AN INTRODUCTION TO USING

Todd Tannenbaum

June 6, 2017

Covered In This Tutorial

- What is HTCondor?
- Running a Job with HTCondor
- How HTCondor Matches and Runs Jobs

  - pause for questions -
- Submitting Multiple Jobs with HTCondor
- Testing and Troubleshooting
- Use Cases and HTCondor Features
- Automation

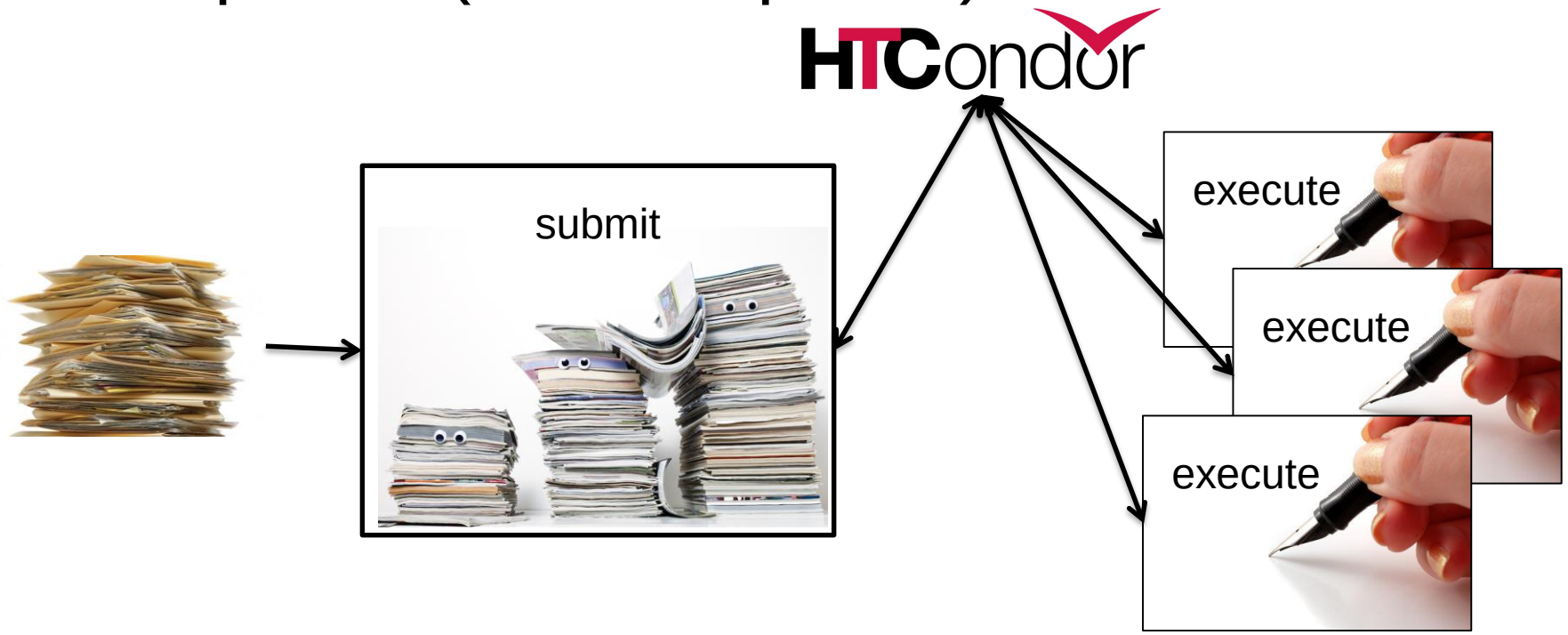
Introduction

What is HTCondor?

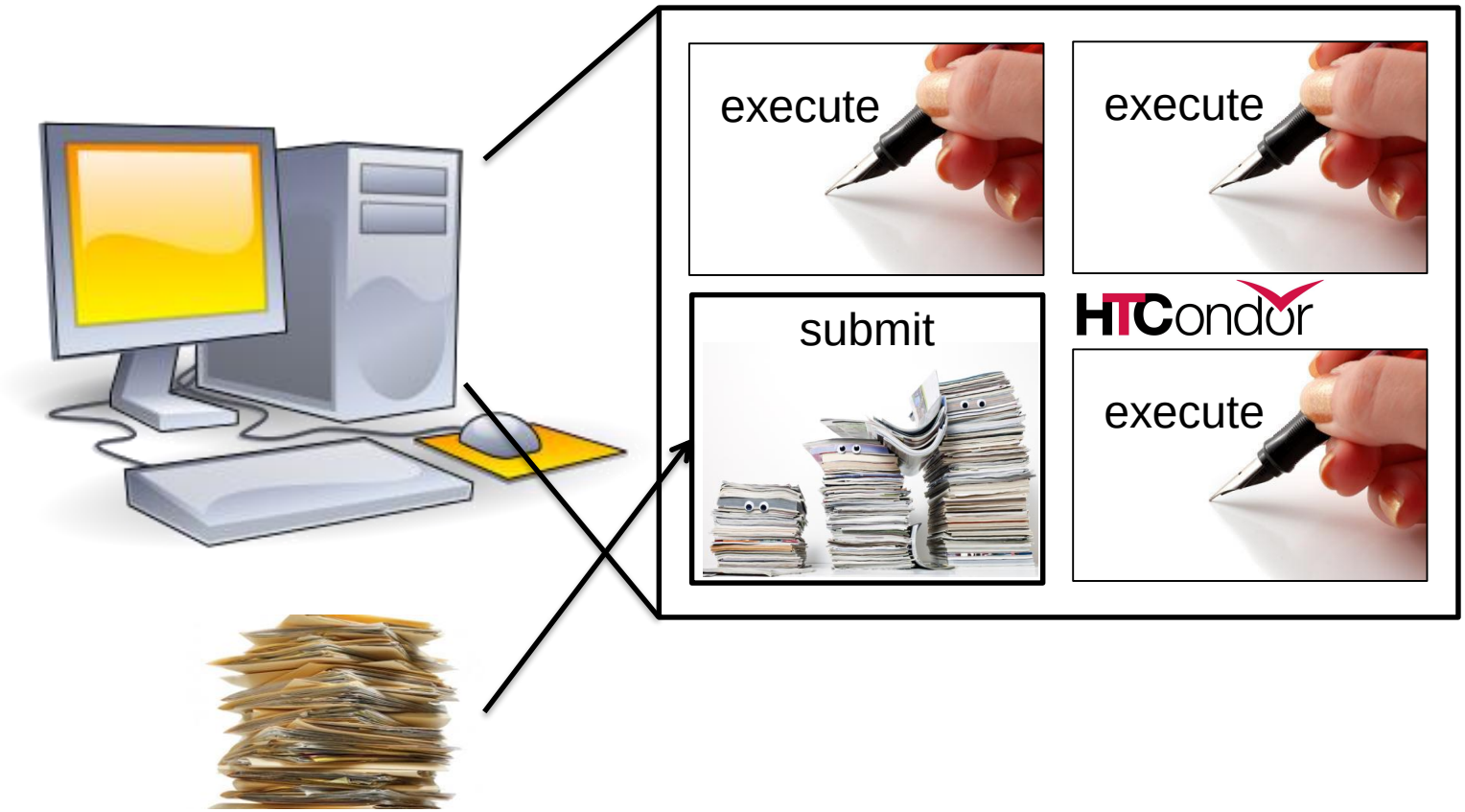
- Software that schedules and runs computing tasks on computers

How It Works

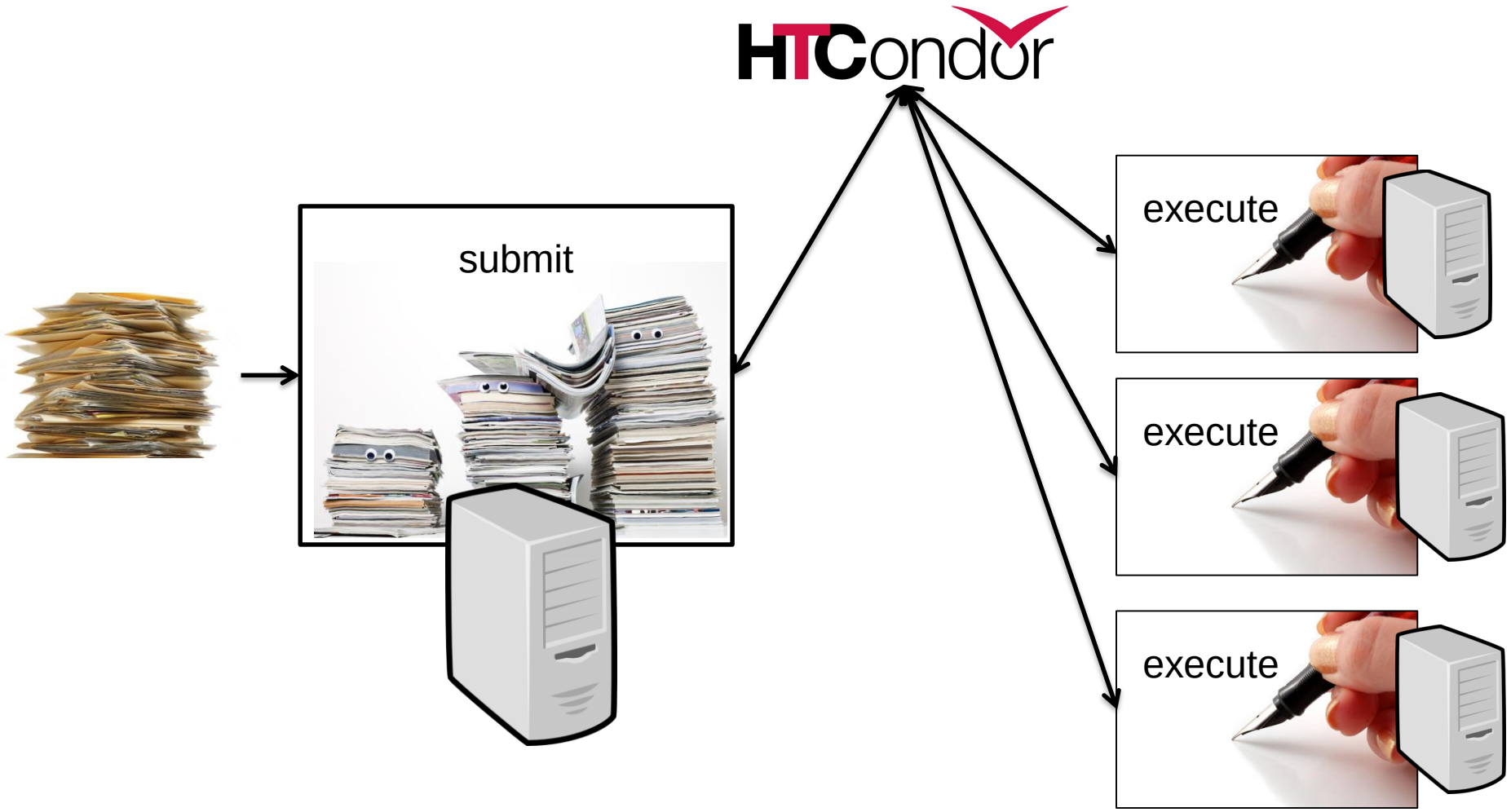
- Submit tasks to a queue (on a submit point)
- HTCondor schedules them to run on computers (execute points)



Single Computer



Multiple Computers



Why HTCondor?

- HTCondor manages and runs work on your behalf
- Schedule tasks on a single computer to not overwhelm the computer
- Schedule tasks on a group* of computers (which may/may not be directly accessible to the user)
- Schedule tasks submitted by multiple users on one or more computers

*in HTCondor-speak, a “pool”

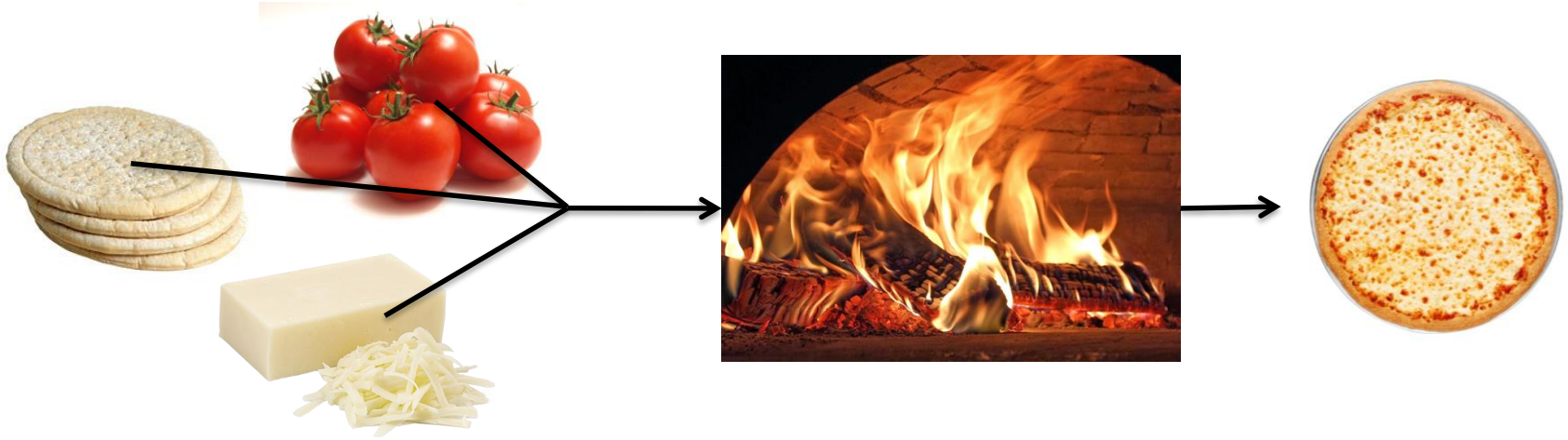
User-Focused Tutorial

- For the purposes of this tutorial, we are assuming that someone else has set up HTCondor on a computer/computers to create a HTCondor “pool”.
- The focus of this talk is an introduction on how to get started running computational work on this system.

Running a Job with HTCondor

Jobs

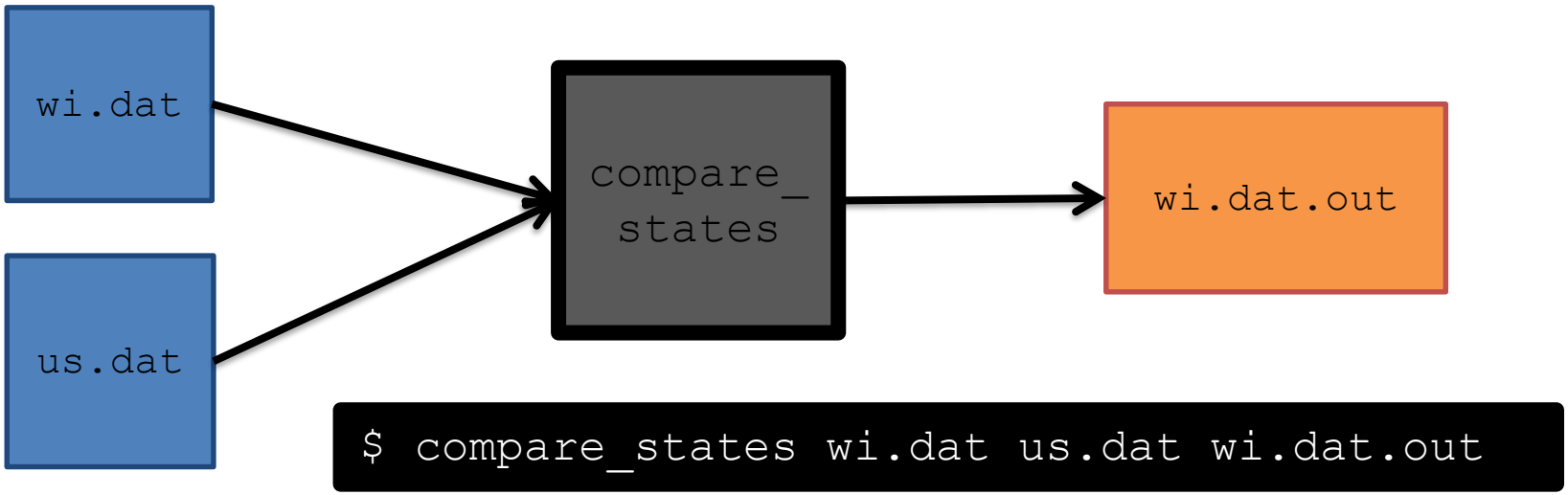
- A single computing task is called a “job”
- Three main pieces of a job are the input, executable (program) and output



- Executable must be runnable from the command line without any interactive input

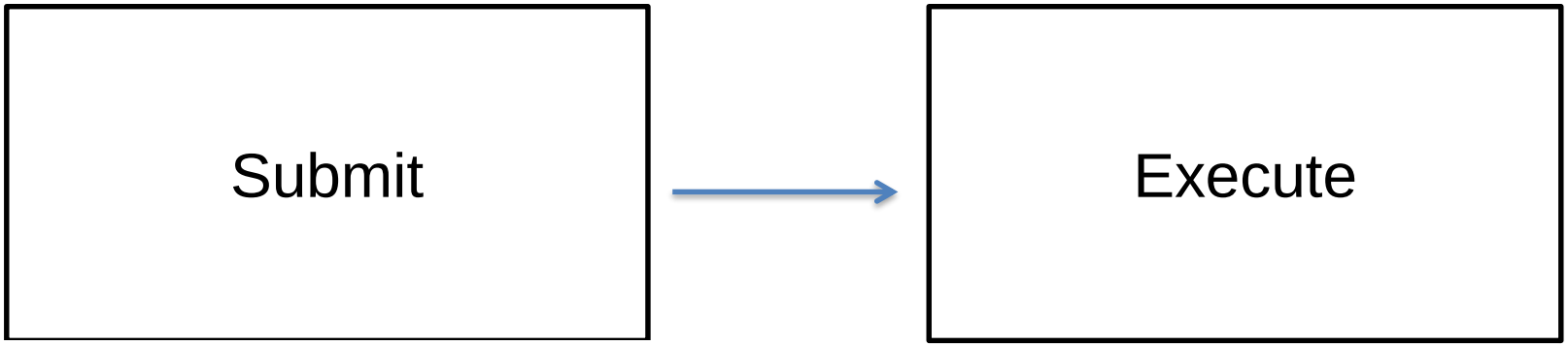
Job Example

- For our example, we will be using an imaginary program called "compare_states", which compares two data files and produces a single output file.



