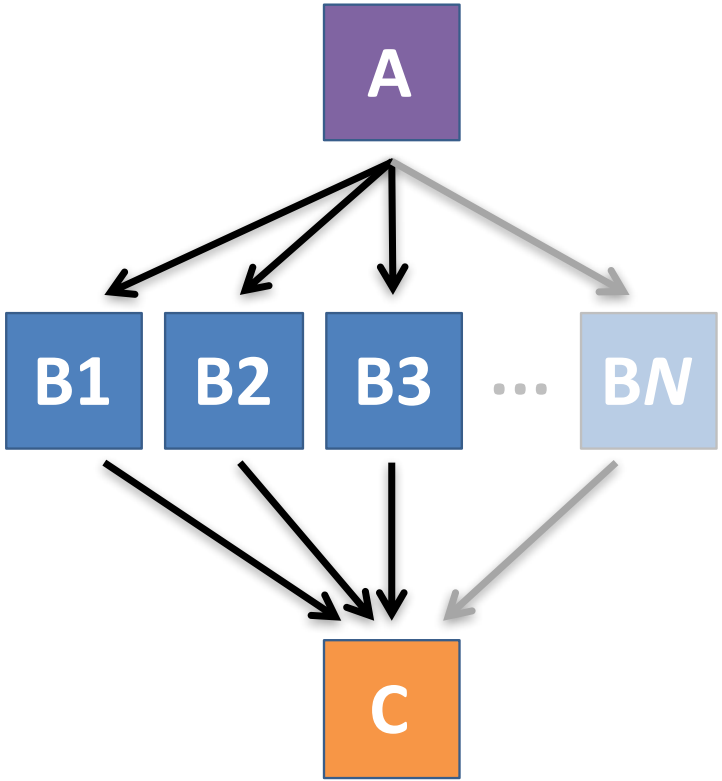
An Example HTC Workflow

- User must communicate the "nodes" and directional "edges" of the DAG



Simple Example for this Tutorial

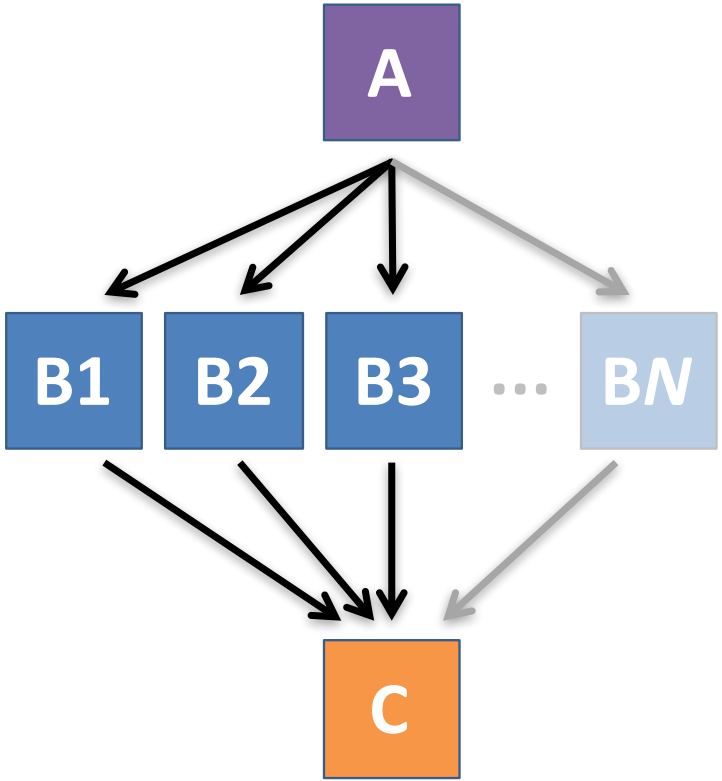
- **The DAG input file will** communicate the "nodes" and directional "edges" of the DAG



HTCondor Manual: DAGMan Applications > DAG Input File

Simple Example for this Tutorial

- **The DAG input file will** communicate the "nodes" and directional "edges" of the DAG



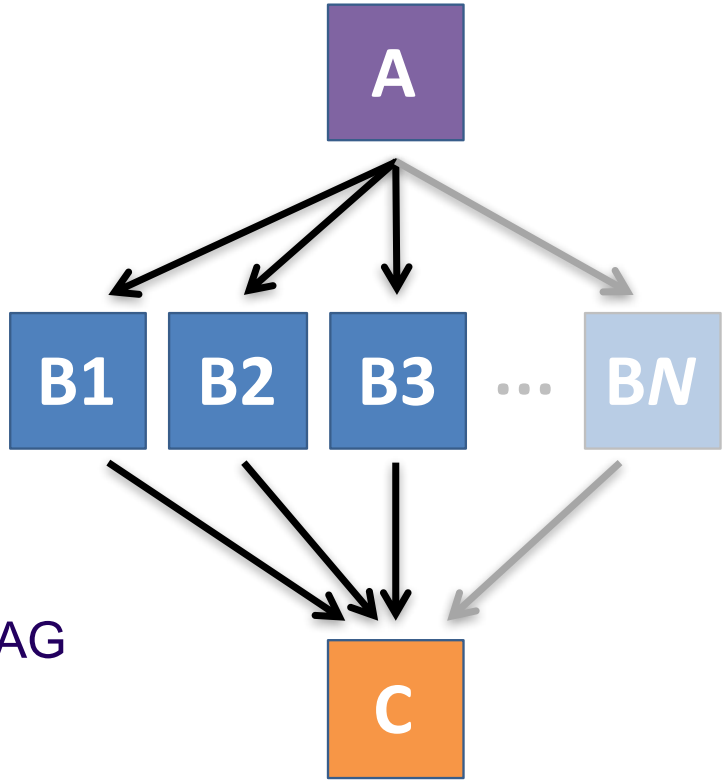
Look for links on future slides

HTCondor Manual: DAGMan Applications > DAG Input File

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
```



- Node names are used by various DAG features to modify their execution by DAG Manager.

HTCondor Manual: DAGMan Applications > DAG Input File

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
```

