CENTER FOR
HIGH THROUGHPUT
COMPUTING

New condor_submit features in HTCondor 8.3/8.4

John (TJ) Knoeller
Condor Week 2015

The goals

- One submit file – many jobs.
  - Process a bunch of similar files or directories
  - For simple cases - use python bindings for the hard cases
- Address some usability issues
  - Reduce the need to wrap submit in a script
- Pick up some of the 8.2 config features
- Prepare the ground for future scalability

Review – The way it works in 8.2

```
Universe = Vanilla
Executable = cook
Output = meal$(Process).out
Args = -i pasta
Queue
Args = -i chicken
Queue
```

- This produces 2 jobs in the same Cluster, each with different arguments

The new way: Submit 'foreach'

- In 8.3.5/8.4 This submit file can be shortened to

```
Universe = Vanilla
Executable = cook
Output = meal$(Process).out
Args = -i $(Item)
Queue Item in (pasta, chicken)
```

- Identical results to the previous submit

Many ways to Queue 'foreach'

```
Queue <N> <var> in (<item-list>)
Queue <N> <var> matching (<glob-list>)
Queue <N> <vars> from <filename>
Queue <N> <vars> from <script> |
Queue <N> <vars> from (
   <multiline-list>
   )
```

- Iterate <items>, creating <N> jobs for each item
- In/from/matching keywords control how we get <items>
- This is not the full syntax description.

Queue in <item-list>

- 'in' keyword indicates a literal item list
- List is comma and/or space separated
  - Items cannot contain commas or whitespace
  - Items are not otherwise interpreted
- If list begins with '(' it continues to the closing ')'
  - Closing ')' must be on the first line, or on a line by itself.

Example: Queue in

```
Args = $(Item)
Queue 2 in ( alpha, beta delta gamma )
```

- Produces 8 jobs (2 for each item)
- It unrolls to this submit file:

```
Item=alpha
Step=0
Queue
Step=1
Queue
Item=beta
Step=0
Queue
 ...
```

Automatic Loop Variables

- Refer to these variables in your submit file
  - **`$(Process)`** - goes from 0 to #Jobs-1
    - Resets to 0 when $(Cluster) changes
  - **`$(Item)`** - current Item from <items>
    - Exists only if <var> is not specified in Queue line
  - **`$(ItemIndex)`** - goes from 0 to #Items-1
  - **`$(Step)`** - goes from 0 to N-1 (repeatedly)
  - **`$(Row)`** - synonym for **`$(ItemIndex)`**

Queue matching <glob-list>

- Each glob in <glob-list> is matched against filenames relative to the current directory
- Each glob expands to zero or more names
  - Globs that match nothing can produce errors or warnings
- Duplicate filenames are removed.
  - Removal of duplicates can produce errors or warnings
  - Resulting set of <items> is sorted alphabetically
- Some OS's don't support directory globbing

Queue matching files

```
Queue 3 Item matching files (*.dat, m*)
```

- Produces 3 jobs for each file that matches *.dat *or* m* (or both)
- Ignores directories because of optional keyword 'files'
- $(Item) holds each filename in turn

Manipulating filenames

› New macro expansion: $F[pdnxq](Item)
  - Expands file parts from Item where p,d,n,x, and q determine which parts:

p = all directories	d = parent directory
n = basename	x = extension with leading .
q = "" around result	

› Suppose $(Item) is `"/home/work/recipe.lst"`

```
$Fp(Item) is /home/work/ $Fd(Item) is work/
$Fn(Item) is recipe      $Fx(Item) is .lst
$Fnx(Item) is recipe.lst
$F(Item) is /home/work/recipe.lst
```

Example: Queue matching files

Not Windows

```
Universe = Vanilla
Executable = $Fnx(Script)
InitialDir = $Fd(Script)
Queue Script matching files (work*/*.sh)
```

If current directory contains:

```
work1/Fish.sh
work1/Rice.sh
work2/Bacon.sh
```

3 jobs will be submitted with:

```
Executable = Fish.sh   InitialDir = work1/
Executable = Rice.sh   InitialDir = work1/
Executable = Bacon.sh  InitialDir = work2/
```