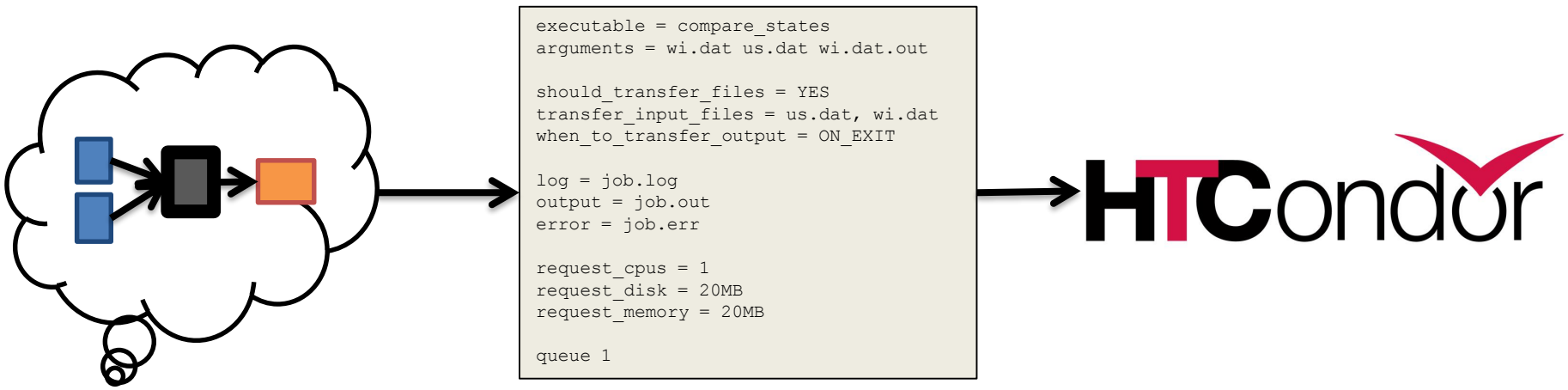
File Transfer

- What about files? Can use a shared file system, chirp, or file transfer mechanism.
- Our example will use HTCondor's file transfer :



Job Translation

- Submit file: communicates everything about your job(s) to HTCondor



```
executable = compare_states
arguments = wi.dat us.dat wi.dat.out

should_transfer_files = YES
transfer_input_files = us.dat, wi.dat
when_to_transfer_output = ON_EXIT

log = job.log
output = job.out
error = job.err

request_cpus = 1
request_disk = 20MB
request_memory = 20MB

queue 1
```

Submit File

job.submit

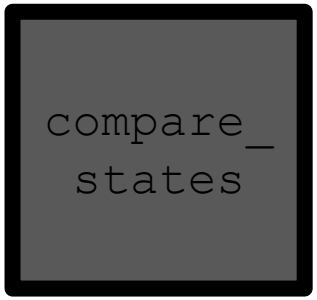
```
executable = compare_states
arguments = wi.dat us.dat wi.dat.out

should_transfer_files = YES
transfer_input_files = us.dat, wi.dat
when_to_transfer_output = ON_EXIT

log = job.log
output = job.out
error = job.err

request_cpus = 1
request_disk = 20MB
request_memory = 20MB

queue 1
```

Submit File

job.submit

```
executable = compare_states
arguments = wi.dat us.dat wi.dat.out

should_transfer_files = YES
transfer_input_files = us.dat, wi.dat
when_to_transfer_output = ON_EXIT

log = job.log
output = job.out
error = job.err

request_cpus = 1
request_disk = 20MB
request_memory = 20MB

queue 1
```

- List your executable and any arguments it takes.



- Arguments are any options passed to the executable from the command line.

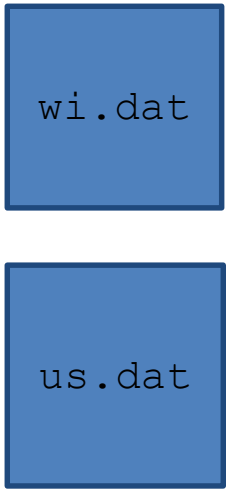
```
$ compare_states wi.dat us.dat wi.dat.out
```

Submit File

job.submit

```
executable = compare_states
arguments = wi.dat us.dat wi.dat.out

should_transfer_files = YES
transfer_input_files = us.dat, wi.dat
when_to_transfer_output = ON_EXIT

log = job.log
output = job.out
error = job.err

request_cpus = 1
request_disk = 20MB
request_memory = 20MB

queue 1
```

- Indicate your input files.

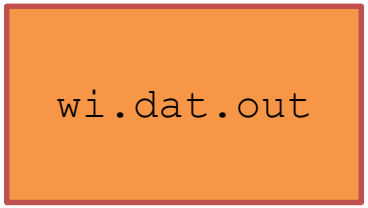


Submit File

job.submit

```
executable = compare_states
arguments = wi.dat us.dat wi.dat.out

should_transfer_files = YES
transfer_input_files = us.dat, wi.dat
when_to_transfer_output = ON_EXIT

log = job.log
output = job.out
error = job.err

request_cpus = 1
request_disk = 20MB
request_memory = 20MB

queue 1
```

- HTCondor will transfer back all new and changed files (usually output) from the job.



Submit File

job.submit

```
executable = compare_states
arguments = wi.dat us.dat wi.dat.out

should_transfer_files = YES
transfer_input_files = us.dat, wi.dat
when_to_transfer_output = ON_EXIT

log = job.log
output = job.out
error = job.err

request_cpus = 1
request_disk = 20MB
request_memory = 20MB

queue 1
```

- `log`: file created by HTCondor to track job progress
- `output/error`: captures stdout and stderr

Submit File

job.submit

```
executable = compare_states
arguments = wi.dat us.dat wi.dat.out

should_transfer_files = YES
transfer_input_files = us.dat, wi.dat
when_to_transfer_output = ON_EXIT

log = job.log
output = job.out
error = job.err

request_cpus = 1
request_disk = 20MB
request_memory = 20MB

queue 1
```

- Request the appropriate resources for your job to run.
- `queue`: keyword indicating "create a job."

Submitting and Monitoring

- To submit a job/jobs:

  `condor_submit` ***`submit_file_name`***

- To monitor submitted jobs, use:

  `condor_q`

```
$ condor_submit job.submit
Submitting job(s).
1 job(s) submitted to cluster 128.
```

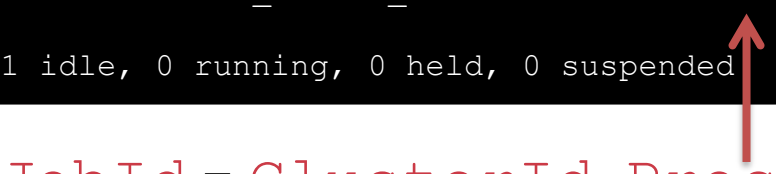
```
$ condor_q
-- Schedd: submit-5.chtc.wisc.edu : <128.104.101.92:9618?... @ 05/01/17 10:35:54
OWNER  BATCH_NAME              SUBMITTED   DONE   RUN    IDLE  TOTAL JOB_IDS
alice  CMD: compare_states   5/9  11:05      _      _      1      1 128.0

1 jobs; 0 completed, 0 removed, 1 idle, 0 running, 0 held, 0 suspended
```

HTCondor Manual: condor_submit
HTCondor Manual: condor_q

More about `condor_q`

- By default **`condor_q`** shows:
  - user's job only (as of 8.6)
  - jobs summarized in "batches" (as of 8.6)
- Constrain with username, `ClusterId` or full `JobId`, which will be denoted `[U/C/J]` in the following slides

```
$ condor_q
-- Schedd: submit-5.chtc.wisc.edu : <128.104.101.92:9618?... @ 05/01/17 10:35:54
OWNER  BATCH_NAME              SUBMITTED   DONE   RUN    IDLE  TOTAL JOB_IDS
alice  CMD: compare_states   5/9  11:05      _      _      1      1 128.0

1 jobs; 0 completed, 0 removed, 1 idle, 0 running, 0 held, 0 suspended
```



More about `condor_q`

- To see individual job information, use:

  `condor_q -nobatch`

```
$ condor_q -nobatch
-- Schedd: submit-5.chtc.wisc.edu : <128.104.101.92:9618?...
 ID          OWNER      SUBMITTED     RUN_TIME ST PRI SIZE CMD
128.0        alice      5/9  11:09   0+00:00:00 I  0    0.0 compare_states wi.dat us.dat

1 jobs; 0 completed, 0 removed, 1 idle, 0 running, 0 held, 0 suspended
```

- We will use the `-nobatch` option in the following slides to see extra detail about what is happening with a job

Job Idle

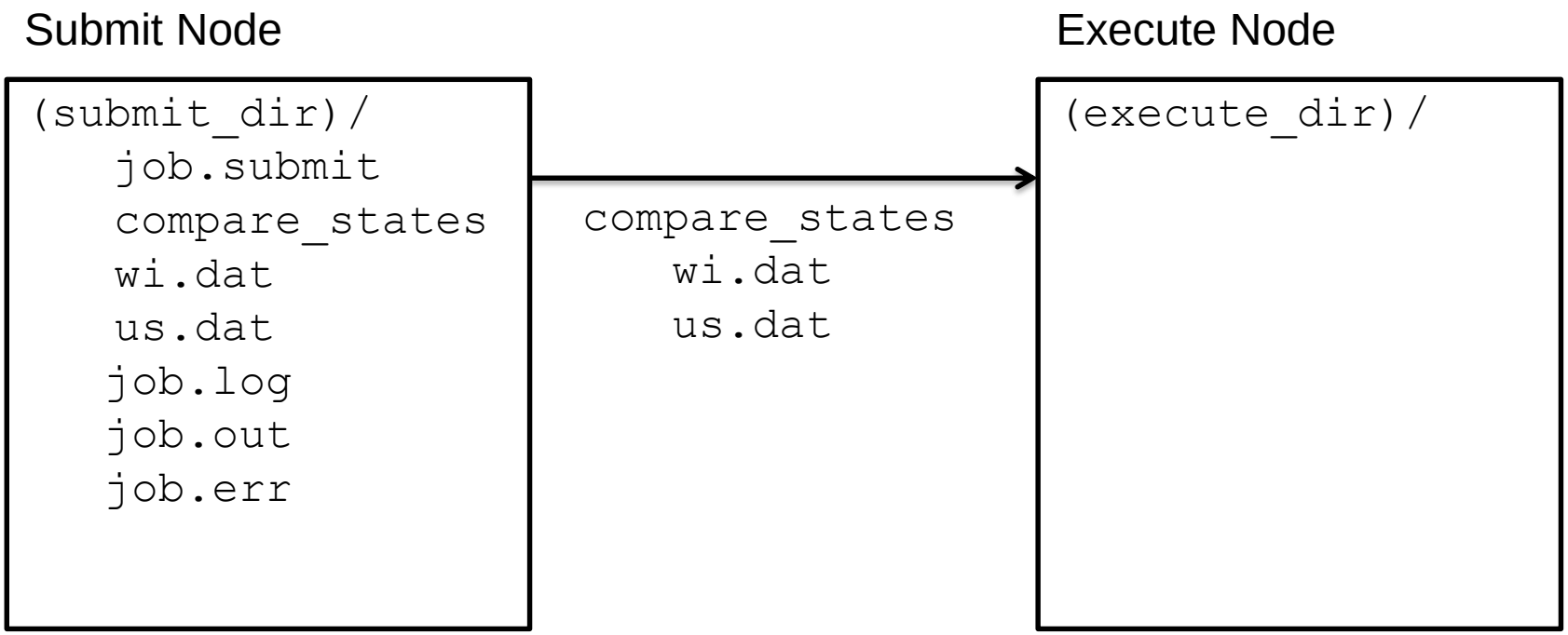
```
$ condor_q - nobatch
-- Schedd: submit-5.chtc.wisc.edu : <128.104.101.92:9618?...
 ID          OWNER      SUBMITTED     RUN_TIME ST PRI SIZE CMD
128.0        alice      5/9  11:09   0+00:00:00 I  0    0.0 compare_states wi.dat us.dat

1 jobs; 0 completed, 0 removed, 1 idle, 0 running, 0 held, 0 suspended
```

Submit Node

```
(submit_dir)/
    job.submit
    compare_states
    wi.dat
    us.dat
    job.log
    job.out
    job.err
```

Job Starts by doing File Transfer

```
$ condor_q -nobatch
-- Schedd: submit-5.chtc.wisc.edu : <128.104.101.92:9618?...
 ID          OWNER      SUBMITTED     RUN_TIME ST PRI SIZE CMD
128.0        alice      5/9  11:09   0+00:00:00 <  0    0.0 compare_states wi.dat us.dat w

1 jobs; 0 completed, 0 removed, 0 idle, 1 running, 0 held, 0 suspended
```



Job Running

```
$ condor_q -nobatch

-- Schedd: submit-5.chtc.wisc.edu : <128.104.101.92:9618?...
 ID          OWNER       SUBMITTED     RUN_TIME ST PRI SIZE CMD
128.0        alice       5/9  11:09   0+00:01:03 R  0    0.0 compare_states wi.dat us.dat

1 jobs; 0 completed, 0 removed, 0 idle, 1 running, 0 held, 0 suspended
```

Submit Node

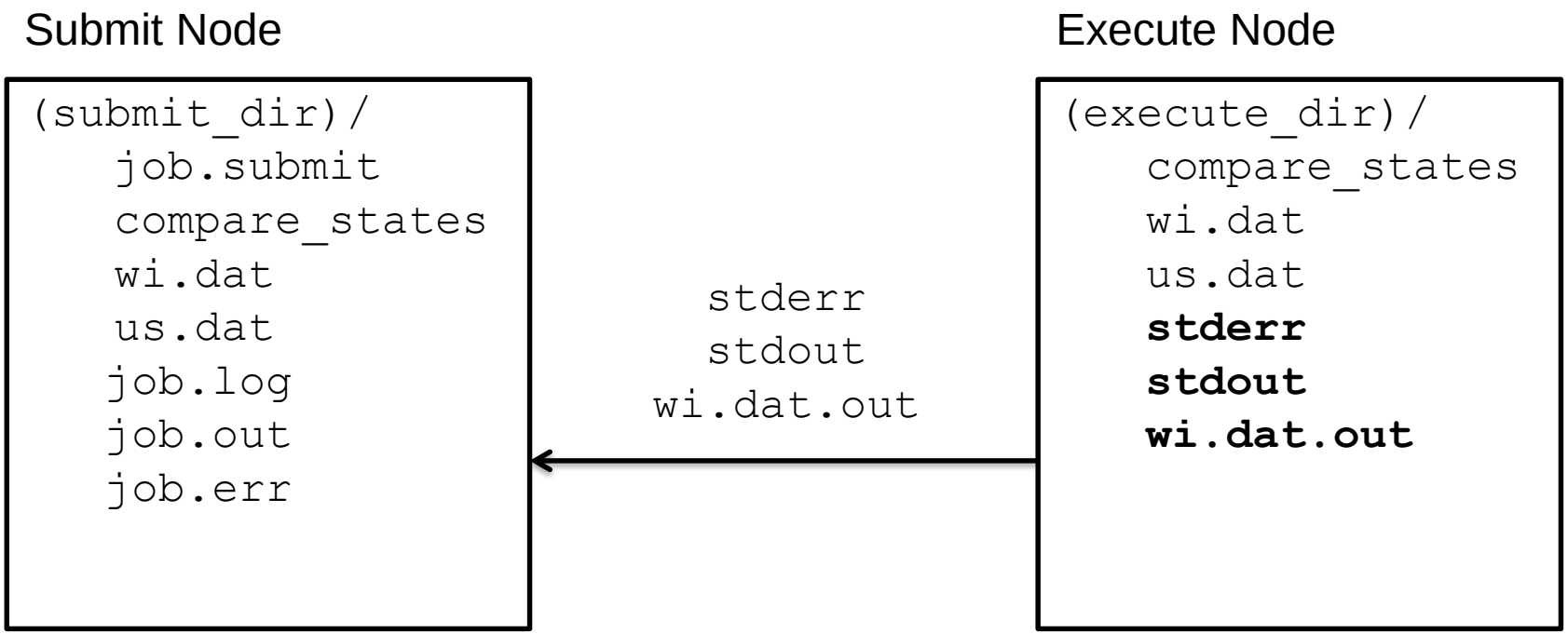
```
(submit_dir)/
    job.submit
    compare_states
    wi.dat
    us.dat
    job.log
    job.out
    job.err
```