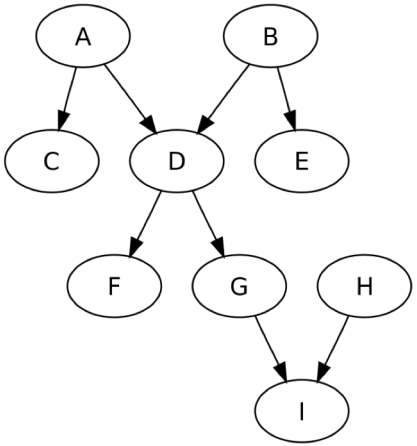
(dag_dir)/

```
A.sub          B1.sub
B2.sub         B3.sub
C.sub          my.dag
(other job files)
```

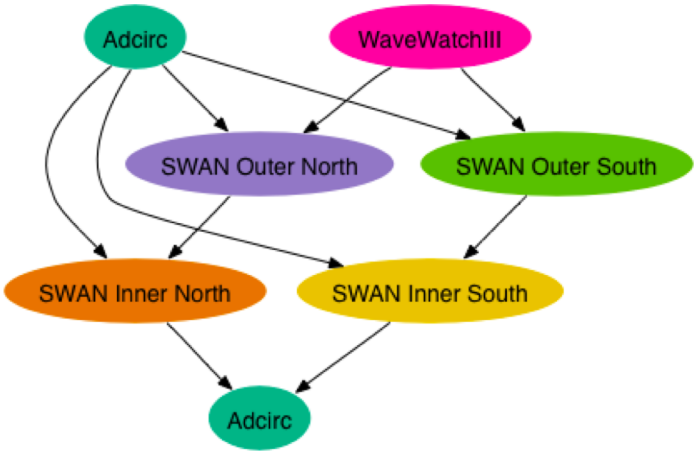
- Node names and filenames can be anything.
- Node name and submit filename do not have to match.

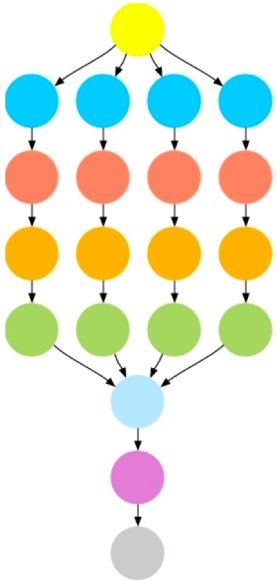
HTCondor Manual: DAGMan Applications > DAG Input File

