Submitting Many Jobs at Once

Monday, Lecture 2

Lauren Michael
Questions so far?
Goals for this Session

• Logs, job states, and resource utilization
• Testing and troubleshooting as part of scaling up.
• Best ways to submit multiple jobs (what we’re here for, right?)
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)

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 00:00:00, Sys 0 00:00:00

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:

<table>
<thead>
<tr>
<th>Resource</th>
<th>Usage</th>
<th>Request</th>
<th>Allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cpus</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Disk (KB)</td>
<td>14</td>
<td>20480</td>
<td>17203728</td>
</tr>
<tr>
<td>Memory (MB)</td>
<td>1</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>
Job States

condor_submit

Idle (I) → Running (R) → Completed (C)

transfer executable and input to execute node
transfer output back to submit node

in the queue
leaving the queue
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

<table>
<thead>
<tr>
<th>Partitionable Resources</th>
<th>Usage</th>
<th>Request</th>
<th>Allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cpus</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Disk (KB)</td>
<td>14</td>
<td>20480</td>
<td>17203728</td>
</tr>
<tr>
<td>Memory (MB)</td>
<td>1</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>
Resource Request

- Jobs are nearly always using a part of a machine (a single slot), and not the whole thing.
- Very important to request appropriate resources (*memory, cpus, disk*)
  - **requesting too little**: causes problems for your and other jobs; jobs might be ‘held’ by HTCondor.
  - **requesting too much**: jobs will match to fewer “slots” than they could, and you’ll block other jobs.
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&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
TESTING AND TROUBLESHOOTING
What Can Go Wrong?

• Jobs can go wrong “internally”:
  – the executable experiences an error
• Jobs can go wrong from HTCondor’s perspective:
  – a job can’t be matched
  – a job is missing files
  – 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

<table>
<thead>
<tr>
<th>Log</th>
<th>Output</th>
<th>Error</th>
</tr>
</thead>
</table>
| • When jobs were submitted, started, held, or stopped  
• Resources used  
• Exit status  
• Where job ran  
• Interruption reasons | Any “print” or “display” information from your program (may contain errors from the executable). | Errors captured by the operating system while the executable ran, or reported by the executable, itself. |
**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.

```bash
$ condor_history alice
```

<table>
<thead>
<tr>
<th>ID</th>
<th>OWNER</th>
<th>SUBMITTED</th>
<th>RUN_TIME</th>
<th>ST</th>
<th>COMPLETED</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>189.1012</td>
<td>alice</td>
<td>5/11 09:52</td>
<td>0+00:07:37 C</td>
<td>C</td>
<td>5/11 16:00</td>
<td>/home/alice</td>
</tr>
<tr>
<td>189.1002</td>
<td>alice</td>
<td>5/11 09:52</td>
<td>0+00:08:03 C</td>
<td>C</td>
<td>5/11 16:00</td>
<td>/home/alice</td>
</tr>
<tr>
<td>189.1081</td>
<td>alice</td>
<td>5/11 09:52</td>
<td>0+00:03:16 C</td>
<td>C</td>
<td>5/11 16:00</td>
<td>/home/alice</td>
</tr>
<tr>
<td>189.944</td>
<td>alice</td>
<td>5/11 09:52</td>
<td>0+00:11:15 C</td>
<td>C</td>
<td>5/11 16:00</td>
<td>/home/alice</td>
</tr>
<tr>
<td>189.659</td>
<td>alice</td>
<td>5/11 09:52</td>
<td>0+00:26:56 C</td>
<td>C</td>
<td>5/11 16:00</td>
<td>/home/alice</td>
</tr>
<tr>
<td>189.653</td>
<td>alice</td>
<td>5/11 09:52</td>
<td>0+00:27:07 C</td>
<td>C</td>
<td>5/11 16:00</td>
<td>/home/alice</td>
</tr>
<tr>
<td>189.1040</td>
<td>alice</td>
<td>5/11 09:52</td>
<td>0+00:05:15 C</td>
<td>C</td>
<td>5/11 15:59</td>
<td>/home/alice</td>
</tr>
<tr>
<td>189.1003</td>
<td>alice</td>
<td>5/11 09:52</td>
<td>0+00:07:38 C</td>
<td>C</td>
<td>5/11 15:59</td>
<td>/home/alice</td>
</tr>
<tr>
<td>189.962</td>
<td>alice</td>
<td>5/11 09:52</td>
<td>0+00:09:36 C</td>
<td>C</td>
<td>5/11 15:59</td>
<td>/home/alice</td>
</tr>
<tr>
<td>189.961</td>
<td>alice</td>
<td>5/11 09:52</td>
<td>0+00:09:43 C</td>
<td>C</td>
<td>5/11 15:59</td>
<td>/home/alice</td>
</tr>
<tr>
<td>189.898</td>
<td>alice</td>
<td>5/11 09:52</td>
<td>0+00:13:47 C</td>
<td>C</td>
<td>5/11 15:59</td>
<td>/home/alice</td>
</tr>
</tbody>
</table>
Held Jobs