Execute Node

```
(execute_dir)/
    compare_states
    wi.dat
    us.dat
    stderr
    stdout
    wi.dat.out
```

Job Completes

```
$ condor_q -nobatch
-- Schedd: submit-5.chtc.wisc.edu : <128.104.101.92:9618?...
 ID          OWNER      SUBMITTED     RUN_TIME ST PRI SIZE CMD
128          alice      5/9  11:09   0+00:02:02 >  0    0.0 compare_states wi.dat us.dat

1 jobs; 0 completed, 0 removed, 0 idle, 1 running, 0 held, 0 suspended
```



Job Completes (cont.)

```
$ condor_q -nobatch


-- Schedd: submit-5.chtc.wisc.edu : <128.104.101.92:9618?...
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD

0 jobs; 0 completed, 0 removed, 0 idle, 0 running, 0 held, 0 suspended
```

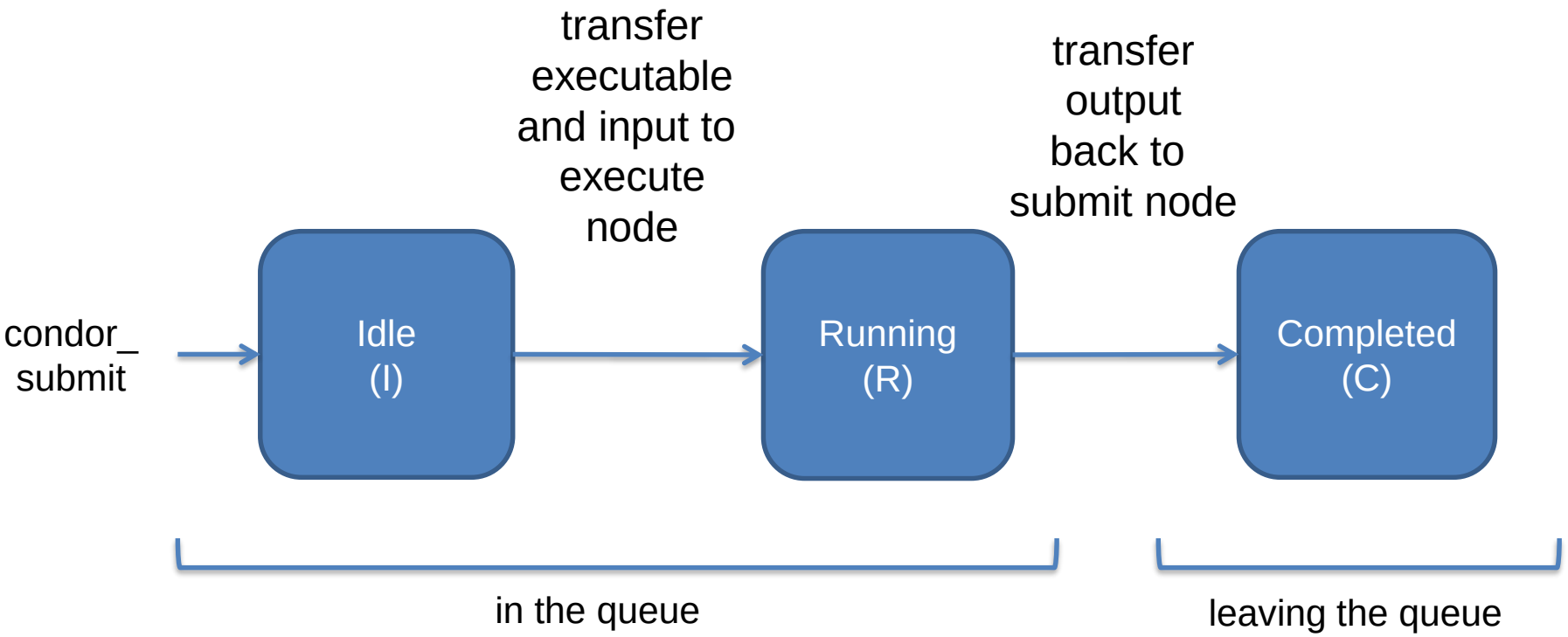
Submit Node

```
(submit_dir)/
    job.submit
    compare_states
    wi.dat
    us.dat
    job.log
    job.out
    job.err
    wi.dat.out
```

Log File

```
000 (128.000.000) 05/09 11:09:08 Job submitted from host:
<128.104.101.92&sock=6423_b881_3>
...
001 (128.000.000) 05/09 11:10:46 Job executing on host:
<128.104.101.128:9618&sock=5053_3126_3>
...
006 (128.000.000) 05/09 11:10:54 Image size of job updated: 220
    1  -  MemoryUsage of job (MB)
    220  -  ResidentSetSize of job (KB)
...
005 (128.000.000) 05/09 11:12:48 Job terminated.
    (1) Normal termination (return value 0)
        Usr 0 00:00:00, Sys 0 00:00:00  -  Run Remote Usage
        Usr 0 00:00:00, Sys 0 00:00:00  -  Run Local Usage
        Usr 0 00:00:00, Sys 0 00:00:00  -  Total Remote Usage
        Usr 0 00:00:00, Sys 0 00:00:00  -  Total Local Usage
    0  -  Run Bytes Sent By Job
    33  -  Run Bytes Received By Job
    0  -  Total Bytes Sent By Job
    33  -  Total Bytes Received By Job
    Partitionable Resources :    Usage  Request Allocated
       Cpus                 :                1         1
       Disk (KB)            :       14    20480  17203728
       Memory (MB)          :        1       20        20
```

Job States


transfer executable and input to execute node
transfer output back to submit node
condor_ submit
Idle (I)
Running (R)
Completed (C)
in the queue
leaving the queue

Assumptions

- Aspects of your submit file may be dictated by infrastructure and configuration
- For example: file transfer
  - previous example assumed files would need to be transferred between submit/execute

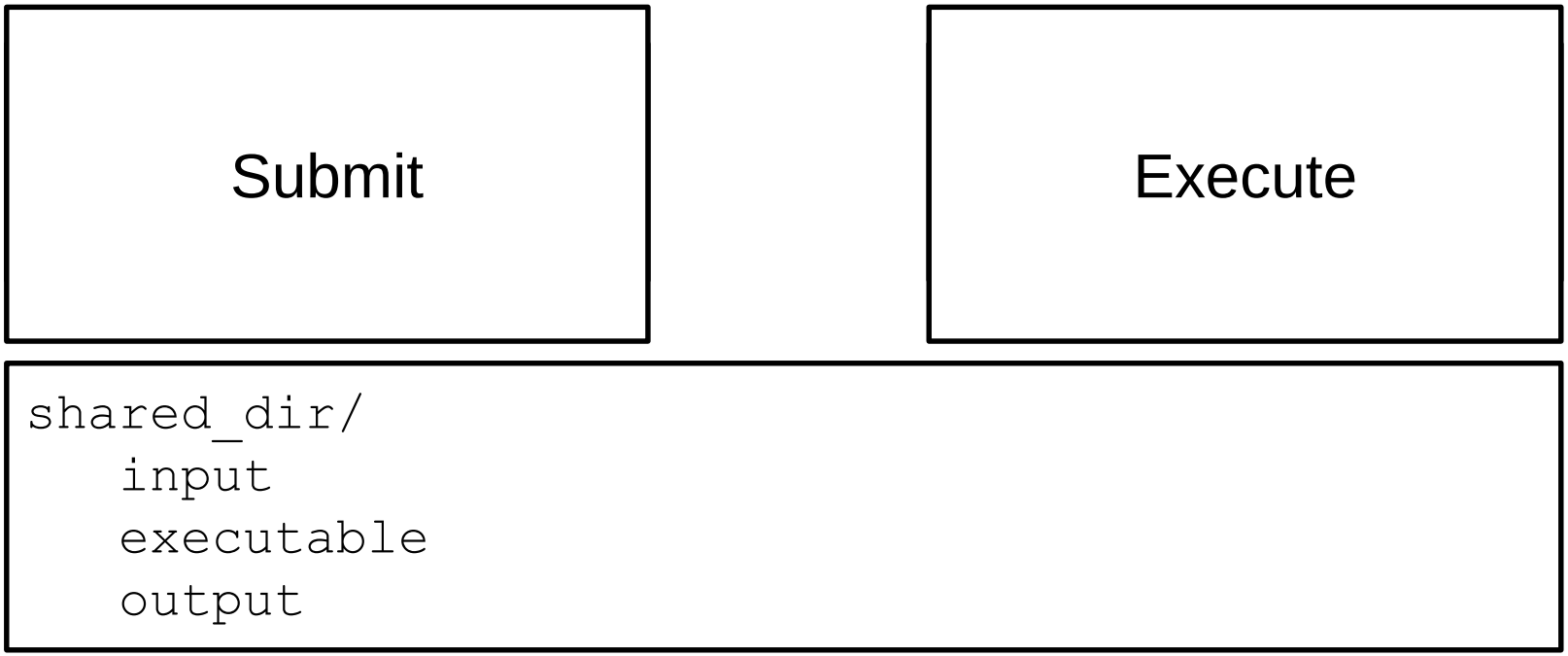
```
should_transfer_files = YES
```

  - not the case with a shared file system

```
should_transfer_files = NO
```

Shared file system

- If a system has a shared file system, where file transfer is not enabled, the submit directory and execute directory are the same.



Resource Request

- Jobs are nearly always using a part of a computer, not the whole thing
- Very important to request appropriate resources (memory, cpus, disk) for a job

Resource Assumptions

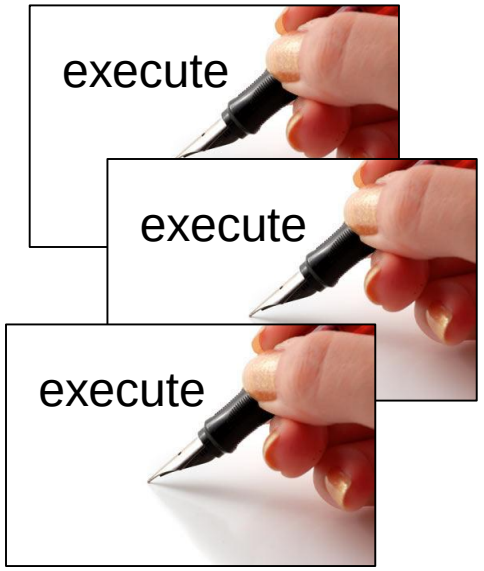
- Even if your system has default CPU, memory and disk requests, these may be too small!
- Important to run test jobs and use the log file to request the right amount of resources:
  - requesting too little: causes problems for your and other jobs; jobs might by held by HTCondor
  - requesting too much: jobs will match to fewer “slots”

Job Matching and Class Ad Attributes

The Central Manager

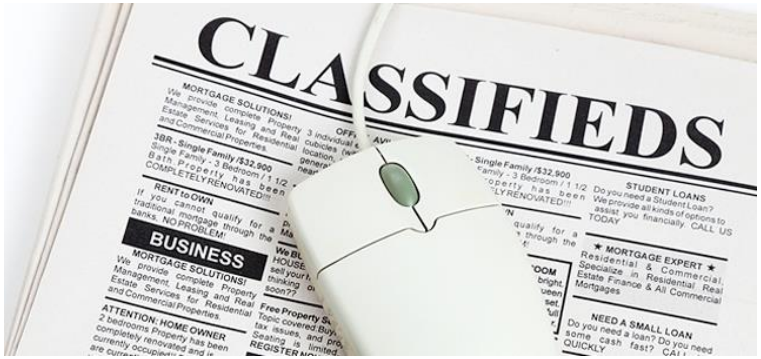
- HTCondor matches jobs with computers via a “central manager”.

central manager



Class Ads

- HTCondor stores a list of information about each job and each computer.
- This information is stored as a “Class Ad”



- Class Ads have the format:

```
AttributeName = value
```

can be a boolean, number, or string

HTCondor Manual: Appendix A: Class Ad Attributes

Job Class Ad

```
executable = compare_states
arguments = wi.dat us.dat wi.dat.out

should_transfer_files = YES
transfer_input_files = us.dat, wi.dat
when_to_transfer_output = ON_EXIT

log = job.log
output = job.out
error = job.err

request_cpus = 1
request_disk = 20MB
request_memory = 20MB

queue 1
```

+

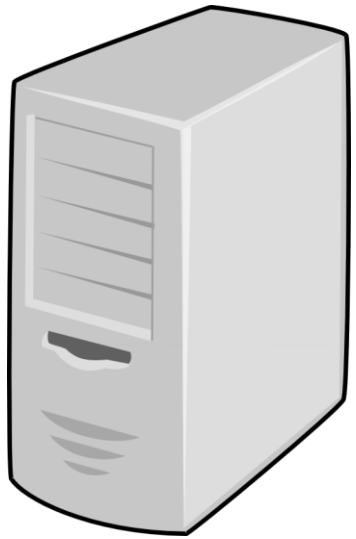
HTCondor configuration*

=

```
RequestCpus = 1
Err = "job.err"
WhenToTransferOutput = "ON_EXIT"
TargetType = "Machine"
Cmd =
"/home/alice/tests/htcondor_week/compar
e_states"
JobUniverse = 5
Iwd = "/home/alice/tests/htcondor_week"
RequestDisk = 20480
NumJobStarts = 0
WantRemoteIO = true
OnExitRemove = true
TransferInput = "us.dat,wi.dat"
MyType = "Job"
Out = "job.out"
UserLog =
"/home/alice/tests/htcondor_week/job.lo
g"
RequestMemory = 20
...
```

*Configuring HTCondor will be covered in "Administering HTCondor", by Greg Thain, at 1:15 today (May 2)

Computer “Machine” Class Ad



+

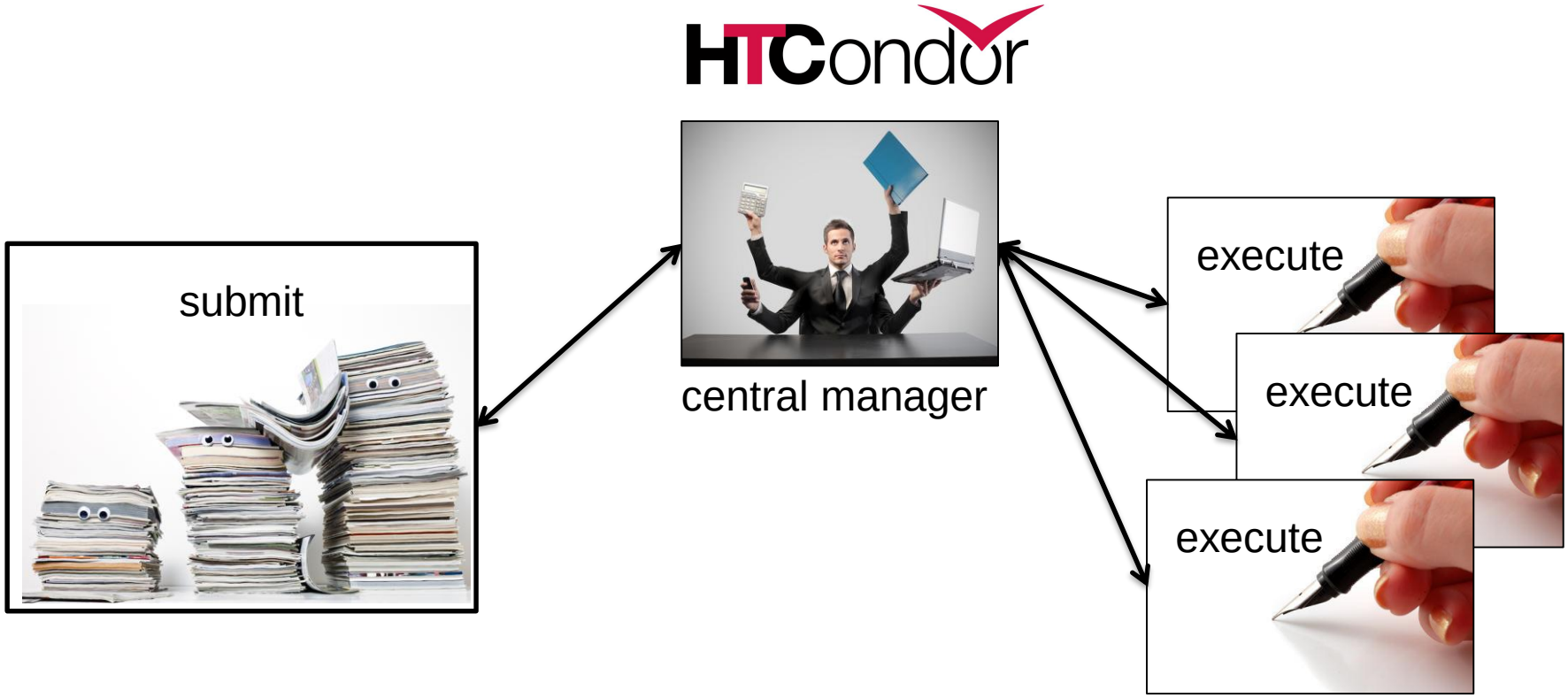
HTCondor configuration

=

```
HasFileTransfer = true
DynamicSlot = true
TotalSlotDisk = 4300218.0
TargetType = "Job"
TotalSlotMemory = 2048
Mips = 17902
Memory = 2048
UtsnameSysname = "Linux"
MAX_PREEMPT = ( 3600 * 72 )
Requirements = ( START ) && (
IsValidCheckpointPlatform ) && (
WithinResourceLimits )
OpSysMajorVer = 6
TotalMemory = 9889
HasGluster = true
OpSysName = "SL"
HasDocker = true

...
```