Endless Workflow Possibilities



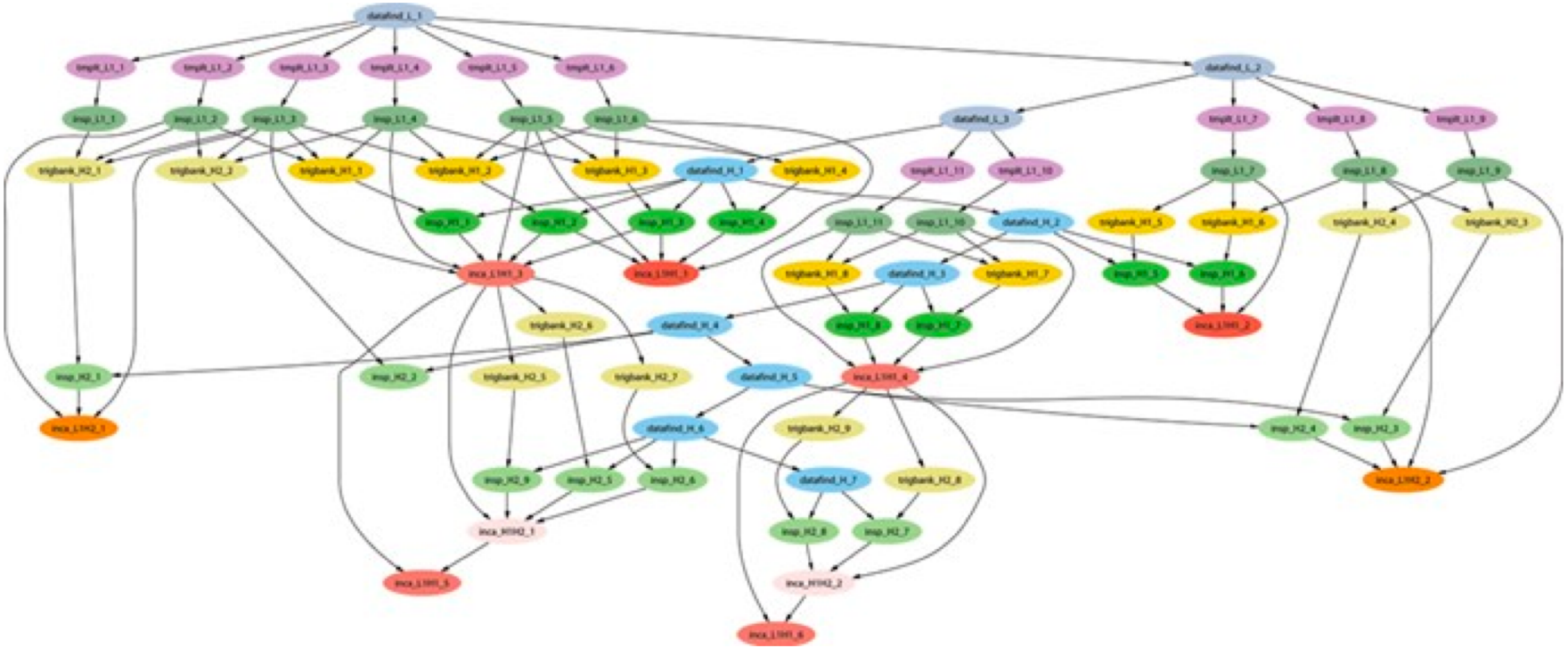
Wikimedia Commons



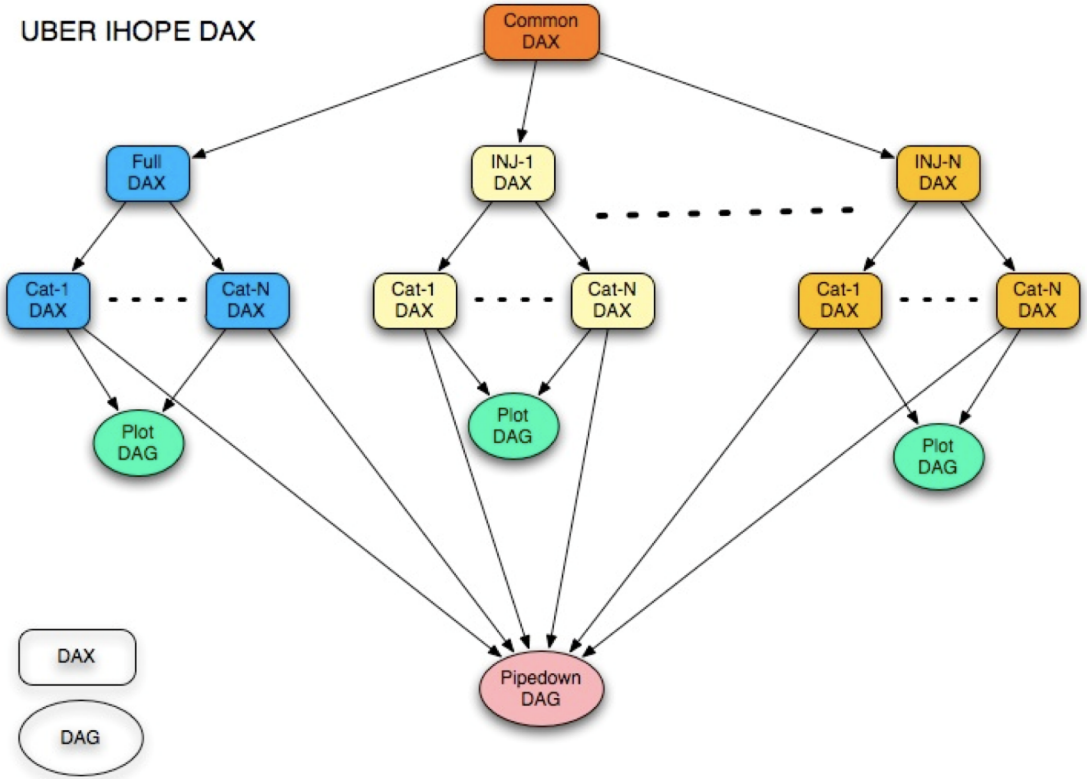


https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator

Endless Workflow Possibilities



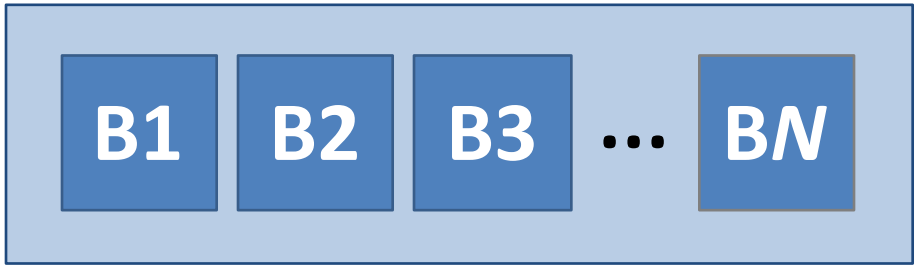
Repeating DAG Components!!



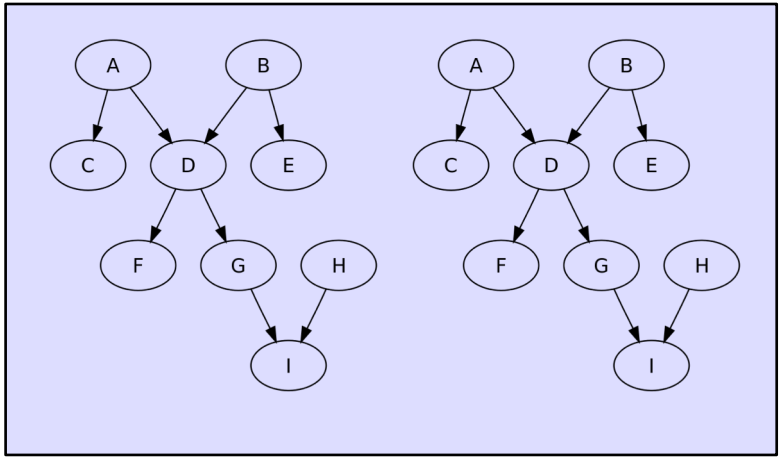
https://confluence.pegasus.isi.edu/display/pegasus/LIGO+IHOPE