- HTCondor will put your job on hold if there’s something YOU need to fix.
  - files not found for transfer, over memory, etc.
- 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 until removed, or (fixed and) released.
Diagnosing Holds

• If HTCondor puts a job on hold, it provides a hold reason, which can be viewed in the log file, with `condor_q -hold <Job.ID>`, or with:

```
condor_q -hold -af HoldReason
```

```bash
$ 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
```
Common Hold Reasons

- Job has used **more memory** than requested.
- Incorrect path to files that need to be transferred.
- Badly formatted executable scripts (have Windows instead of Unix line endings).
- Submit directory is **over quota**.
- Job has run for too long. (72 hours allowed in CHTC Pool)
- The admin has put your job on hold.
Fixing Holds

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

  \texttt{condor\_qedit \texttt{\[U/C/J\] \ Attribute \ Value}}

  \$ \texttt{condor\_qedit 128.0 RequestMemory 3072}
  
  Set attribute "RequestMemory".

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

  \texttt{condor\_release \texttt{\[U/C/J\]}}

  \$ \texttt{condor\_release 128.0}
  
  Job 18933774.0 released
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]`
    ```
    $ condor_rm bob
    All jobs of user "bob" have been removed
    $$
    $ condor_rm 128
    All jobs in cluster 128 have been removed
    $$
    $ condor_rm 128.0
    Job 128.0 removed
    ```
SUBMITTING MULTIPLE JOBS
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
  ▪ scale up by breaking up!
  ▪ *we’re learning HTC, right?*

• ...without having to:
  – create separate submit files for each job
  – submit and monitor each job, individually
From one job …

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

```bash
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
Multiple numbered input files

**job.submit**

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

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

queue 3
```

(submit_dir)/

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

job.submit
```

- Generates 3 jobs, but doesn’t change inputs and will overwrite the outputs
- So how can we specify different values to each job?
One submit file per job
(not recommended!)

job0.submit

executable = analyze.exe
arguments = file0.in file0.out
transfer_input_files = file0.in
output = job0.out
error = job0.err
queue 1

(job0.submit/

analyze.exe
file0.in
file1.in
file2.in
(etc.)

job0.submit
job1.submit
job2.submit
(etc.)

(etc...)

job1.submit

executable = analyze.exe
arguments = file0.in file0.out
transfer_input_files = file0.in
output = job0.out
error = job0.err
queue 1
Automatic Variables

Each job’s `ClusterId` and `ProcId` numbers are autogenerated and saved as job attributes.

The user can reference them inside the submit file using:

- `$(ClusterId)`
- `$(ProcId)`

* `$(ClusterId)` and `$(ProcId)` are also okay
Using $(Process) for Numbered Files

```bash
job.submit

executable = analyze.exe
arguments = file$(Process).in file$(Process).out
transfer_input_files = file$(Process).in

log = job_$(Cluster).log
output = job_$(Process).out
error = job_$(Process).err

queue 3
```

$(submit_dir)/

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

job.submit
```

- $(Process) and $(Cluster) allow us to provide unique values to each job and submission!
Organizing Files in Sub-Directories

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

* must be created before the job is submitted
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/
    ```
- Useful for jobs with many shared files; transfer a directory of files instead of listing files individually
  ```
  (submit_dir)/
  ```
  ```
  job.submit
  shared/
  reference.db
  parse.py
  analyze.py
  cleanup.py
  links.config
  ```
Use Paths for File Type