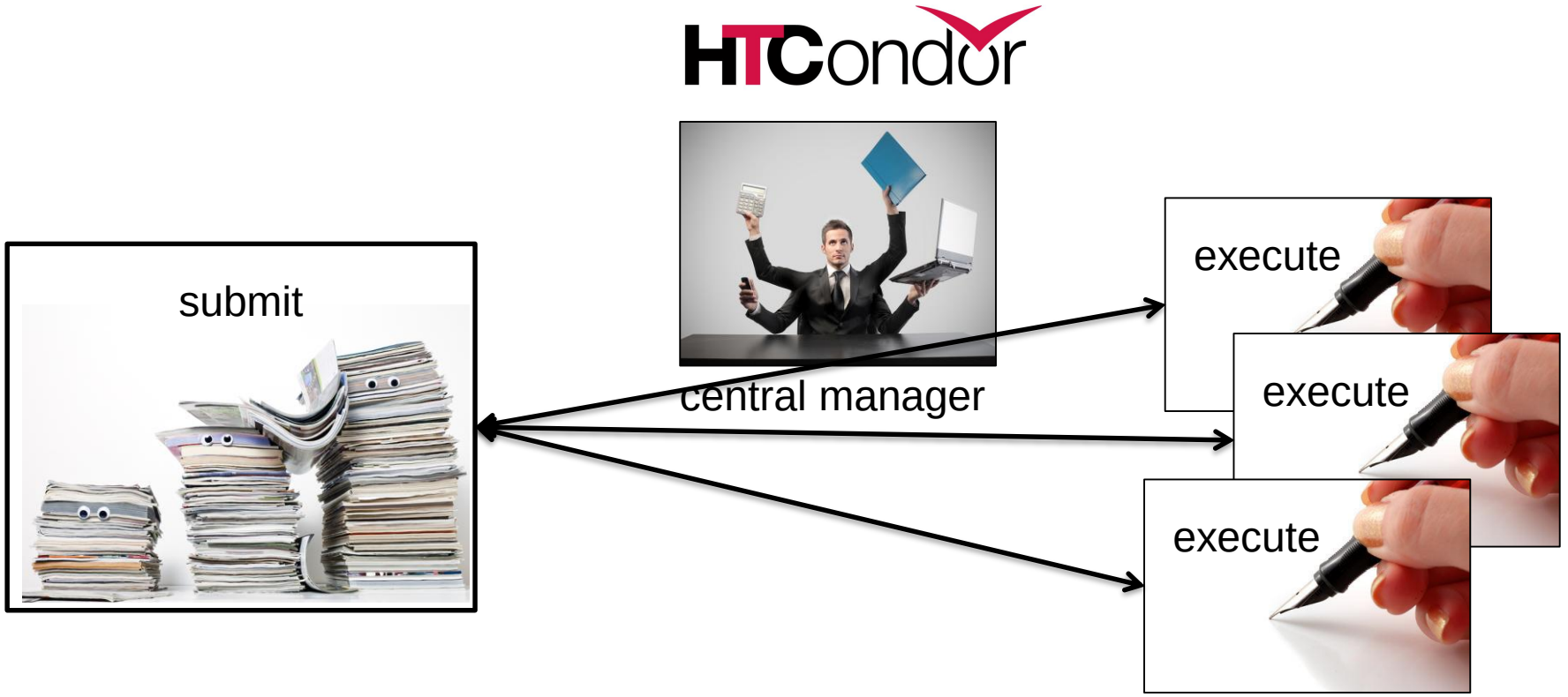
Job Matching

- On a regular basis, the central manager reviews Job and Machine Class Ads and matches jobs to computers.



Job Execution

- (Then the submit and execute points communicate directly.)



Class Ads for People

- Class Ads also provide lots of useful information about jobs and computers to HTCondor users and administrators

Finding Job Attributes

- Use the “long” option for `condor_q`

  `condor_q -l` ***`JobId`***

```
$ condor_q -l 128.0
WhenToTransferOutput = "ON_EXIT"
TargetType = "Machine"
Cmd = "/home/alice/tests/htcondor_week/compare_states"
JobUniverse = 5
Iwd = "/home/alice/tests/htcondor_week"
RequestDisk = 20480
NumJobStarts = 0
WantRemoteIO = true
OnExitRemove = true
TransferInput = "us.dat,wi.dat"
MyType = "Job”
UserLog = "/home/alice/tests/htcondor_week/job.log"
RequestMemory = 20
...
```

Some Useful Job Attributes

- `UserLog`: location of job log
- `Iwd`: Initial Working Directory (i.e. submission directory) on submit node
- `MemoryUsage`: maximum memory the job has used
- `RemoteHost`: where the job is running
- `BatchName`: attribute to label job batches
- ...and more

Selectively display specific attributes

- Use the “auto-format” option:

  condor_q [U/C/J] **-af** ***Attribute1 Attribute2 ...***

```
$ condor_q -af ClusterId ProcId RemoteHost MemoryUsage

17315225 116 slot1_1@e092.chtc.wisc.edu 1709
17315225 118 slot1_2@e093.chtc.wisc.edu 1709
17315225 137 slot1_8@e125.chtc.wisc.edu 1709
17315225 139 slot1_7@e121.chtc.wisc.edu 1709
18050961 0 slot1_5@c025.chtc.wisc.edu 196
18050963 0 slot1_3@atlas10.chtc.wisc.edu 269
18050964 0 slot1_25@e348.chtc.wisc.edu 245
18050965 0 slot1_23@e305.chtc.wisc.edu 196
18050971 0 slot1_6@e176.chtc.wisc.edu 220
```

Other Displays

- See the whole queue (all users, all jobs)

  `condor_q -all`

```
$ condor_q -all

-- Schedd: submit-5.chtc.wisc.edu : <128.104.101.92:9618?...
OWNER    BATCH_NAME    SUBMITTED   DONE   RUN    IDLE   HOLD  TOTAL JOB_IDS
alice    DAG: 128      5/9  02:52    982      2       _      _   1000 18888976.0 ...
bob      DAG: 139      5/9  09:21     _       1      89      _    180 18910071.0 ...
alice    DAG: 219      5/9  10:31     1     997       2      _   1000 18911030.0 ...
bob      DAG: 226      5/9  10:51    10       _       1      _     44 18913051.0
bob      CMD: ce.sh    5/9  10:55     _       _       _      2      _ 18913029.0 ...
alice    CMD: sb       5/9  10:57     _       2     998      _      _ 18913030.0-999
```

`condor_q` Reminder

- Default output is batched jobs
  - Batches can be grouped manually using the `JobBatchName` attribute in a submit file:

```
+JobBatchName = "CoolJobs"
```

  - Otherwise HTCondor groups jobs automatically
- To see individual jobs, use:

`condor_q -nobatch`

Class Ads for Computers

as **`condor_q`** is to jobs, **`condor_status`** is to computers (or “machines”)

```
$ condor_status
Name                                OpSys     Arch State          Activity  LoadAv    Mem Actvty
slot1@c001.chtc.wisc.edu              LINUX      X86_64 Unclaimed Idle      0.000     673 25+01
slot1_1@c001.chtc.wisc.edu            LINUX      X86_64 Claimed   Busy      1.000    2048  0+01
slot1_2@c001.chtc.wisc.edu            LINUX      X86_64 Claimed   Busy      1.000    2048  0+01
slot1_3@c001.chtc.wisc.edu            LINUX      X86_64 Claimed   Busy      1.000    2048  0+00
slot1_4@c001.chtc.wisc.edu            LINUX      X86_64 Claimed   Busy      1.000    2048  0+14
slot1_5@c001.chtc.wisc.edu            LINUX      X86_64 Claimed   Busy      1.000    1024  0+01
slot1@c002.chtc.wisc.edu              LINUX      X86_64 Unclaimed Idle      1.000    2693 19+19
slot1_1@c002.chtc.wisc.edu            LINUX      X86_64 Claimed   Busy      1.000    2048  0+04
slot1_2@c002.chtc.wisc.edu            LINUX      X86_64 Claimed   Busy      1.000    2048  0+01
slot1_3@c002.chtc.wisc.edu            LINUX      X86_64 Claimed   Busy      0.990    2048  0+02
slot1@c004.chtc.wisc.edu              LINUX      X86_64 Unclaimed Idle      0.010     645 25+05
slot1_1@c004.chtc.wisc.edu            LINUX      X86_64 Claimed   Busy      1.000    2048  0+01

                     Total Owner Claimed Unclaimed Matched Preempting Backfill  Drain

        X86_64/LINUX 10962     0   10340       613       0          0        0      9
      X86_64/WINDOWS     2     2       0         0       0          0        0      0

               Total 10964     2   10340       613       0          0        0      9
```


HTCondor Manual: condor_status

Machine Attributes

- Use same options as **condor_q**:

condor_status **-l** ***Slot/Machine***

condor_status **[Machine]** **-af** ***Attribute1 Attribute2 ...***

```
$ condor_status -l slot1_1@c001.chtc.wisc.edu
HasFileTransfer = true
COLLECTOR_HOST_STRING = "cm.chtc.wisc.edu”
TargetType = "Job”
TotalTimeClaimedBusy = 43334c001.chtc.wisc.edu
UtsnameNodename = ""
Mips = 17902
MAX_PREEMPT = ( 3600 * ( 72 - 68 * ( WantGlidein =?= true ) ) )
Requirements = ( START ) && ( IsValidCheckpointPlatform ) && (
WithinResourceLimits )
State = "Claimed"
OpSysMajorVer = 6
OpSysName = "SL”
...
```

Machine Attributes

- To summarize, use the “-compact” option

  `condor_status -compact`

```
$ condor_q -compact
Machine                          Platform      Slots Cpus Gpus  TotalGb FreCpu  FreeGb  CpuLoad ST
e007.chtc.wisc.edu               x64/SL6           8    8          23.46      0    0.00    1.24 Cb
e008.chtc.wisc.edu               x64/SL6           8    8          23.46      0    0.46    0.97 Cb
e009.chtc.wisc.edu               x64/SL6          11   16          23.46      5    0.00    0.81 **
e010.chtc.wisc.edu               x64/SL6           8    8          23.46      0    4.46    0.76 Cb
matlab-build-1.chtc.wisc.edu     x64/SL6           1   12          23.45     11   13.45    0.00 **
matlab-build-5.chtc.wisc.edu     x64/SL6           0   24          23.45     24   23.45    0.04 Ui
mem1.chtc.wisc.edu               x64/SL6          24   80        1009.67      8    0.17    0.60 **

                     Total Owner Claimed Unclaimed Matched Preempting Backfill  Drain

          x64/SL6 10416     0    9984       427       0          0        0      5
      x64/WinVista     2     2       0         0       0          0        0      0

             Total 10418     2    9984       427       0          0        0      5
```

(60 SECOND) PAUSE

Questions so far?

Submitting Multiple Jobs with HTCondor

Many Jobs, One Submit File

- HTCondor has built-in ways to submit multiple independent jobs with one submit file

Advantages

- Run many independent jobs...
  - analyze multiple data files
  - test parameter or input combinations
  - and more!
- ...without having to:
  - start each job individually
  - create separate submit files for each job

Multiple, Numbered, Input Files

job.submit

```
executable = analyze.exe
arguments = file.in file.out
transfer_input_files = file.in

log = job.log
output = job.out
error = job.err

queue
```

(submit_dir)/

```
analyze.exe
file0.in
file1.in
file2.in

job.submit
```

- Goal: create 3 jobs that each analyze a different input file.

Multiple Jobs, No Variation

job.submit

```
executable = analyze.exe
arguments = file0.in file0.out
transfer_input_files = file.in

log = job.log
output = job.out
error = job.err

queue 3
```

(submit_dir)/

```
analyze.exe
file0.in
file1.in
file2.in

job.submit
```

- This file generates 3 jobs, but doesn't use multiple inputs and will overwrite outputs

Automatic Variables

- Each job's `ClusterId` and `ProcId` numbers are saved as job attributes
- They can be accessed inside the submit file using:
  - `$(ClusterId)`
  - `$(ProcId)`

Job Variation

job.submit

```
executable = analyze.exe
arguments = file0.in file0.out
transfer_input_files = file0.in

log = job.log
output = job.out
error = job.err

queue
```

(submit_dir)/

```
analyze.exe
file0.in
file1.in
file2.in

job.submit
```

- How to uniquely identify each job (filenames, log/out/err names)?

Using $(ProcId)

job.submit

```
executable = analyze.exe
arguments = file$(ProcId).in file$(ProcId).out
should_transfer_files = YES
transfer_input_files = file$(ProcId).in
when_to_transfer_output = ON_EXIT

log = job_$(ClusterId).log
output = job_$(ClusterId)_$(ProcId).out
error = job_$(ClusterId)_$(ProcId).err

queue 3
```

- Use the `$(ClusterId)`, `$(ProcId)` variables to provide unique values to jobs.*

* May also see `$(Cluster)`, `$(Process)` in documentation

Organizing Jobs

```
12181445_0.err  16058473_0.err  17381628_0.err  18159900_0.err  5175744_0.err  7266263_0.err
12181445_0.log  16058473_0.log  17381628_0.log  18159900_0.log  5175744_0.log  7266263_0.log
12181445_0.out  16058473_0.out  17381628_0.out  18159900_0.out  5175744_0.out  7266263_0.out
13609567_0.err  16060330_0.err  17381640_0.err  3446080_0.err   5176204_0.err  7266267_0.err
13609567_0.log  16060330_0.log  17381640_0.log  3446080_0.log   5176204_0.log  7266267_0.log
13609567_0.out  16060330_0.out  17381640_0.out  3446080_0.out   5176204_0.out  7266267_0.out
13612268_0.err  16254074_0.err  17381665_0.err  3446306_0.err   5295132_0.err  7937420_0.err
13612268_0.log  16254074_0.log  17381665_0.log  3446306_0.log   5295132_0.log  7937420_0.log
13612268_0.out  16254074_0.out  17381665_0.out  3446306_0.out   5295132_0.out  7937420_0.out
13630381_0.err  17134215_0.err  17381676_0.err  4347054_0.err   5318339_0.err  8779997_0.err
13630381_0.log  17134215_0.log  17381676_0.log  4347054_0.log   5318339_0.log  8779997_0.log
13630381_0.out  17134215_0.out  17381676_0.out  4347054_0.out   5318339_0.out  8779997_0.out
```

Shared Files

- HTCondor can transfer an entire directory or all the contents of a directory
  - transfer whole directory

```
transfer_input_files = shared
```

  - transfer contents only

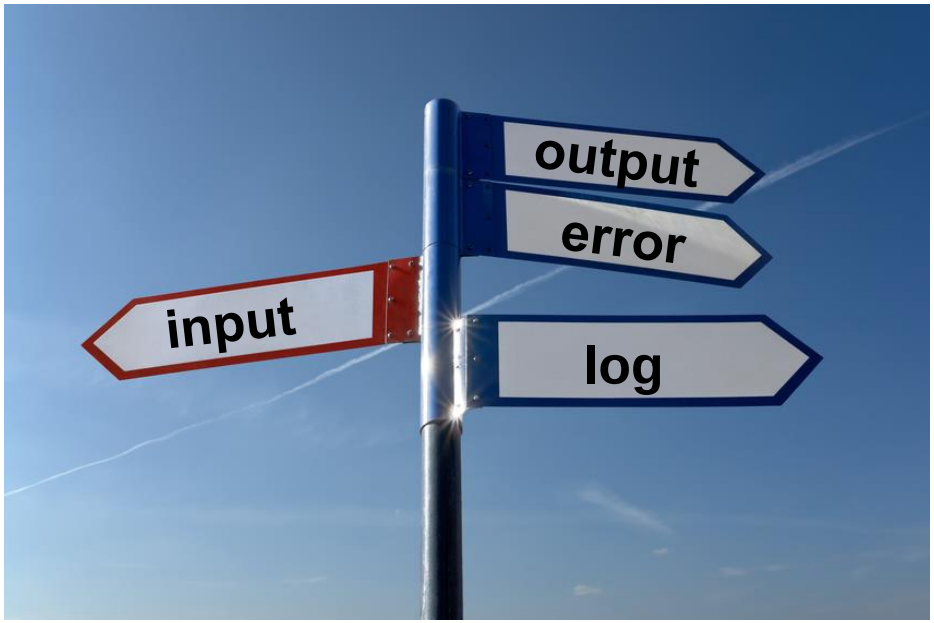
```
transfer_input_files = shared/
```

```
(submit_dir)/
job.submit
shared/
    reference.db
    parse.py
    analyze.py
    cleanup.py
    links.config
```

- Useful for jobs with many shared files; transfer a directory of files instead of listing files individually

Organize Files in Sub-Directories

- Create sub-directories* and use paths in the submit file to separate input, error, log, and output files.