DAGs are also useful for non-sequential work

‘bag’ of HTC jobs



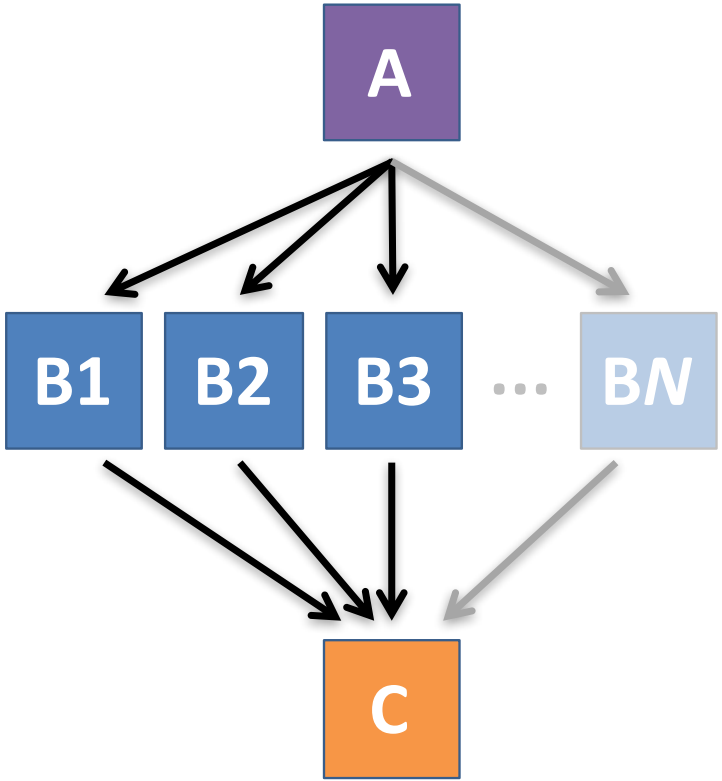
disjointed workflows



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 87274940.
------------------------------------------------------------------------
```


HTCondor Manual: DAGMan > DAG Submission

A submitted DAG creates and 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
```


HTCondor Manual: DAGMan > DAG Submission

Jobs are automatically submitted by the DAGMan job

- Seconds later, node **A** 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     _      _      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
```


HTCondor Manual: DAGMan > DAG Submission

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_IDS
alice    my.dag+128   4/30 8:08      1     _      3       5   129.0...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?...
 ID      OWNER    SUBMITTED     RUN_TIME ST PRI SIZE CMD
128.0    alice   4/30 18:08   0+00:20:36 R  0    0.3 condor_dagman
130.0    alice   4/30 18:18   0+00:00:00 I  0    0.3 B_run.sh
131.0    alice   4/30 18:18   0+00:00:00 I  0    0.3 B_run.sh
132.0    alice   4/30 18:18   0+00:00:00 I  0    0.3 B_run.sh
4 jobs; 0 completed, 0 removed, 3 idle, 1 running, 0 held, 0 suspended
```


HTCondor Manual: DAGMan > DAG Submission

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 8:08      4     _      1       5   129.0...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
```


HTCondor Manual: DAGMan > DAG Submission

Status files are Created at the time of DAG submission

(dag_dir)/

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.nodes.log		

- ***.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

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

```
$ 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
```

DAGMan > DAG Monitoring and DAG Removal

DAGMan > The Rescue DAG

Removal of a DAG results in a *rescue file*

```
(dag_dir)/
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
```

- 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

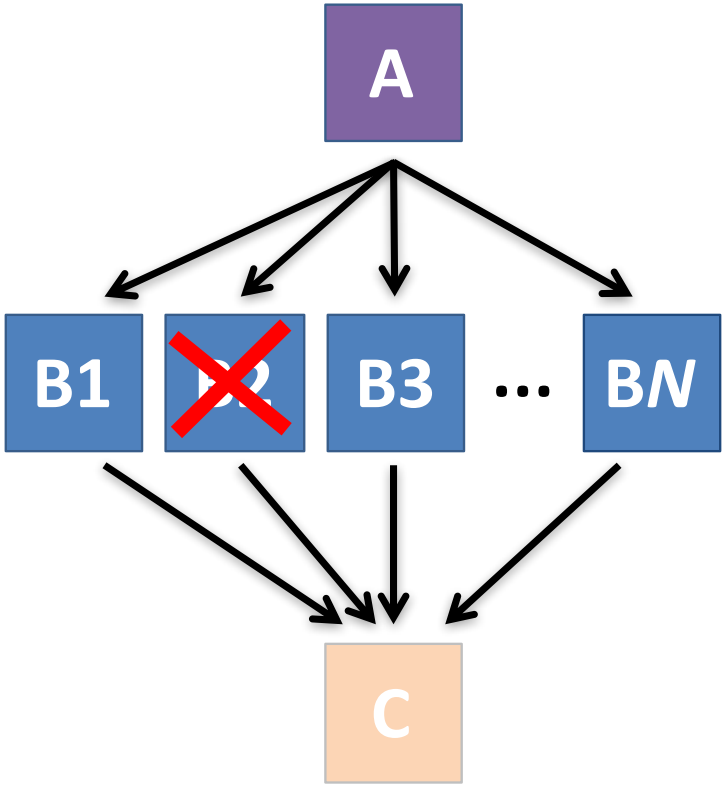
DAGMan > DAG Monitoring and DAG Removal
DAGMan > The Rescue DAG

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**; covered later)
  - the DAG is **halted** and not unhalted (covered later)
- 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**



DAGMan > The Rescue DAG

Resolving held node jobs

```
$ 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+00:20:36 R  0    0.3 condor_dagman
130.0   alice   4/30 18:18   0+00:00:00 H  0    0.3 B_run.sh
131.0   alice   4/30 18:18   0+00:00:00 H  0    0.3 B_run.sh
132.0   alice   4/30 18:18   0+00:00:00 H  0    0.3 B_run.sh
4 jobs; 0 completed, 0 removed, 0 idle, 1 running, 3 held, 0 suspended
```

- 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!)

HTCondor Manual: DAGMan > DAG Submission

DAG Completion