Example: Queue matching dirs

```
Universe = Vanilla
Executable = $ENV(HOME)/cook.sh
Queue InitialDir matching dirs *
```

NOT OSX/UNIX

If current directory contains:

```
Fish/
Rice/
Bacon!/
```

3 jobs will be submitted with:

```
InitialDir = Fish/
InitialDir = Rice/
InitialDir = Bacon!/
```

Queue from : Lines are Items

```
Queue from <filename>
```

- Read <filename> and treat lines as items

```
Queue from <script> |
```

- Execute <script> and treat output lines as items

```
Queue from (
   <item>
   <item>
   ...
   )
```

- Read submit file, treating lines as items

Queue from allows multiple vars

```
Args = -m $(Method) -- $(Items)
Queue Method,Items from (
   Braise Carrots
   Grill  Steak
   Bake   Bread Cake
   )
```

- Produces 3 jobs, one for each line
- each line is tokenized on space and/or comma until all but last variable have a value.
- Last variable gets remainder of the line.

Commenting out Queue items.

```
Queue from (
  <item1>
 # <item2>
  ...
  )
```

- When item list is read directly from the submit file, the usual submit file rules for comments and line-continuation apply.
  - Lines that begin with # are ignored.
  - Lines that end with \ are continued on the next line. (remember that lines are items…)

Slicing

- Python style slice [start:end:step] syntax.
- Only jobs which have $(ItemIndex) within the slice are submitted

```
Queue Item in [:1] (Alpha, Beta
   Delta Gamma
   )
```

This slice selects only ItemIndex==0
so only Alpha jobs are submitted

Formatted Numbers

```
$INT(<name>|<math>[,<printf-format>])
$REAL(<name>|<math>[,<printf-format>])
```

- Lookup <name> and evaluate it, or just evaluate <math>.

› Result is printed using <printf-format>

```
Output = out_$INT(Process,%06d)
MyId = $(ItemIndex) + $(Step)/1000.0
Args = -id $REAL(MyId,%.4f) -delay $INT(12*60)

Output = out_000021
Args = -id 2.0010 -delay 720
```

CENTER FOR HIGH THROUGHPUT COMPUTING

HTCondor

Choice

`$CHOICE(<index-name>|<math>,<list-name>)`

- Lookup <index-name> and evaluate it, or just evaluate <math>. Then use as an index into <list-name> to extract a single item.

```
Args = $CHOICE(Step,Items)
Queue 2 Executable,Items from (
  Braise Carrots,Beets
  Grill  Steak,Chicken
)
```

comma is required by $CHOICE

<N> as expression

```
Queue 4*5 Item in (alpha beta)
```

- For Queue <N>, <N> can be an expression

It can refer to command line attributes

```
> condor_submit cook.sub num=2
> cat cook.sub
...
Queue $(num:0)*2
```

Command line attributes

- Any argument with = in the middle is treated as a submit attribute assignment
- Parsed *before* the submit file is read
  - Can be used in Queue or If statements

```
> condor_submit cook.sub trial=1
> cat cook.sub
Executable = cook
If $(trial)
  Hold = $(Process) > 0
endif
...
```

Condor_submit -queue

- Only if submit file has no Queue statement
- It should be the *last* argument. because:

```
condor_submit cook.sub -queue in *.dat
```

- Item list can be read from standard input

```
dir /b *.dat | condor_submit cook.sub -que from -
```

Condor_submit -dry-run

```
condor_submit cook.sub -dry-run cook.ads
```

- writes to job to cook.ads instead of Schedd
- $(Cluster) is always 1
- First Ad has all job attributes for ProcId=0
- Remaining Ads have only attrs that differ

```
condor_submit cook.sub -dry -
```

- Quickly see what Queue 'foreach' will do

Circuit breakers

```
condor_submit -maxjobs <number>
```

- Fail the submit if number of jobs would exceed <number>

```
condor_submit -onecluster
```

- Fail the submit if more than one cluster would be created
  - For automated submission tools like DAGMan

Any Questions?