* must be created before the job is submitted

Use Paths for File Type

(submit_dir)/

job.submit	file0.out	**input**/	**log**/	**err**/
analyze.exe	file1.out	file0.in	job0.log	job0.err
	file2.out	file1.in	job1.log	job1.err
		file2.in	job2.log	job2.err

job.submit

```
executable = analyze.exe
arguments = file$(Process).in file$(ProcId).out
transfer_input_files = input/file$(ProcId).in

log = log/job$(ProcId).log
error = err/job$(ProcId).err

queue 3
```

InitialDir

- Change the submission directory for each job using `initialdir`
- Allows the user to organize job files into separate directories.
- Use the same name for all input/output files
- Useful for jobs with lots of output files

Separate Jobs with InitialDir

```
(submit_dir)/
job.submit          job0/          job1/          job2/
analyze.exe           file.in        file.in        file.in
                      job.log        job.log        job.log
                      job.err        job.err        job.err
                      file.out       file.out       file.out
```

job.submit

```
executable = analyze.exe
initialdir = job$(ProcId)
arguments = file.in file.out
transfer_input_files = file.in

log = job.log
error = job.err

queue 3
```

Executable should be in the directory with the submit file, *not* in the individual job directories

Other Submission Methods

- What if your input files/directories aren't numbered from 0 - (N-1)?
- There are other ways to submit many jobs!

Submitting Multiple Jobs

```
executable = compare_states
arguments = wi.dat us.dat wi.dat.out

transfer_input_files = us.dat, wi.dat

queue 1
```

Replacing single job inputs

```
executable = compare_states
arguments = $(infile) us.dat $(infile).out

transfer_input_files = us.dat, $(infile)

queue ...
```

with a variable of choice

Possible Queue Statements

multiple "queue" statements	`infile = wi.dat`
	`queue 1`
	`infile = ca.dat`
	`queue 1`
	`infile = ia.dat`
	`queue 1`

matching ... pattern	`queue infile matching *.dat`
in ... list	`queue infile in (wi.dat ca.dat ia.dat)`
from ... file	`queue infile from state_list.txt`

`wi.dat`
`ca.dat`
`ia.dat`

`state_list.txt`

Possible Queue Statements

multiple "queue" statements	~~infile = wi.dat queue 1 infile = ca.dat queue 1 infile = ia.dat queue 1~~ Not Recommended
matching ... pattern	`queue infile matching *.dat`
in ... list	`queue infile in (wi.dat ca.dat ia.dat)`

from ... file	`queue infile from state_list.txt`
	`wi.dat`
	`ca.dat`
	`ia.dat`
	`state_list.txt`

Queue Statement Comparison

multiple queue statements	Not recommended. Can be useful when submitting job batches where a single (non-file/argument) characteristic is changing
matching .. pattern	Natural nested looping, minimal programming, use optional “files” and “dirs” keywords to only match files or directories Requires good naming conventions,
in .. list	Supports multiple variables, all information contained in a single file, reproducible Harder to automate submit file creation
from .. file	Supports multiple variables, highly modular (easy to use one submit file for many job batches), reproducible Additional file needed

Using Multiple Variables

- Both the "`from`" and "`in`" syntax support using multiple variables from a list.

job.submit

```
executable = compare_states
arguments = -year $(option) -input
$(file)

should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = $(file)

queue file,option from job_list.txt
```

job_list.txt

```
wi.dat, 2010
wi.dat, 2015
ca.dat, 2010
ca.dat, 2015
ia.dat, 2010
ia.dat, 2015
```

HTCondor Manual: submit file options

Other Features

- Match only files or directories:

```
queue input matching files *.dat
```

```
queue directory matching dirs job*
```

- Submit multiple jobs with same input data

```
queue 10 input matching files *.dat
```

  – Use other automatic variables: `$(Step)`

```
arguments = -i $(input) -rep $(Step)
queue 10 input matching files *.dat
```

Testing and Troubleshooting

What Can Go Wrong?

- Jobs can go wrong “internally”:
  - something happens after the executable begins to run
- Jobs can go wrong from HTCondor’s perspective:
  - A job can’t be started at all,
  - Uses too much memory,
  - Has a badly formatted executable,
  - And more...

Reviewing Failed Jobs

- A job's log, output and error files can provide valuable information for troubleshooting

Log	Output	Error
• When jobs were submitted, started, and stopped • Resources used • Exit status • Where job ran • Interruption reasons	Any "print" or "display" information from your program	Captured by the operating system

Reviewing Jobs

- To review a large group of jobs at once, use **`condor_history`**

  As **`condor_q`** is to the present, **`condor_history`** is to the past

```
$ condor_history alice
 ID      OWNER    SUBMITTED   RUN_TIME    ST  COMPLETED   CMD
189.1012 alice    5/11 09:52   0+00:07:37 C   5/11 16:00 /home/alice
189.1002 alice    5/11 09:52   0+00:08:03 C   5/11 16:00 /home/alice
189.1081 alice    5/11 09:52   0+00:03:16 C   5/11 16:00 /home/alice
189.944  alice    5/11 09:52   0+00:11:15 C   5/11 16:00 /home/alice
189.659  alice    5/11 09:52   0+00:26:56 C   5/11 16:00 /home/alice
189.653  alice    5/11 09:52   0+00:27:07 C   5/11 16:00 /home/alice
189.1040 alice    5/11 09:52   0+00:05:15 C   5/11 15:59 /home/alice
189.1003 alice    5/11 09:52   0+00:07:38 C   5/11 15:59 /home/alice
189.962  alice    5/11 09:52   0+00:09:36 C   5/11 15:59 /home/alice
189.961  alice    5/11 09:52   0+00:09:43 C   5/11 15:59 /home/alice
189.898  alice    5/11 09:52   0+00:13:47 C   5/11 15:59 /home/alice
```

HTCondor Manual: condor_history

“Live” Troubleshooting

- To log in to a job where it is running, use:

  condor_ssh_to_job *JobId*

```
$ condor_ssh_to_job 128.0
Welcome to slot1_31@e395.chtc.wisc.edu!
Your condor job is running with pid(s) 3954839.
```


HTCondor Manual: condor_ssh_to_job

Held Jobs

- HTCondor will put your job on hold if there's something YOU need to fix.
- A job that goes on hold is interrupted (all progress is lost) and kept from running again, but remains in the queue in the "H" state.

Diagnosing Holds

- If HTCondor puts a job on hold, it provides a hold reason, which can be viewed with:

  condor_q -hold [-wide]

```
$ condor_q -hold -af HoldReason
Error from slot1_1@wid-003.chtc.wisc.edu: Job has gone over
  memory limit of 2048 megabytes.
Error from slot1_20@e098.chtc.wisc.edu: SHADOW at
  128.104.101.92 failed to send file(s) to <128.104.101.98:35110>: error
  reading from /home/alice/script.py: (errno 2) No such file or directory;
  STARTER failed to receive file(s) from <128.104.101.92:9618>
Error from slot1_11@e138.chtc.wisc.edu: STARTER
  at 128.104.101.138 failed to send file(s) to <128.104.101.92:9618>; SHADOW at
  128.104.101.92 failed to write to file /home/alice/Test_18925319_16.err:
  (errno 122) Disk quota exceeded
Error from slot1_38@e270.chtc.wisc.edu: Failed
  to execute '/var/lib/condor/execute/slot1/dir_2471876/condor_exec.exe' with
  arguments 2: (errno=2: 'No such file or directory')
```

Common Hold Reasons

- Job has used more memory than requested
- Incorrect path to files that need to be transferred
- Badly formatted bash scripts (have Windows instead of Unix line endings)
- Submit directory is over quota
- The admin has put your job on hold

Fixing Holds

- Job attributes can be edited while jobs are in the queue using:

  `condor_qedit [U/C/J] Attribute Value`

```
$ condor_qedit 128.0 RequestMemory 3072
Set attribute ”RequestMemory".
```

- If a job has been fixed and can run again, release it with:

  `condor_release [U/C/J]`

```
$ condor_release 128.0
Job 18933774.0 released
```

HTCondor Manual: condor_qedit
HTCondor Manual: condor_release

Holding or Removing Jobs

- If you know your job has a problem and it hasn't yet completed, you can:
  - Place it on hold yourself, with **`condor_hold [U/C/J]`**

```
$ condor_hold bob
All jobs of user "bob" have been held
```

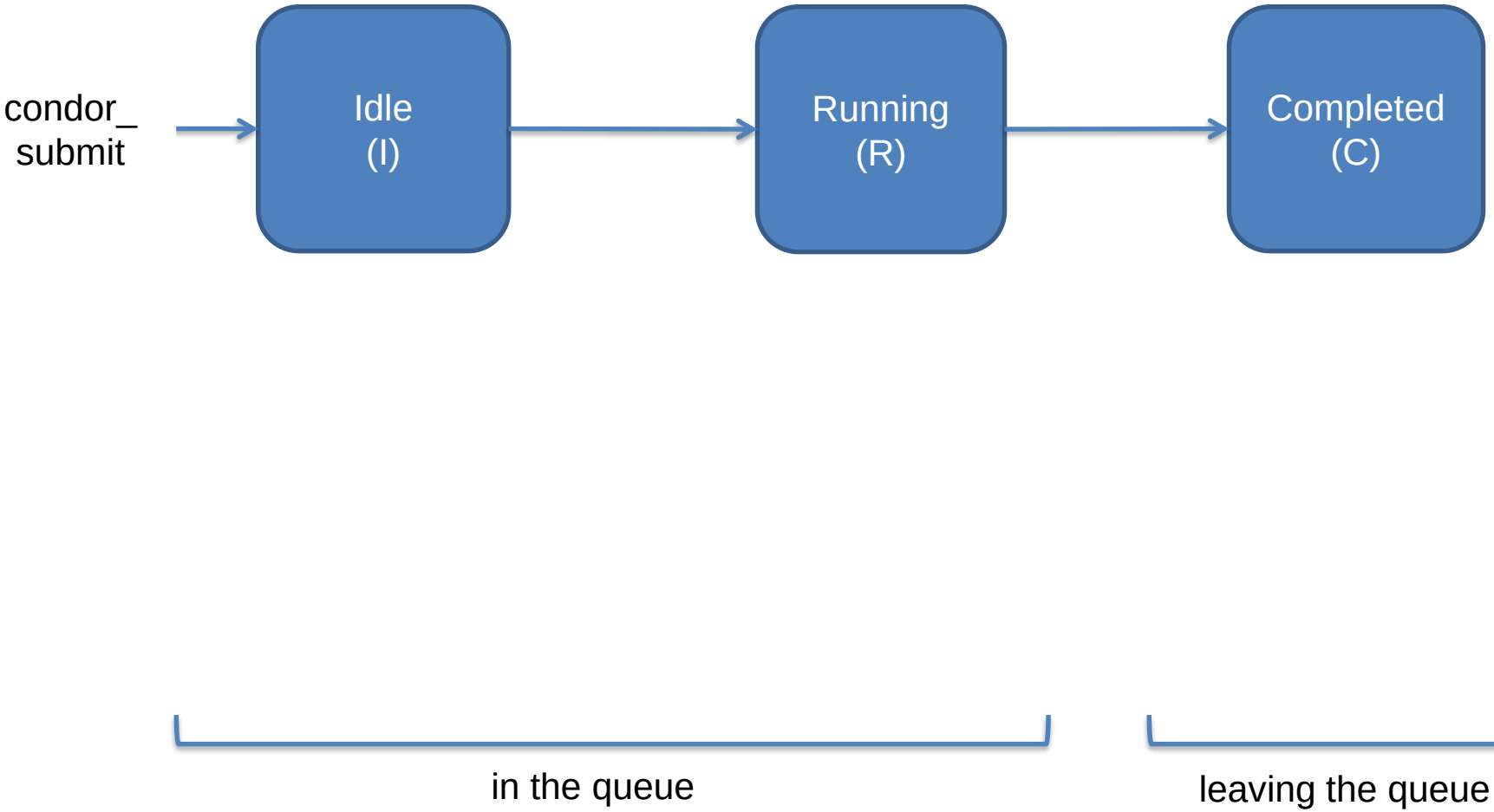
```
$ condor_hold 128
All jobs in cluster 128 have been held
```

```
$ condor_hold 128.0
Job 128.0 held
```

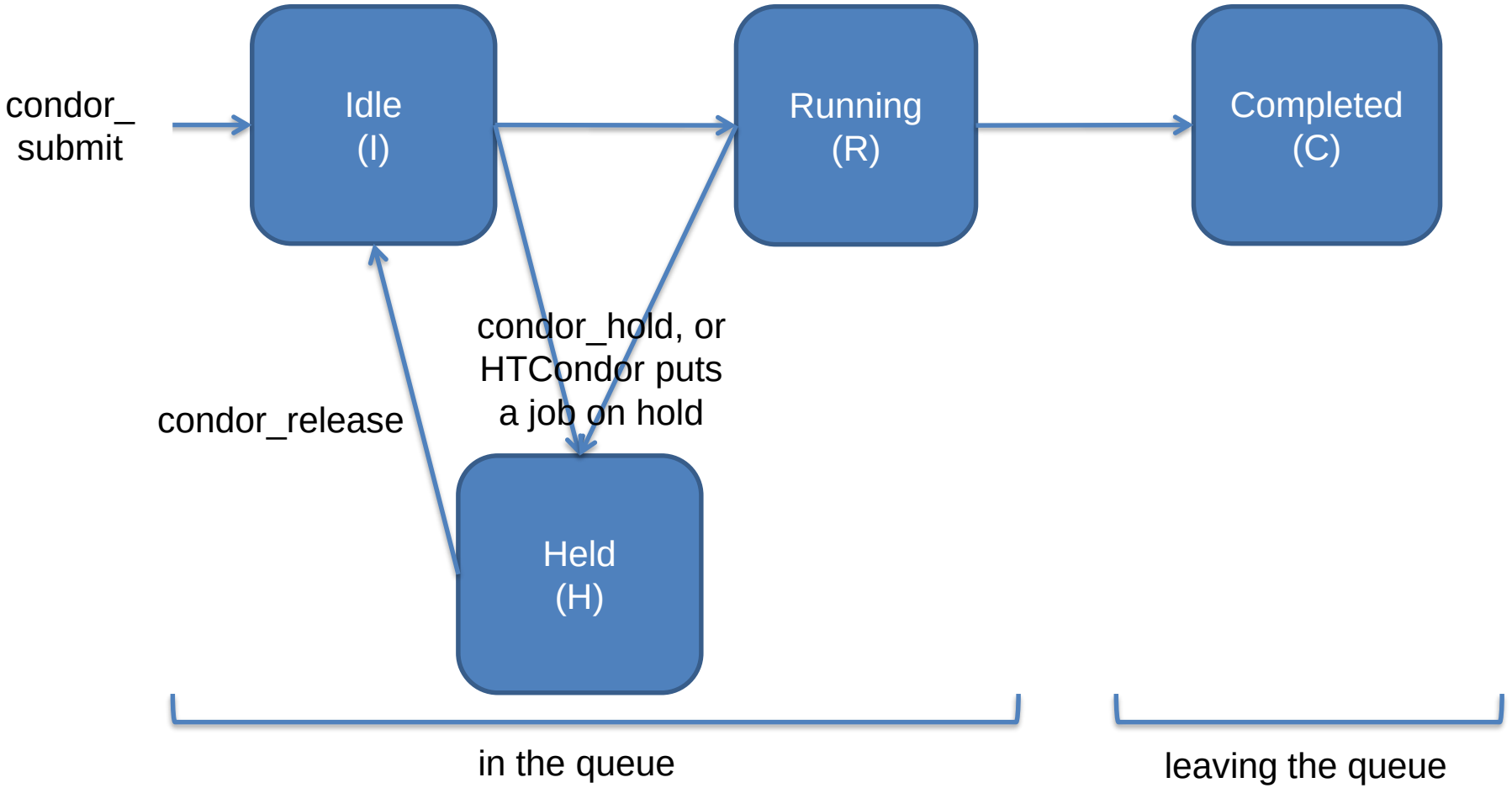
  - Remove it from the queue, using **`condor_rm [U/C/J]`**