```
(dag_dir)/
```

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.nodes.log	**my.dag.dagman.metrics**	

- ***.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)

DAGMan > DAG Monitoring and DAG Removal

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?

HTCondor Manual: DAGMan Applications > DAG Input File

Node-specific File Organization with *DIR*

- **DIR** sets the submission directory of the node

my.dag

```
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
```

(dag_dir)/

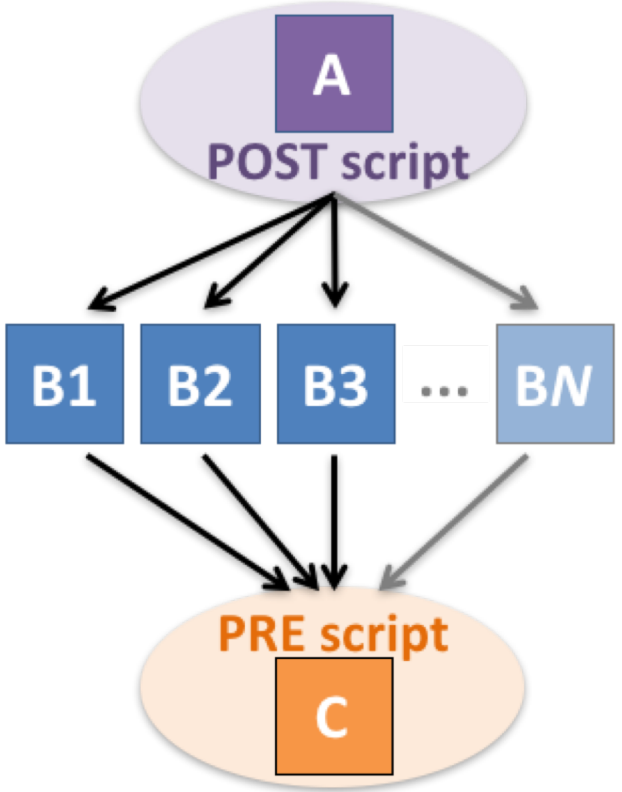
```
my.dag
A/      A.sub    (A job files)
B/      B1.sub   B2.sub
        B3.sub   (B job files)
C/      C.sub    (C job files)
```


HTCondor Manual: DAGMan Applications > DAG Input File

PRE and *POST* scripts run on the submit server, as part of the node

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**

HTCondor Manual: DAGMan Applications > DAG Input File

SCRIPT Arguments and Argument Variables

```
JOB A A.sub
SCRIPT POST A checkA.sh my.out $RETURN
RETRY A 5
```

$JOB: node name

$JOBID: *cluster.proc*

$RETURN: exit code of the node

$PRE_SCRIPT_RETURN: exit code of PRE script

$RETRY: current retry count

(more variables described in the manual)

DAGMan Applications > DAG Input File > SCRIPT
DAGMan Applications > Advanced Features > Retrying

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`)

DAGMan Applications > Advanced Features > Retrying
DAGMan Applications > DAG Input File > SCRIPT

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

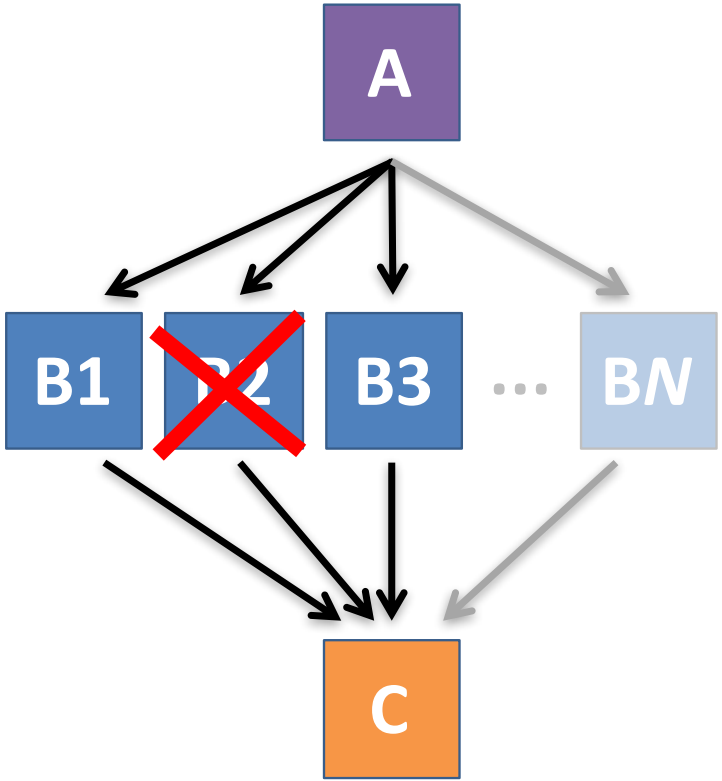
Example:

```
SCRIPT PRE A download.sh
JOB A A.sub
SCRIPT POST A checkA.sh
RETRY A 5
```

DAGMan Applications > Advanced Features > Retrying
DAGMan Applications > DAG Input File > SCRIPT

Best Control Achieved with One Process per JOB Node

- While submit files can ‘`queue`’ many processes, a *single process per submit* file is best for DAG JOBs
  - Failure of any process in a JOB node results in failure of the *entire node* and immediate removal of other processes in the node.
  - RETRY of a JOB node retries the entire submit file.



HTCondor Manual: DAGMan Applications > DAG Input File

Submit File Templates via *VARS*

- **VARS** line defines node-specific values that are passed into submit file variables

  VARS ***node_name var1="value" [var2="value"]***

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

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
```