(submit_dir)/

<table>
<thead>
<tr>
<th>job.submit</th>
<th>file0.out</th>
<th>input/</th>
<th>log/</th>
<th>err/</th>
</tr>
</thead>
<tbody>
<tr>
<td>analyze.exe</td>
<td>file1.out</td>
<td>file0.in</td>
<td>job0.log</td>
<td>job0.err</td>
</tr>
<tr>
<td></td>
<td>file2.out</td>
<td>file1.in</td>
<td>job1.log</td>
<td>job1.err</td>
</tr>
<tr>
<td></td>
<td></td>
<td>file2.in</td>
<td>job2.log</td>
<td>job2.err</td>
</tr>
</tbody>
</table>

**job.submit**

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

- **log** = log/job$(Process).log
- **error** = err/job$(Process).err

- **queue 3**
Separating Files by Job with InitialDir

- **Initialdir** sets the initial location for each job’s files, allowing each job to “live” in separate directories on the submit server.
- Allows same filenames for input/output files across jobs.
- Also useful for jobs with lots of output files.
Separating jobs with *initialdir*

<table>
<thead>
<tr>
<th>Directory</th>
<th>file.in</th>
<th>job.log</th>
<th>job.err</th>
<th>file.out</th>
</tr>
</thead>
<tbody>
<tr>
<td>job0/</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>job1/</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>job2/</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**job.submit**

- **executable** = analyze.exe
- **initialdir** = *job$((Process))*
- **arguments** = file.in file.out
- **transfer_input_files** = file.in
- **log** = job.log
- **error** = job.err
- **queue** = 3

**executable** must be relative to the submission directory, and *not* in the InitialDir.
What about non-numbered jobs?

- Back to our compare_states example…
- What if we had data for each state? We could do 50 submit files (or 50 "queue 1" statements) …
What about non-numbered jobs?

• We could rename (map) our data to fit the $(Process)$ or approach …

• Or we could use HTCondor’s powerful `queue` language to submit jobs using our own variables!
# Submitting Multiple Jobs – Queue Statements

## Multiple Submit Files

<table>
<thead>
<tr>
<th>var matching pattern</th>
<th>queue state matching *.dat</th>
</tr>
</thead>
<tbody>
<tr>
<td>var in (i ii iii ...)</td>
<td>queue state in (wi.dat ca.dat co.dat)</td>
</tr>
<tr>
<td>var1, var2 from csv_file</td>
<td>queue state from <code>state_list.txt</code></td>
</tr>
</tbody>
</table>

*state_list.txt:*

- wi.dat
- ca.dat
- mo.dat
- ...

Not Recommended
Using Multiple Variables

• Both the “from” and “in” syntax support multiple variables from a list.

```
job.submit

executable = compare_states
arguments = -y $(year) -i $(infile)

transfer_input_files = $(infile)

queue infile,year from job_list.txt
```

```
job_list.txt

wi.dat, 2010
wi.dat, 2015
cia.dat, 2010
cia.dat, 2015
mo.dat, 2010
mo.dat, 2015
```
## Multiple Job Use Cases – Queue Statements

<table>
<thead>
<tr>
<th>multiple submit files</th>
<th><strong>Not recommended.</strong> Though, can be useful for separating job batches, conceptually, for yourself.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>var</code> matching pattern</td>
<td>Natural nested looping, minimal programming, can use “files” or “dirs” keywords to narrow possible matches. Requires good naming conventions, less reproducible.</td>
</tr>
<tr>
<td><code>var</code> in <code>(i,ii,iii,...)</code></td>
<td>All information contained in the submit file: reproducible. Harder to automate submit file creation.</td>
</tr>
<tr>
<td><code>var1,var2 from csv_file</code></td>
<td>Supports multiple variables, highly modular (easy to use one submit file for many job batches that have different <code>var</code> lists), reproducible. Additional file needed, but can be automated.</td>
</tr>
</tbody>
</table>
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
    ```

- **Combine with InitialDir:**
  
  ```
  InitialDir = $(directory)
  ```
  ```
  queue directory matching dirs job*
  ```
YOUR TURN!
Exercises!

- Ask questions!
- Lots of instructors around

- Coming up:
  - Now-12:15 Hands-on Exercises
  - 12:15 – 1:15 Lunch
  - 1:15 – 5:00 Afternoon sessions