HTCondor Manual: condor_hold
HTCondor Manual: condor_rm

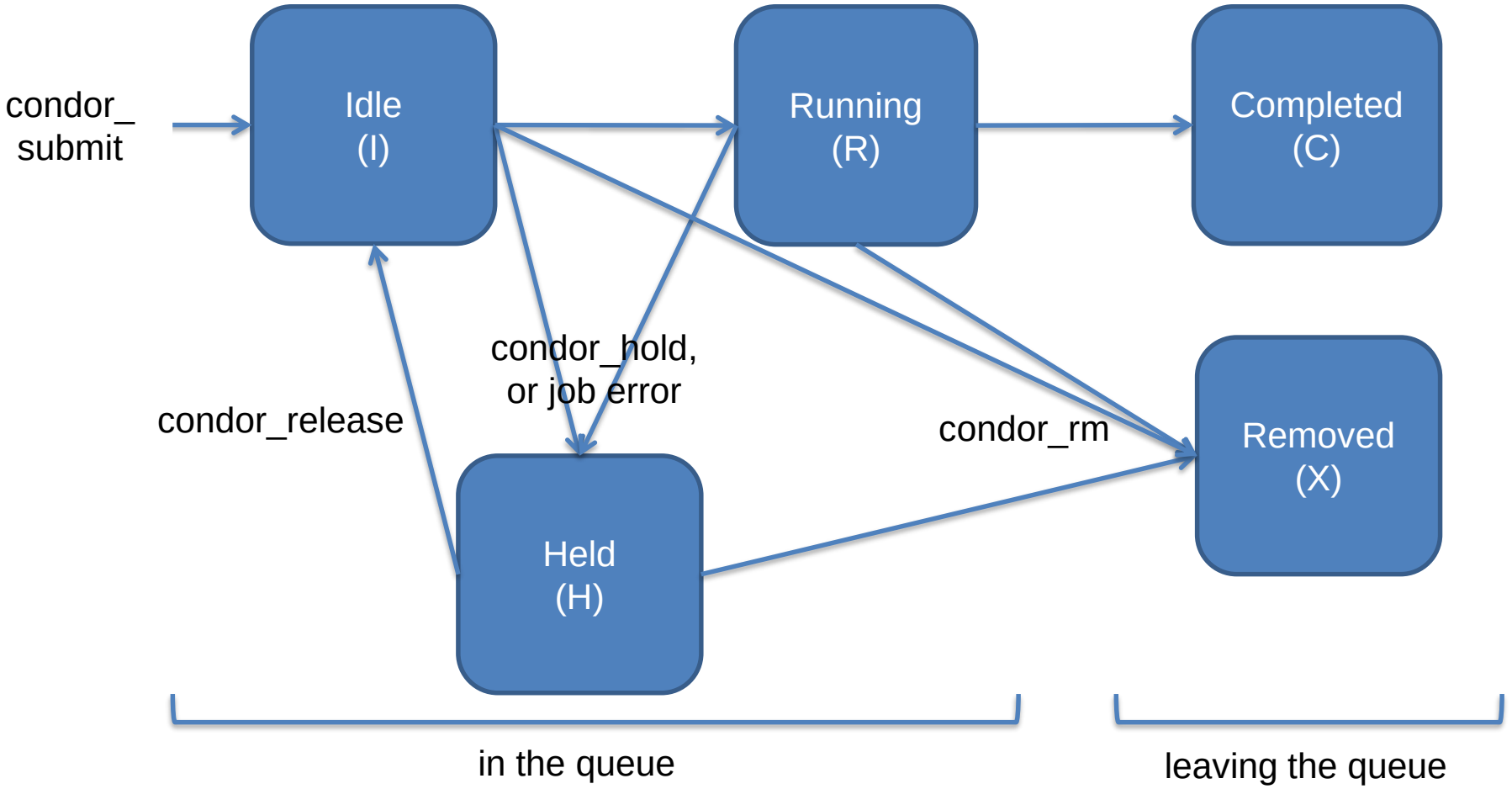
Job States, Revisited



Job States, Revisited



Job States, Revisited*



*not comprehensive

Use Cases and HTCondor Features

Interactive Jobs

- An interactive job proceeds like a normal batch job, but opens a bash session into the job's execution directory instead of running an executable.

condor_submit ***-i submit_file***

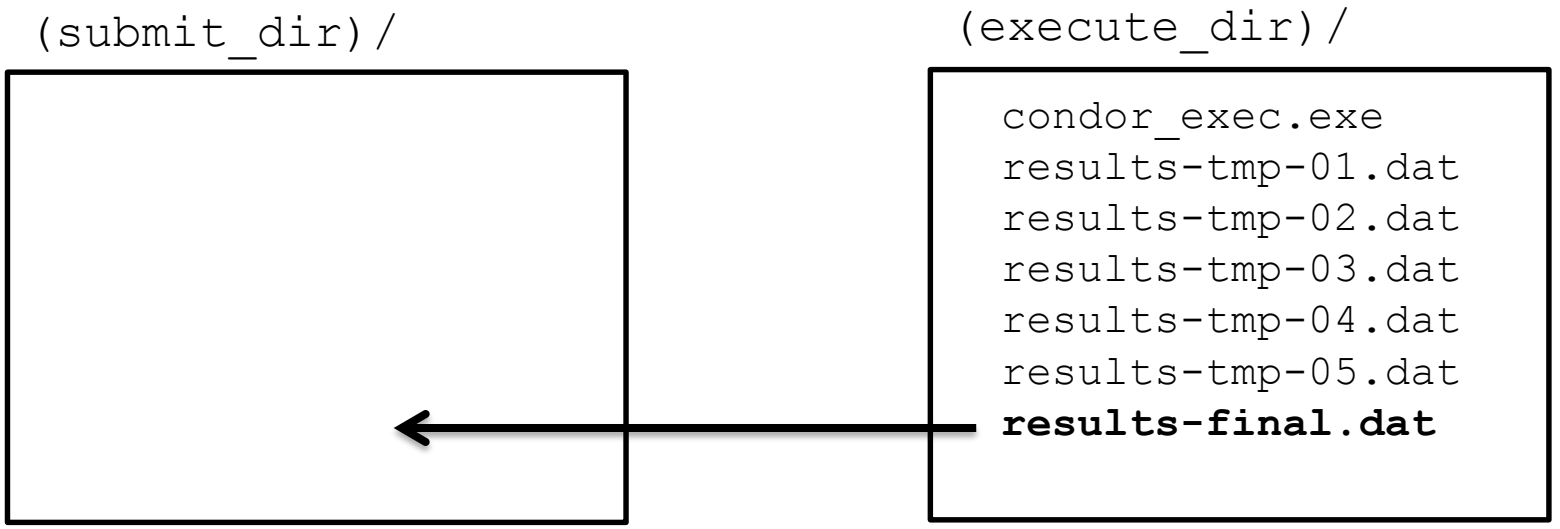
```
$ condor_submit -i interactive.submit
Submitting job(s).
1 job(s) submitted to cluster 18980881.
Waiting for job to start...
Welcome to slot1_9@e184.chtc.wisc.edu!
```

- Useful for testing and troubleshooting

Output Handling

- Only transfer back specific files from the job's execution using `transfer_ouput_files`

```
transfer_output_files = results-final.dat
```



condor_chirp

- What if you want to only read part of a file?
- What if you want to write records into an output file?

Use condor_chirp !

http://htcondor.org/manual/current/condor_chirp.html

(can also edit job classad or add entries to the job event log file)

Self-Checkpointing

- By default, a job that is interrupted will start from the beginning if it is restarted.
- It is possible to implement self-checkpointing, which will allow a job to restart from a saved state if interrupted.
- Self-checkpointing is useful for very long jobs, and being able to run on opportunistic resources.

Self-Checkpointing How-To

- Edit executable:
  - Atomically save intermediate states to a checkpoint file
  - Always check for a checkpoint file when starting
- Add HTCondor option that a) saves all intermediate/output files from the interrupted job and b) transfers them to the job when HTCondor runs it again

```
when_to_transfer_output = ON_EXIT_OR_EVICT
```

Job Universes

- HTCondor has different “universes” for running specialized job types

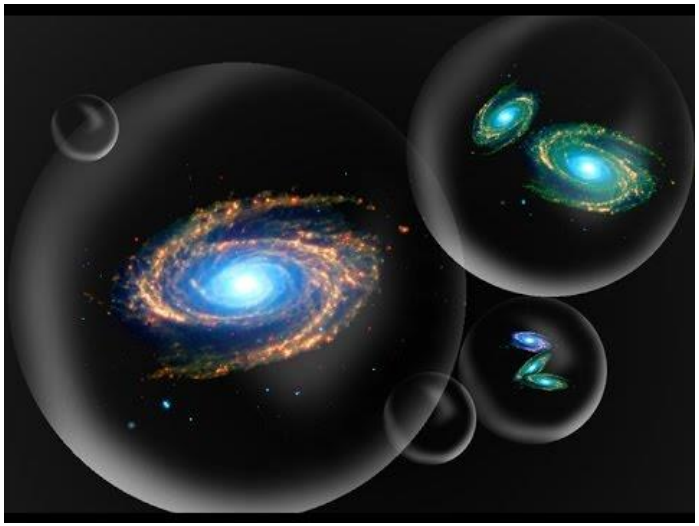
  HTCondor Manual: Choosing an HTCondor Universe

- Vanilla (default)
  - good for most software

  HTCondor Manual: Vanilla Universe

- Set in the submit file using:

```
universe = vanilla
```



Other Universes

- Standard
  - Built for code (C, fortran) that can be statically compiled with `condor_compile`

  HTCondor Manual: Standard Universe
- Java
  - Built-in Java support

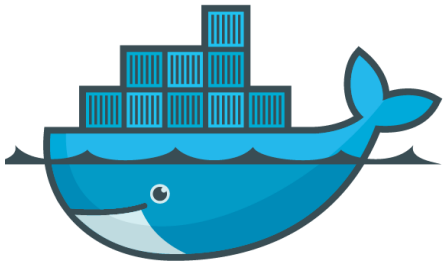
  HTCondor Manual: Java Applications
- Local
  - Run jobs on the submit node

  HTCondor Manual: Local Universe

Other Universes (cont.)

- Docker
  - Run jobs inside a Docker container

  HTCondor Manual: Docker Universe Applications



- VM
  - Run jobs inside a virtual machine

  HTCondor Manual: Virtual Machine Applications

- Parallel
  - Used for coordinating jobs across multiple servers (e.g. MPI code)
  - Not necessary for single server multi-core jobs

  HTCondor Manual: Parallel Applications

Multi-CPU and GPU Computing

- Jobs that use multiple cores on a single computer can be run in the vanilla universe (parallel universe not needed):

```
request_cpus = 16
```

- If there are computers with GPUs, request them with:

```
request_gpus = 1
```

Docker Universe

```
universe = docker
executable = /bin/my_executable
```

Executable comes either from submit machine or image

NOT FROM execute machine

Docker Universe

```
universe = docker
executable = /bin/my_executable
docker_image =deb7_and_HEP_stack
```

Image is the name of the docker image stored on execute machine

Docker Universe

```
universe = docker
executable = /bin/my_executable
docker_image =deb7_and_HEP_stack
transfer_input_files = some_input
```

HTCondor can transfer input files from submit machine into container

(same with output in reverse)

Docker Universe

```
universe = docker
executable = /bin/my_executable
arguments = arg1
docker_image = deb7_and_HEP_stack
transfer_input_files = some_input
output = out
error = err
log = log
queue
```

Automation

Automation

- After job submission, HTCondor manages jobs based on its configuration
- You can use options that will customize job management even further
- These options can automate when jobs are started, stopped, and removed.

Retries

- Problem: a small number of jobs fail with a known error code; if they run again, they complete successfully.
- Solution: If the job exits with the error code, leave it in the queue to run again

```
max_retries = 3
```

Retries, cont.

- Can also combine with

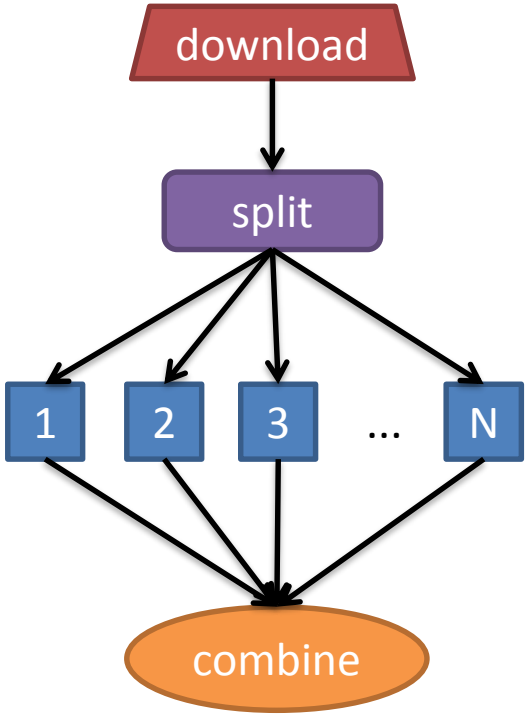
  success_exit_code = < *Integer* >

  retry_until = < *Integer* | *Expression* >

```
executable = foo.exe
max_retries = 5
retry_untl = ExitCode >= 0
queue
```

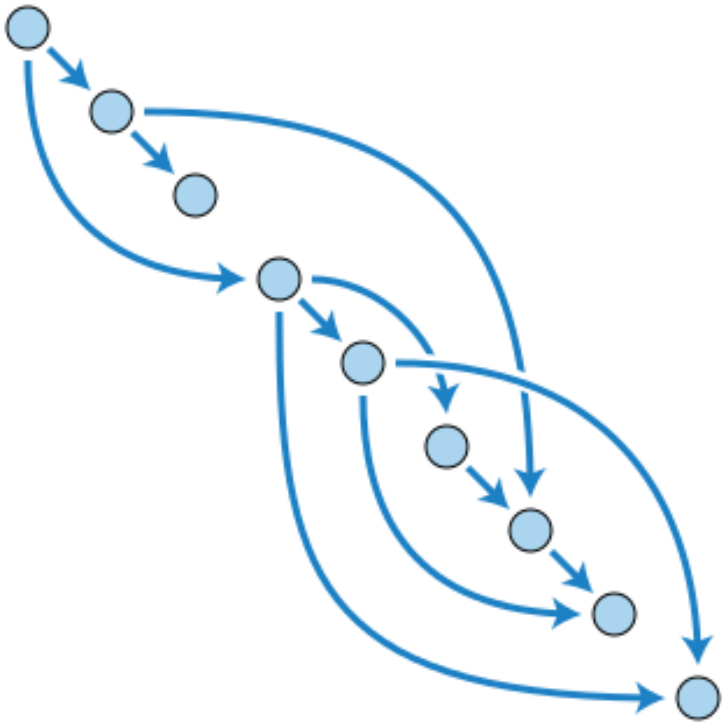
Workflows

- Problem: Want to submit jobs in a particular order, with dependencies between groups of jobs
- Solution: Write a DAG



DAG = ”directed acyclic graph”

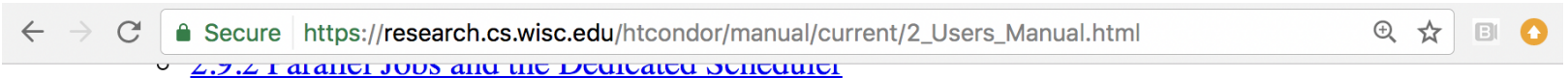
- topological ordering of vertices (“**nodes**”) is established by directional connections (“**edges**”)
- “acyclic” aspect requires a start and end, with no looped repetition
  - can contain cyclic subcomponents, covered in later slides for workflows



Wikimedia Commons

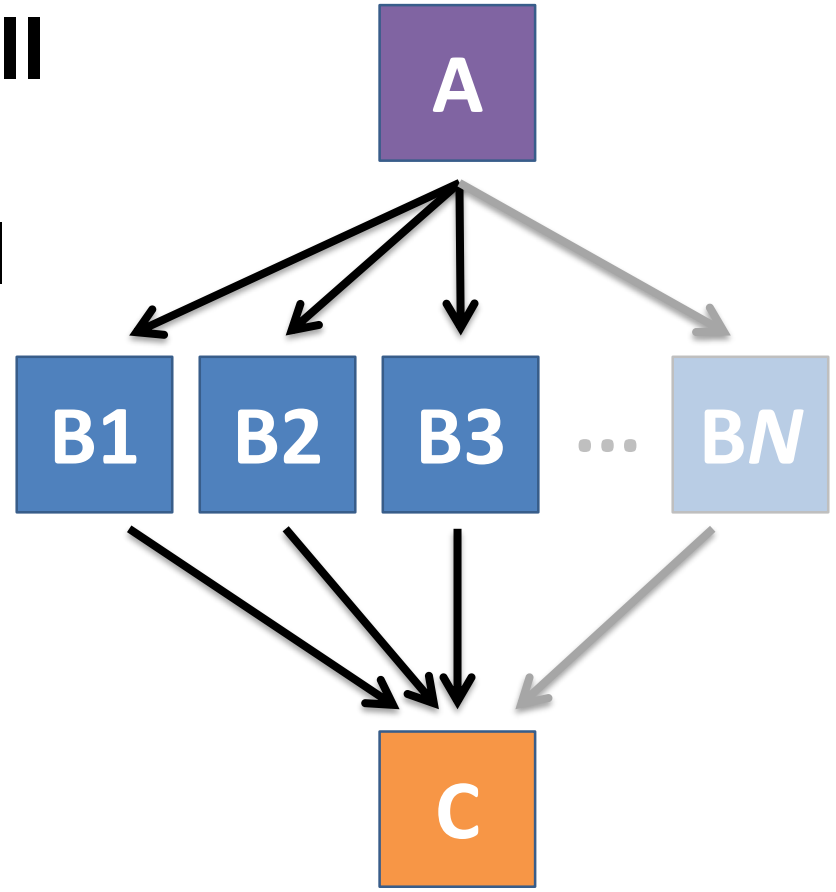
Describing Workflows with DAGMan

DAGMan in the HTCondor Manual



Simple Example for this Tutorial

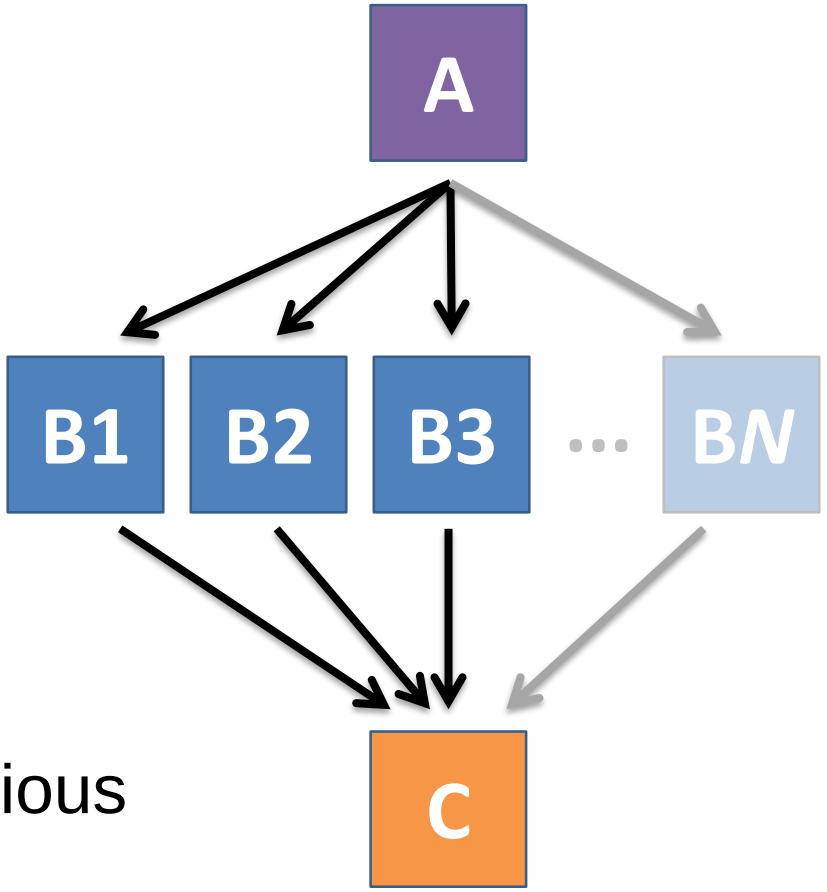
- **The DAG input file will** communicate the "nodes" and directional "edges" of the DAG