DAGMan Applications > Advanced Features > Variable Values

MODULAR ORGANIZATION OF DAG COMPONENTS

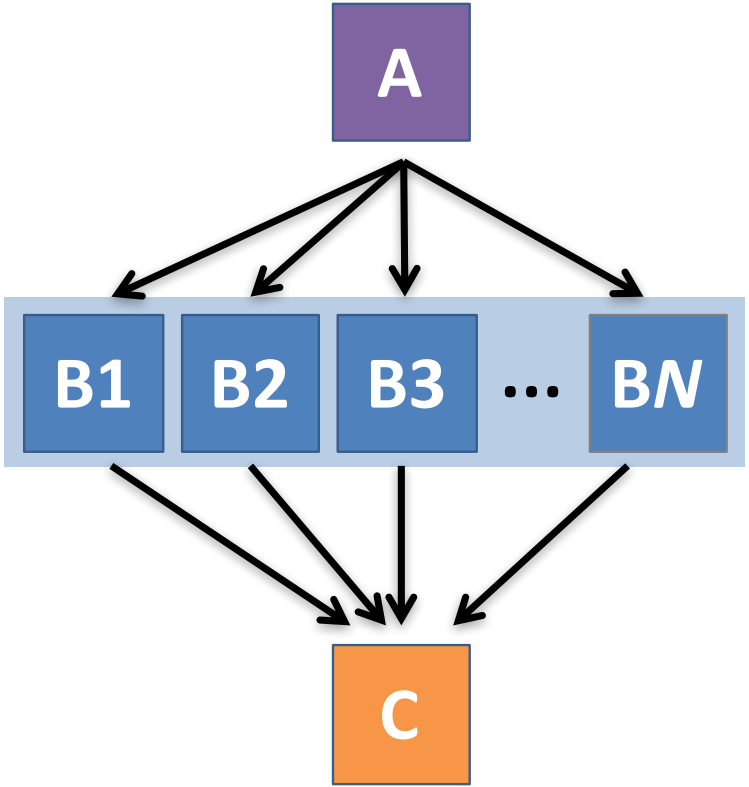
SPLICE groups of nodes to simplify lengthy DAG files

my.dag

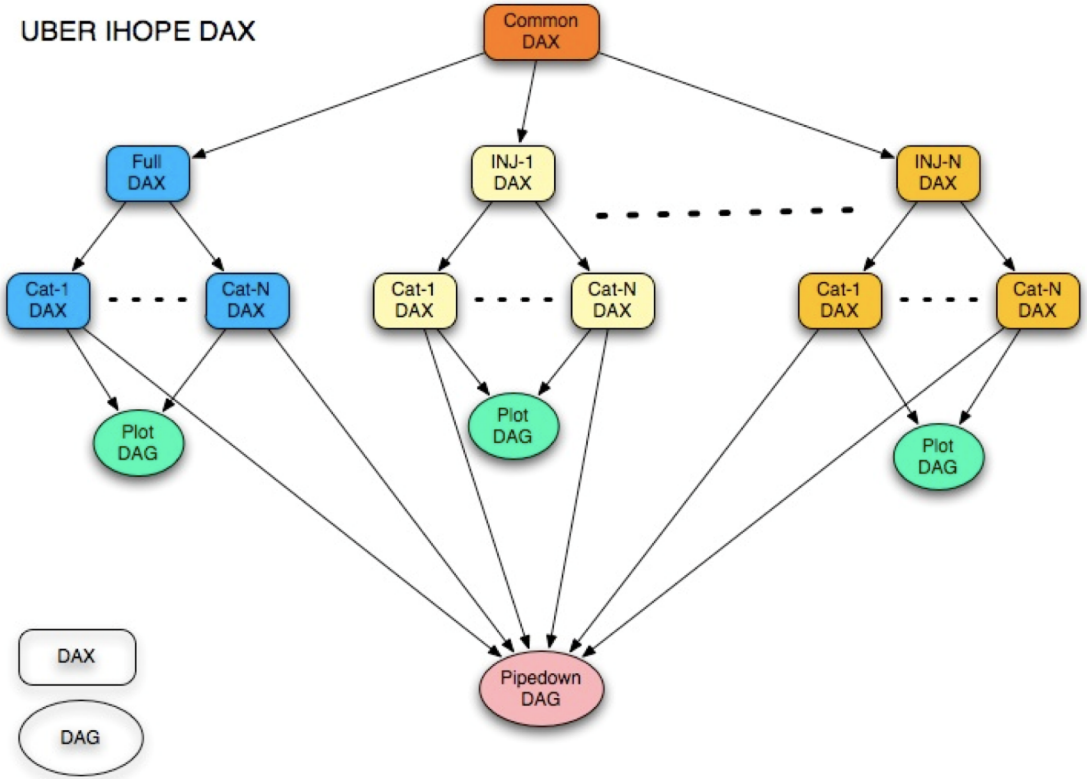
```
JOB A A.sub
SPLICE B B.spl
JOB C C.sub
PARENT A CHILD B
PARENT B CHILD C
```