Basic DAG input file: JOB nodes, PARENT-CHILD edges

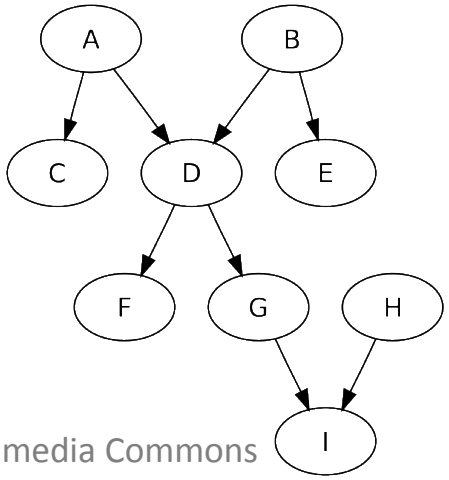
my.dag

```
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
```

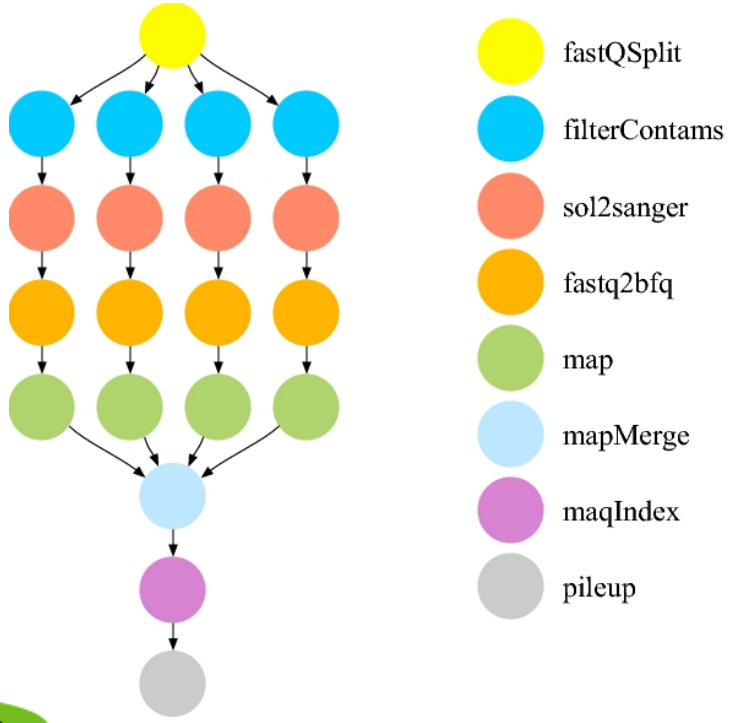


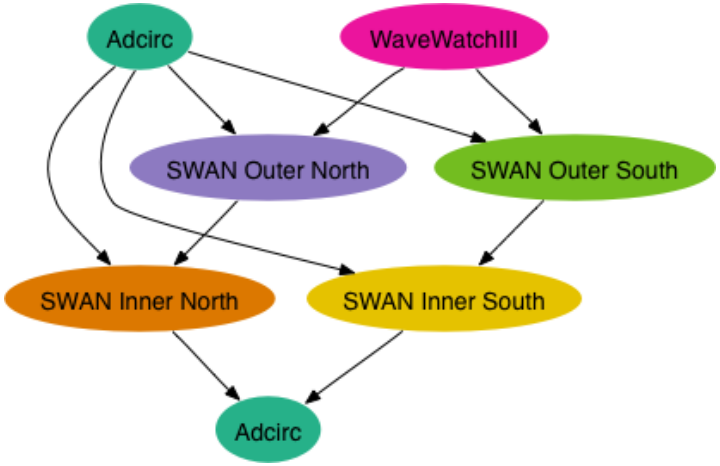
- Node names are used by various DAG features to modify their execution by DAG Manager.

Endless Workflow Possibilities



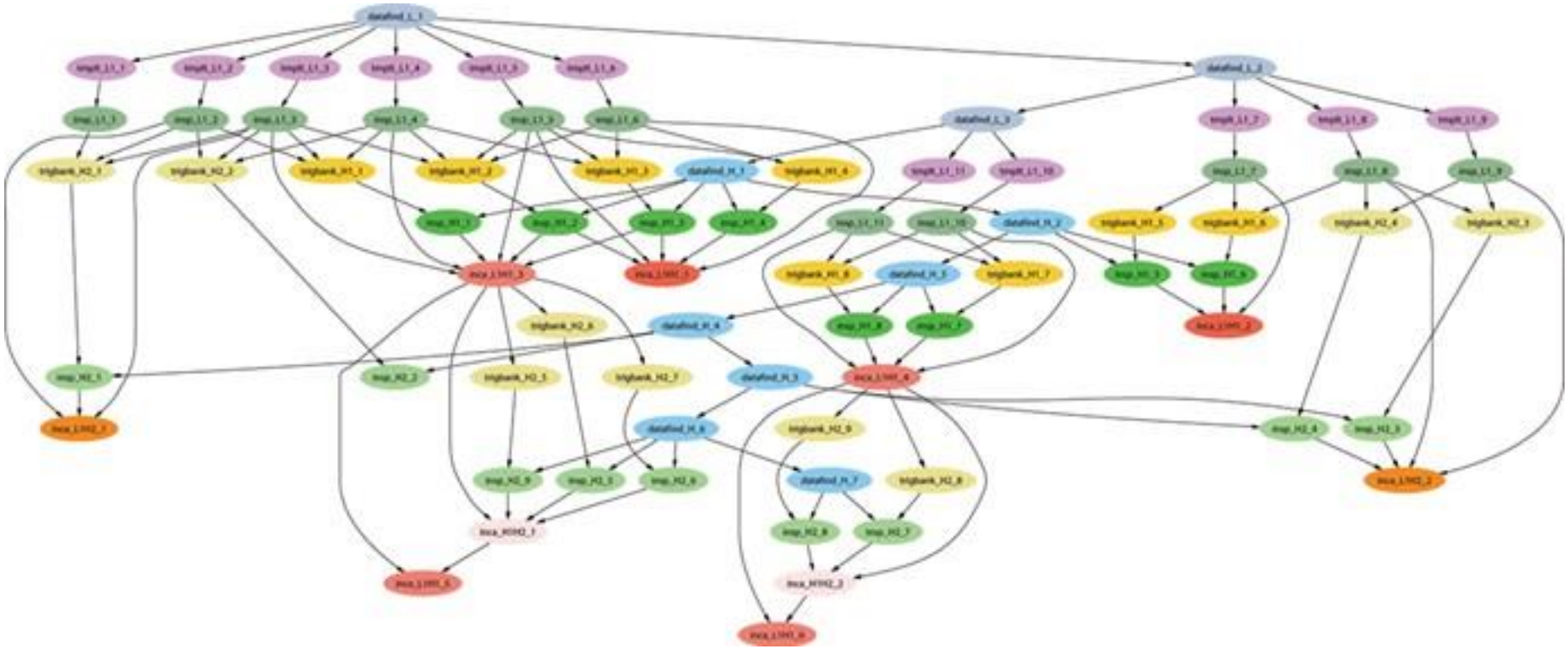
Wikimedia Commons





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

Endless Workflow Possibilities

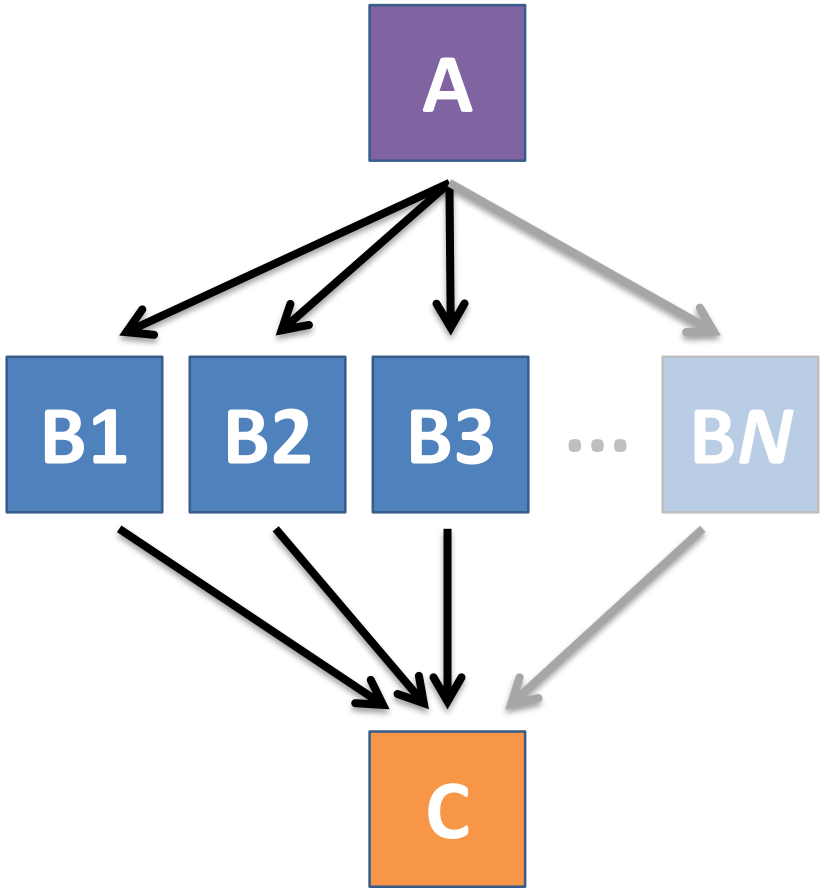


Submitting and Monitoring a DAGMan Workflow

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
```



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.
------------------------------------------------------------------------
```

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 all queued jobs
- ***.dagman.out** has detailed 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

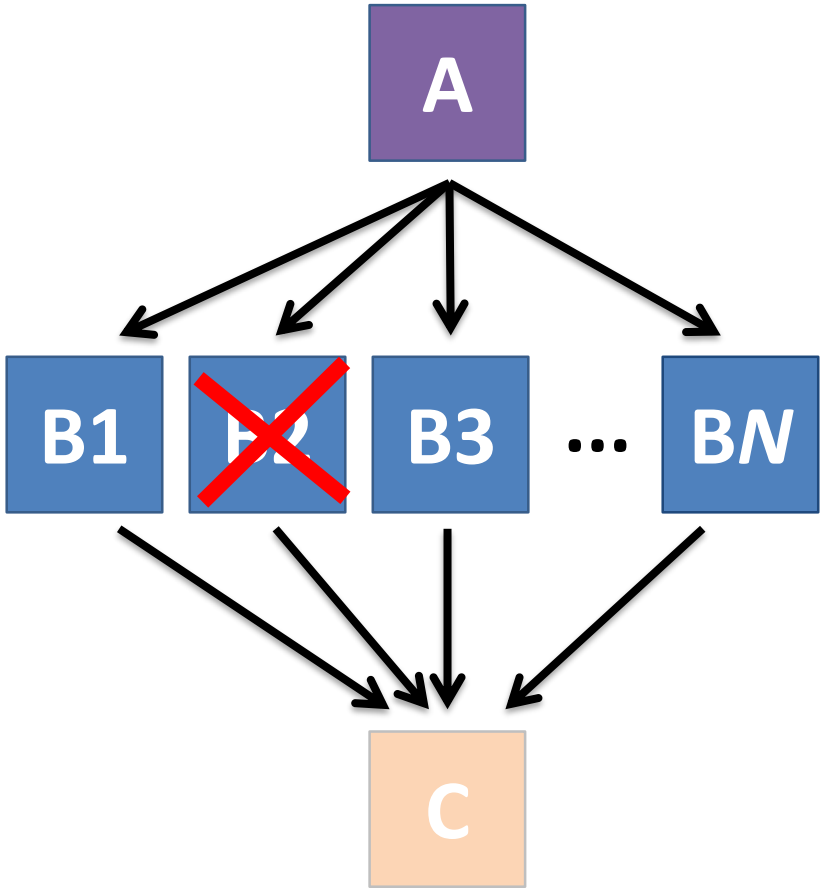
Rescue Files For Resuming a Failed DAG

- 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`

DAGMan > The Rescue DAG

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

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**	

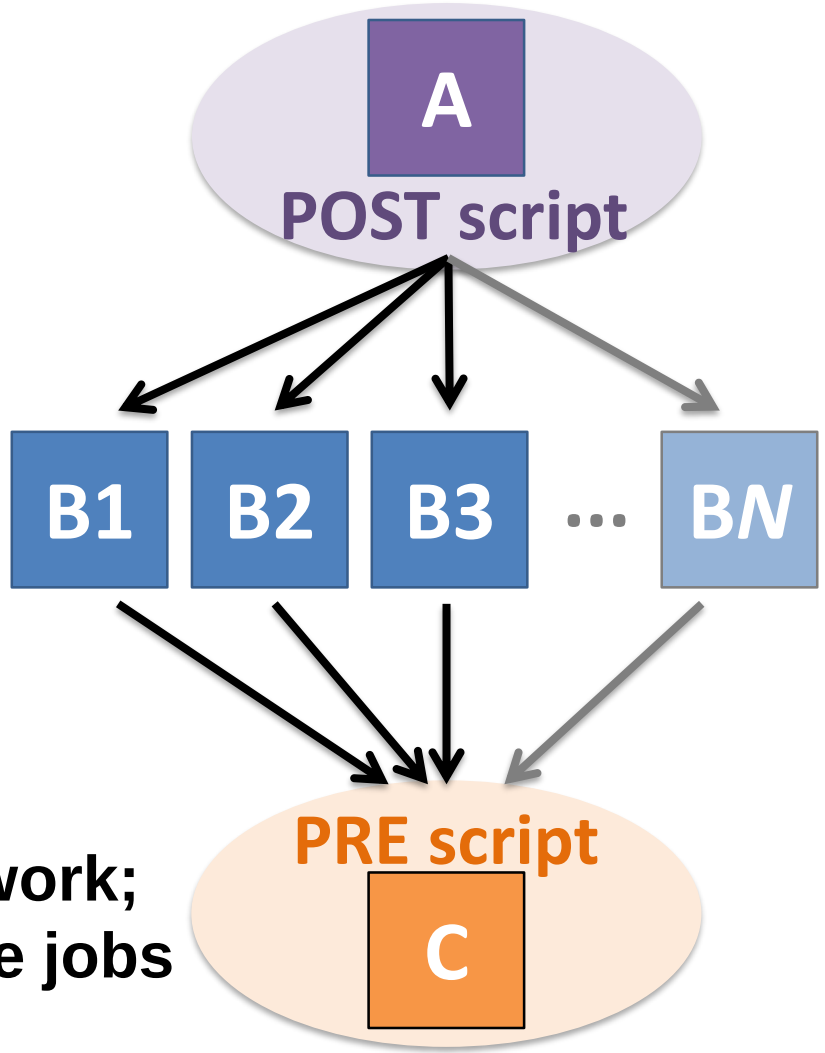
- ***.dagman.metrics** is a summary of events and outcomes
- ***.dagman.log** will note the completion of the DAGMan job
- ***.dagman.out** has detailed logging (look to first for errors)

Beyond the Basic DAG: Some Node-level Modifiers

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**

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
```

- See also: retry except for a particular exit code (`UNLESS-EXIT`), or retry scripts (`DEFER`)
- **Note:** Unnecessary for nodes (jobs) that can use `max_retries` in the submit file

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

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

Modular Organization and Control of DAG Components

- Splices and SubDags
- Node Throttling
- Node Priorities
- Lots more in the Manual…

Additional Resources

- Nice HTCondor FAQs, examples, and documentation from our friends in Canary Islands:

https://is.gd/TjRvY8

- Email list:

http://htcondor.org/mail-lists/

- HTCondor HOWTO Recipes has FAQ on job submission

http://wiki.htcondor.org/index.cgi/wiki?p=HowToAdminRecipes

THANK YOU AND QUESTIONS

ADDITIONAL DAGMAN SLIDES

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
```