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

Use nested *SPLICE*s with DIR for repeating workflow components

my.dag

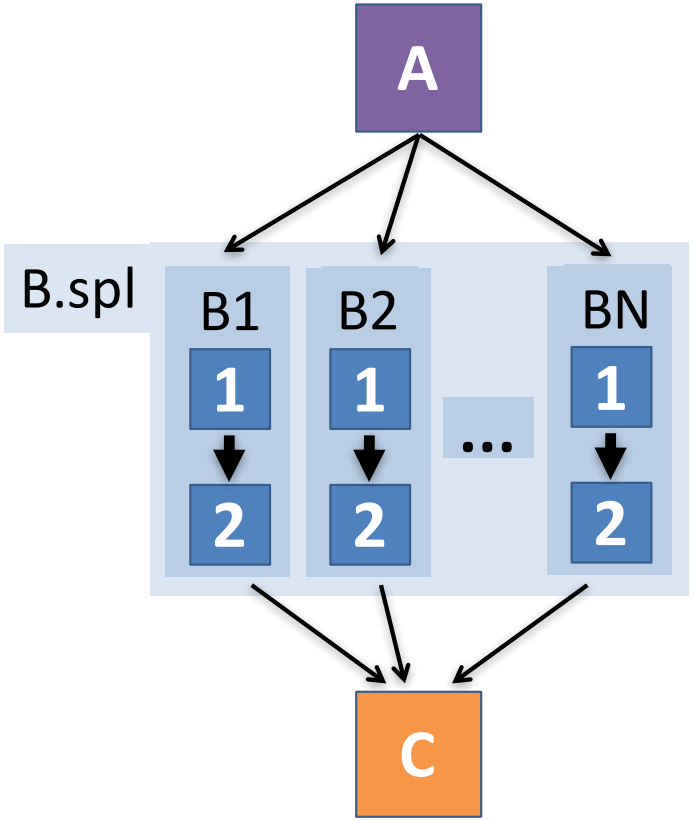
```
JOB A A.sub DIR A
SPLICE B B.spl DIR B
JOB C C.sub DIR C
PARENT A CHILD B
PARENT B CHILD C
```

B.spl

```
SPLICE B1 ../inner.spl DIR B1
SPLICE B2 ../inner.spl DIR B2
…
SPLICE BN ../inner.spl DIR BN
```

inner.spl

```
JOB 1 ../1.sub
JOB 2 ../2.sub
PARENT 1 CHILD 2
```



Use nested *SPLICE*s with DIR for repeating workflow components

my.dag

```
JOB A A.sub DIR A
SPLICE B B.spl DIR B
JOB C C.sub DIR C
PARENT A CHILD B
PARENT B CHILD C
```

B.spl

```
SPLICE B1 ../inner.spl DIR B1
SPLICE B2 ../inner.spl DIR B2
…
SPLICE BN ../inner.spl DIR BN
```

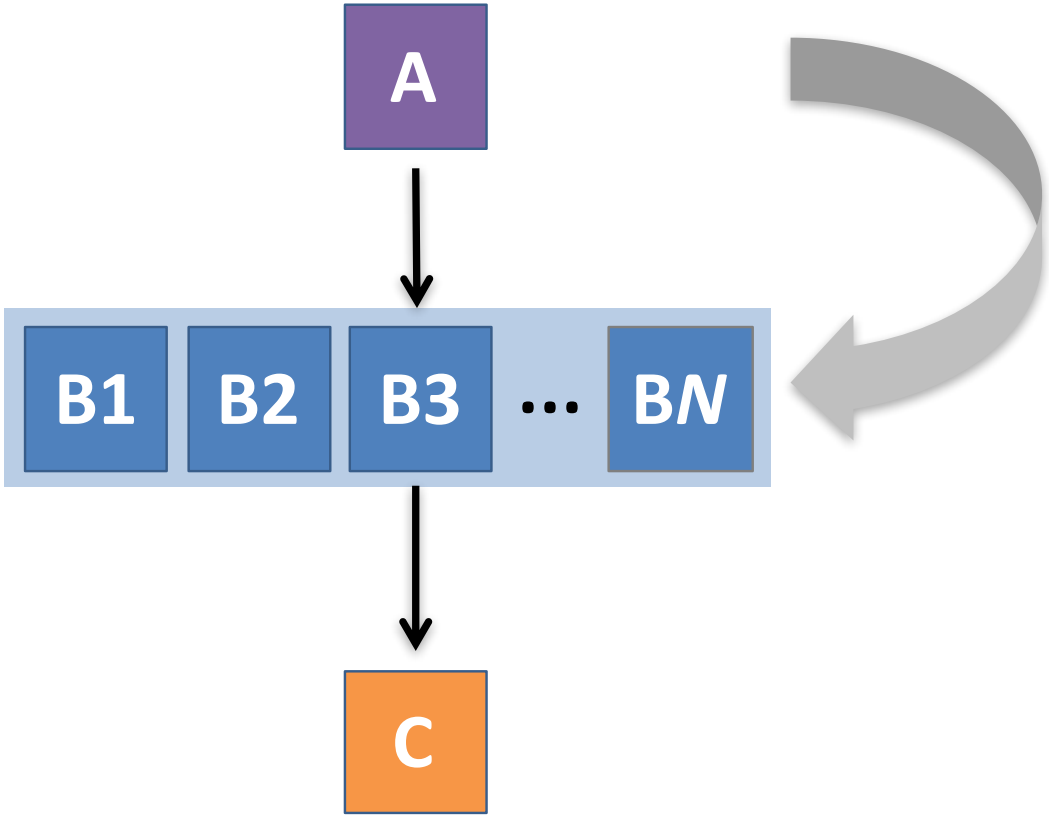
inner.spl

```
JOB 1 ../1.sub
JOB 2 ../2.sub
PARENT 1 CHILD 2
```

(dag_dir)/

```
my.dag
A/      A.sub (A job files)
B/      B.spl  inner.spl
        1.sub  2.sub
        B1/    (1-2 job files)
        B2/    (1-2 job files)
        …
        BN/    (1-2 job files)
C/      C.sub (C job files)
```