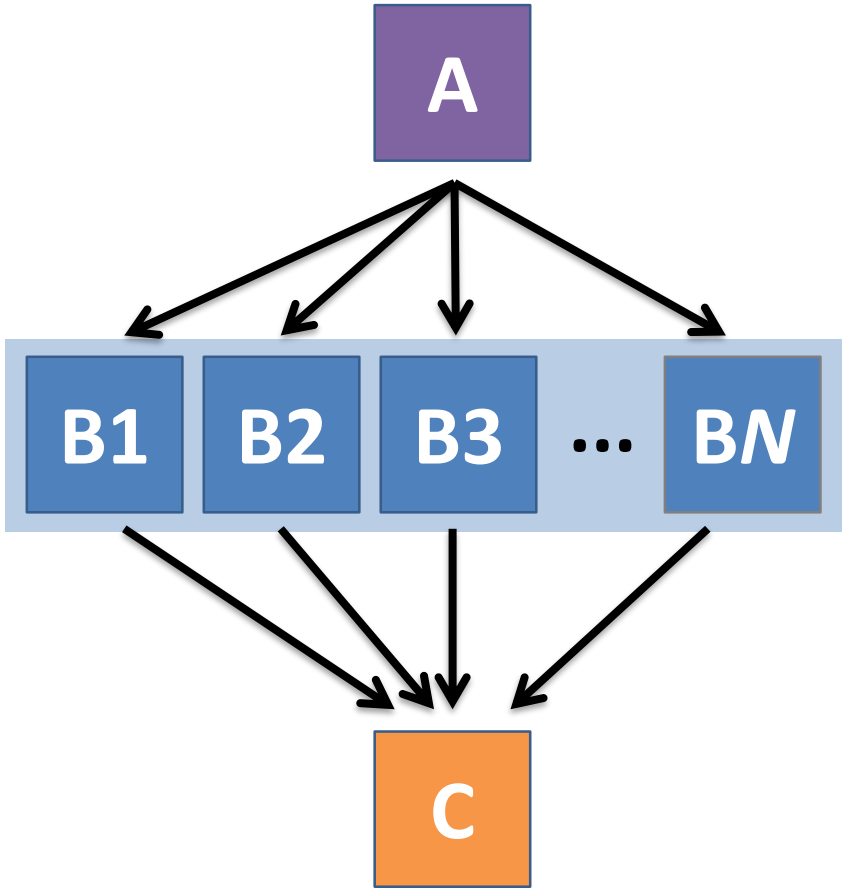
SPLICE groups of nodes to simplify lengthy DAG files

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
```



Use nested SPLICEs with DIR for repeating workflow components

my.dag

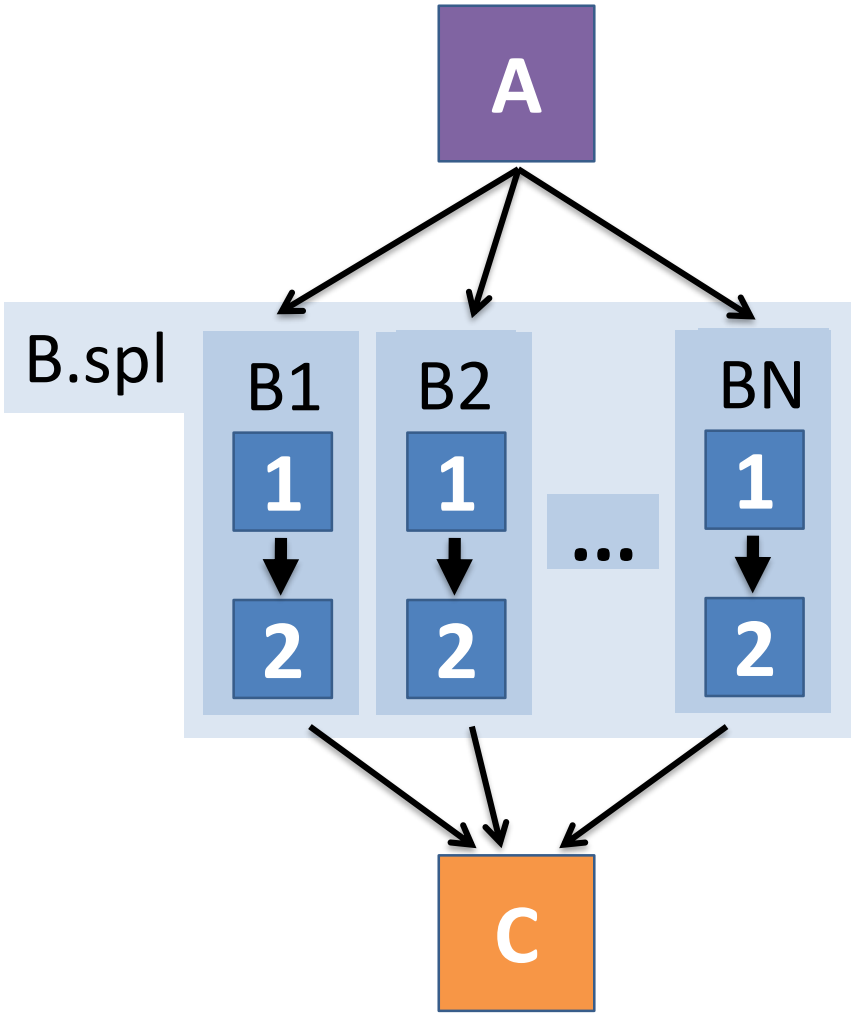
```
JOB A A.sub DIR A
SPLICE B B.spl DIR B
JOB C C.sub DIR C
PARENT A CHILD B
PARENT B CHILD C
```

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 SPLICEs 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
```

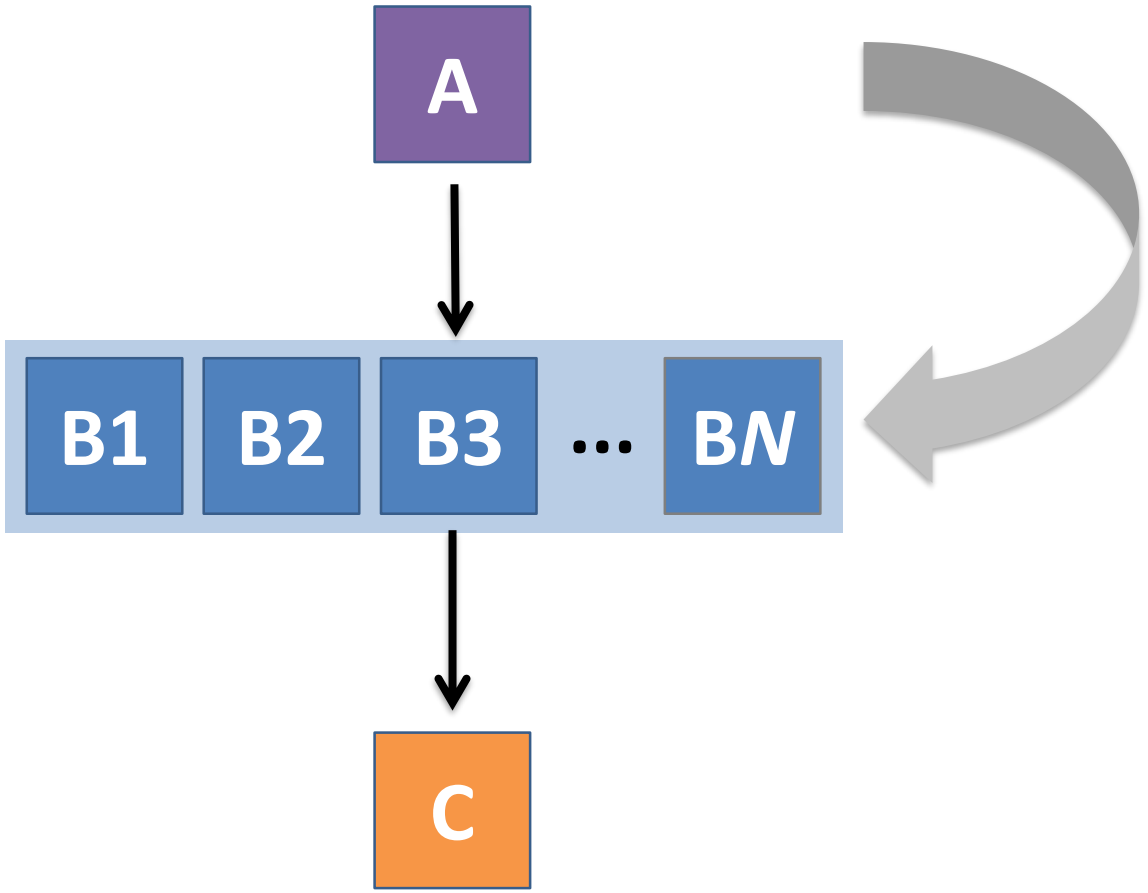
(dag_dir)/

```
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)
```

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)

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** …

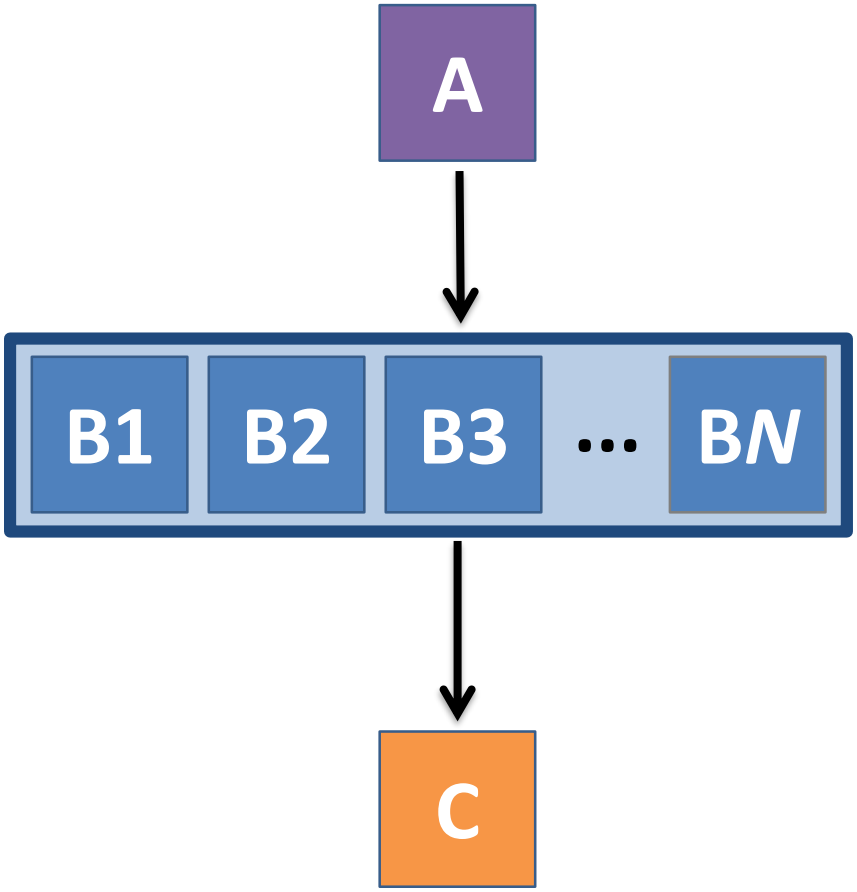
A SUBDAG within a DAG

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
```



More on SUBDAG Behavior

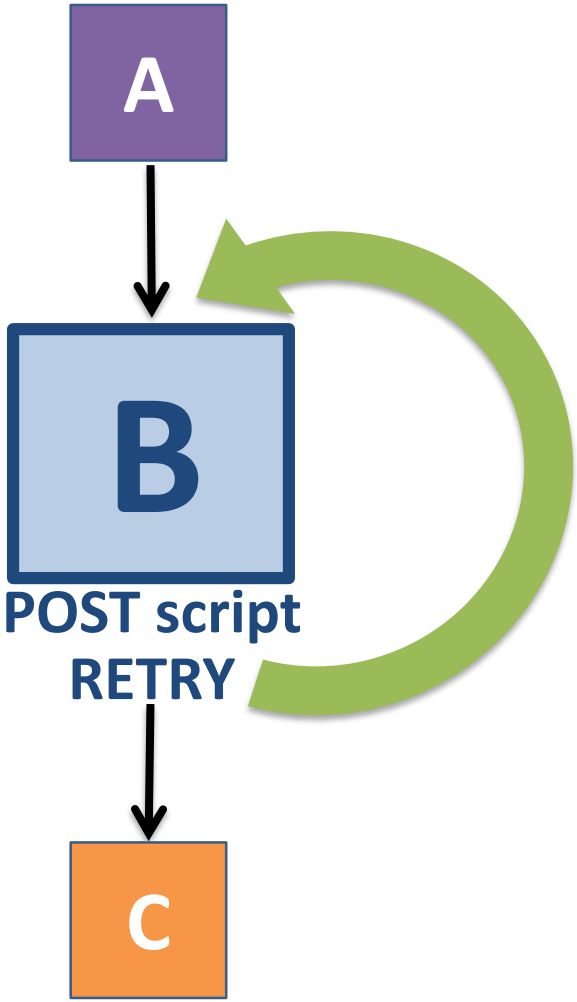
- **WARNING:** SUBDAGs should only be used (over SPLICES) when absolutely necessary!
  - Each SUBDAG EXTERNAL has it's own DAGMan job running in the queue.
- SUBDAGs are nodes (can have PRE/POST scripts, retries, etc.)
- A SUBDAG is not submitted until prior nodes in the outer DAG have completed.

Use a SUBDAG to achieve Cyclic Components 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
```



DAG-level Control

Pause a running DAG with hold/release

- Hold the DAGMan job process:

  `condor_hold` *`dagman_jobID`*

- Pauses the DAG
  - No new node jobs submitted
  - Queued node jobs continue to run (including SUBDAGs), but no PRE/POST scripts
  - DAGMan jobs remains in the queue until released (`condor_release`) or removed

DAGMan > Suspending a Running DAG

Pause a DAG with a halt file

- Create a file named `DAG_file.halt` in the same directory as the submitted DAG file
- Pauses the DAG
  - No new node jobs submitted
  - Queued node jobs, SUBDAGs, and POST scripts continue to run, but not PRE scripts
- DAGMan resumes after the file is deleted
  - If not deleted, the DAG creates rescue DAG file and exits after all queued jobs have completed

DAGMan > Suspending a Running DAG

DAGMan > The Rescue DAG

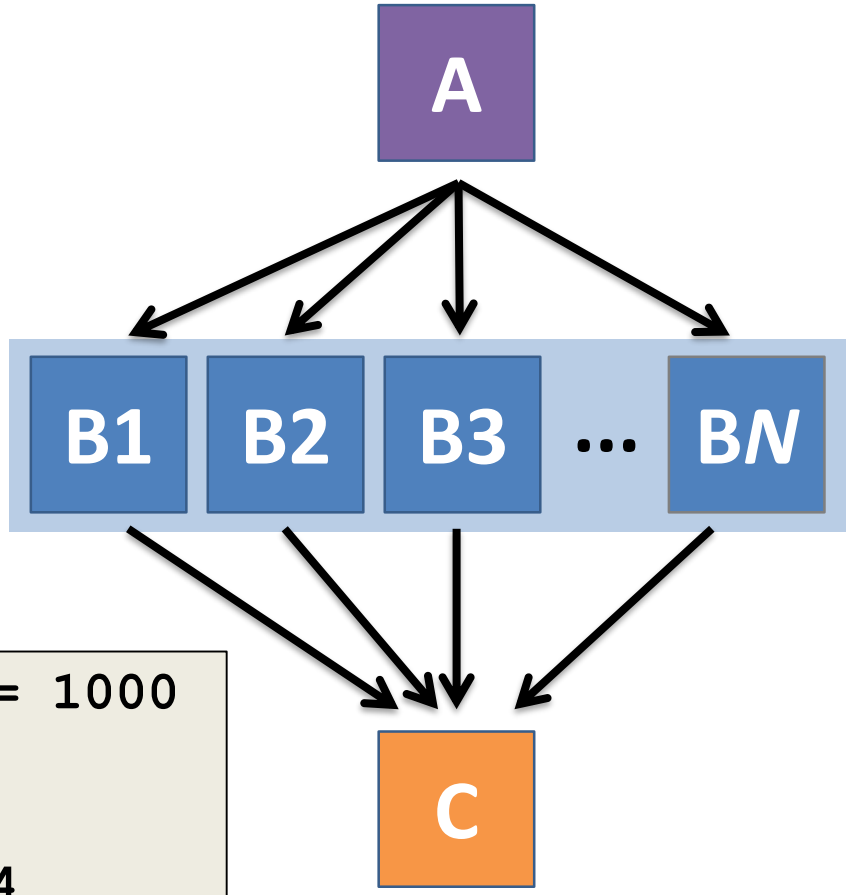
Throttle job nodes of large DAGs via DAG-level configuration

- If a DAG has *many* (thousands or more) jobs, performance of the submit server and queue can be assured by limiting:
  - Number of jobs in the queue
  - Number of jobs idle (waiting to run)
  - Number of PRE or POST scripts running
- Limits can be specified in a DAG-specific **CONFIG** file (recommended) or as arguments to `condor_submit_dag`

DAG-specific throttling via a CONFIG file

my.dag

```
JOB A A.sub
SPLICE B B.dag
JOB C C.sub
PARENT A CHILD B
PARENT B CHILD C
CONFIG my.dag.config
```



my.dag.config

```
DAGMAN_MAX_JOBS_SUBMITTED = 1000
DAGMAN_MAX_JOBS_IDLE = 100
DAGMAN_MAX_PRE_SCRIPTS = 4
DAGMAN_MAX_POST_SCRIPTS = 4
```

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