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** …

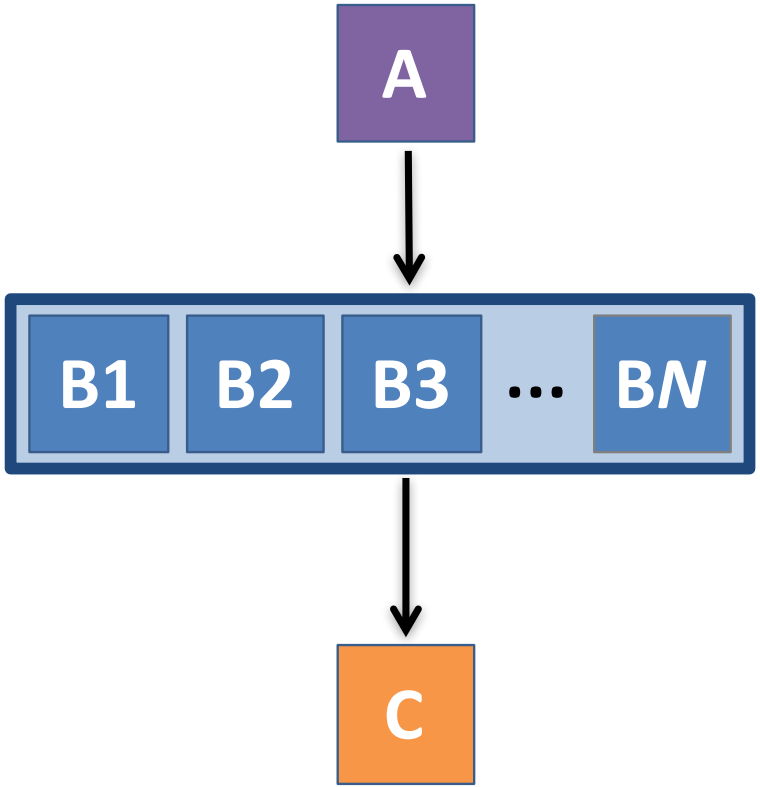
A *SUBDAG* within a DAG

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
```



Much More at the end of the presentation and in the HTCondor Manual!!!

https://research.cs.wisc.edu/htcondor/manual/current/2_Users_Manual.html

YOUR TURN!

DAGMan Exercises!

- Ask questions!
- Lots of instructors around

- Coming up:
  - now–5:00pm Hands-On Exercises
  - 5:00pm - on On Your Own

More on *SPLICE* Behavior

- Upon submission of the outer DAG, nodes in the SPLICE(s) are added by DAGMan into the overall DAG structure.
  - A single DAGMan job is queued with single set of status files.
- Great for gradually testing and building up a large DAG (since a SPLICE file can be submitted by itself, as a complete DAG).
- SPLICE lines are not treated like nodes.
  - no PRE/POST scripts or RETRIES (though this may change)

More on *SUBDAG* Behavior

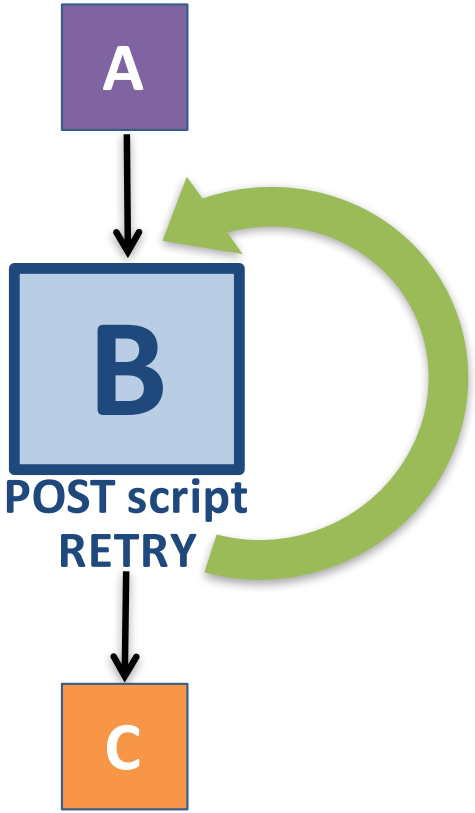
- **WARNING:** SUBDAGs should only be used (over SPLICES) when absolutely necessary!
  - *Each SUBDAG EXTERNAL has it's own DAGMan job running in the queue, on the submit server.*
- SUBDAGs *are nodes* in the outer DAG (can have PRE/POST scripts, retries, etc.)
- A SUBDAG is not submitted until prior nodes in the outer DAG have completed.

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

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
```



Other DAGMan Features

Other DAGMan Features: Node-Level Controls

- Set the **PRIORITY** of JOB nodes with:

```
PRIORITY node_name priority_value
```

- Use a **PRE_SKIP** to skip a node and mark it as successful, if the PRE script exits with a specific exit code:

```
PRE_SKIP node_name exit_code
```

DAGMan Applications > Advanced Features > Setting Priorities
DAGMan Applications > The DAG Input File > PRE_SKIP

Other DAGMan Features: Modular Control

- Append **NOOP** to a JOB definition so that its JOB process isn't run by DAGMan
  - Test DAG structure without running jobs (node-level)
  - Simplify combinatorial PARENT-CHILD statements (modular)
- Communicate DAG features separately with **INCLUDE**
  - e.g. separate file for JOB nodes and for VARS definitions, as part of the same DAG
- Define a **CATEGORY** to throttle only a specific subset of jobs

DAGMan Applications > The DAG Input File > JOB
DAGMan Applications > Advanced Features > INCLUDE
DAGMan Applications > Advanced > Throttling by Category

Other DAGMan Features: DAG-Level Controls

- Replace the ***node_name*** with **ALL_NODES** to apply a DAG feature to all nodes of the DAG

- Abort the entire DAG if a specific node exits with a specific exit code:

```
ABORT-DAG-ON node_name exit_code
```

- Define a **FINAL** node that will always run, even in the event of DAG failure (to clean up, perhaps).

```
FINAL node_name submit_file
```

DAGMan Applications > Advanced > ALL_NODES
DAGMan Applications > Advanced > Stopping the Entire DAG
DAGMan Applications > Advanced > FINAL Node