Getting To Multicore: Resizing jobs in CMS

Brian Bockelman

HTCondor Week 2016

WARNING:
Abundant ClassAd Expressions Ahead

A problem of partitioning…

- Within CMS, we've been busily working to enable multithreading in our physics application, CMSSW.
  - We can now run about half of our applications
- Application in hand, we must ask ourselves how we roll it out in our historically-single core pool.
  - Some workflows are multicore, some are singlecore-only.
    - **The multicore workflows can be submitted as single-core or multicore.**
  - Some sites are multicore, some sites are multicore.
- How do we partition our workflow such that we fully utilize all our cores?

The Global Pool and Multicore

- The CMS HTCondor pool - the “global pool” consists of HTCondor startds allocated from about 50 CMS sites around the planet. Everything goes into a single condor_collector.
    - About 70%* of the startds are 8-core partitionable slots; the rest are single-core startds.
    - The single-core startds are due to sites that have not yet reconfigured their batch system. Some are opportunistic and **will never be converted to multicore**.
- Multicore allows us:
    - Reduce per-core memory requirements.
    - Reduce the number of jobs (both in the schedd and in the upstream CMS infrastructure).
    - Decrease total runtime of jobs (48 hour single-core jobs may only take 16 hours on 4-core slots).

* number increases almost daily

What to do?

Inject all our multicore workflows as multicore, make the most of our multicore advantages?

Submit all our multicore jobs as single core to maximize site utilization?

Split the workflows 50/50 and hope we don’t idle some of the cores?

Flip a coin?

The Resizable Job

- Fundamentally, partitioning our workflows leads to inefficiency as **we will inevitably do it wrong.**
    - Even if CPUs don’t idle, we’re almost guarantee to have priority inversion: we’ll run low-priority singlecore jobs while high-priority workflow waits because it was injected as multicore.
- New idea: **the resizable job**.
    - A job that will match a range of resources, and reconfigure itself at runtime based on the resources allocated.
    - Examples to follow

That's not an integer, it's an expression!

- The key insight here is the `RequestedCpus` attribute can be an expression. Hence, both of these are valid inputs:

```
RequestedCpus = 1
RequestedCpus = 1 + 1
```

- We can do something even more interesting:

```
RequestedCpus = Cpus
```

Goes into job ad

Refers to `Cpus` attribute from machine ad

What does it mean?

```
RequestCpus = Cpus
```

- What does this expression mean?
  - When the matchmaker considers a machine for a certain job, `RequestCpus` evaluates to be the same as the number of available cores.
  - Hence, we will match against any number of cores, 1 to 100.
- What happens when `RequestCpus` is evaluated outside the negotiator?
  - Whoops…

```
RequestCpus = isUndefined(Cpus) ? 1 : Cpus
```

Example

Job

```
ClusterId = 1
ProcId = 0
Cmd = "…"
RequestMemory = 1000
Requirements = true
RequestCpus =
    isUndefined(Cpus)?
    1 : Cpus
```

Machine

```
Name = "slot1@ex.com"
State = "Unclaimed"
PartitionableSlot = true
Activity = "Idle"
Cpus = 3
Memory = 2000
```

Result

```
Cpus = 3
Memory = 2000
```

Consider the Context

- There's a few rules we'd like these jobs to obey:
  - Make sure jobs stay within a certain range.

```
MinCores = 1
MaxCores = 4
RequestResizedCpus = (Cpus > MaxCores) ?
                      MaxCores :
                     ((Cpus < MinCores) ? MinCores : Cpus)
```

  - When idle, report `RequestCpus` as the value originally requested by the user.

```
RequestCpus = !isUndefined(Cpus) ?
                RequestResizedCpus :
                JobCpus)
```

Consider the Context

- When running, report `RequestCpus` as the value allocated on the worker node.

```
OriginalCpus = 4
GlideinCpusIsGood = !isUndefined(MATCH_EXP_JOB_GLIDEIN_Cpus) &&
                    (int(MATCH_EXP_JOB_GLIDEIN_Cpus) isnt error)
JOB_GLIDEIN_Cpus = "$$(Cpus:0)"
JobIsRunning = (JobStatus =!= 1) &&
               (JobStatus =!= 5) &&
               GlideinCpusIsGood
JobCpus = JobIsRunning ?
          int(MATCH_EXP_JOB_GLIDEIN_Cpus) :
          OriginalCpus
```

In technicolor

```
MinCores = 1
MaxCores = 4
RequestResizedCpus = (Cpus > MaxCores) ?
                      MaxCores :
                     ((Cpus < MinCores) ? MinCores : Cpus)
OriginalCpus = 4
GlideinCpusIsGood = !isUndefined(MATCH_EXP_JOB_GLIDEIN_Cpus) &&
                    (int(MATCH_EXP_JOB_GLIDEIN_Cpus) isnt error)
JOB_GLIDEIN_Cpus = "$$(Cpus:0)"
JobIsRunning = (JobStatus =!= 1) &&
               (JobStatus =!= 5) &&
               GlideinCpusIsGood
JobCpus = JobIsRunning ?
          int(MATCH_EXP_JOB_GLIDEIN_Cpus) :
          OriginalCpus
RequestCpus = !isUndefined(Cpus) ?
                RequestResizedCpus :
                JobCpus)
```

Other Attributes

- Some aspects of the job request are independent of the number of cores, such as the gigabytes of output.
- Others will vary with the number of CPUs allocated:
  - `RequestMemory`: Each additional core used adds ~500MB of RAM to the application.
  - Estimated wall time: The estimated wall time should scale with the inverse of number of cores.
  - We want to `Rank` potential matches by preferring as many cores as possible and minimizing fragmenting very large slots.

Other Attributes

```
EstimatedSingleCoreMins = 120
MaxWallTimeMins =
    (EstimatedSingleCoreMins/RequestCpus + 15)
OriginalMemory = 4000
RequestMemory =
   OriginalMemory +
   500 * (RequestCpus-OriginalCpus)
Rank = isUndefined(Cpus) ?
       0 :
       ifThenElse(Cpus > MaxCores, -Cpus, Cpus)
```

Runtime

- The `condor_starter` creates a copy of the machine ClassAd in the file referenced by `$_CONDOR_MACHINE_AD`.
    - The CMS job wrapper parses this file and adjust’s CMSSW’s configuration appropriately.
    - This way, at startup, CMSSW knows the number of cores to utilize.

Why give this talk?

- Please don't copy/paste from the slides!
- I want to provide two take-aways:
    - The `RequestCpus` attribute - normally an integer - can actually express a policy.
    - A few **highly-nontrivial** examples of ClassAds in action

Where Next?

- Resizable jobs allow us to pack jobs more efficiently into a startd with “irregularly-sized” idle resources.
  - Let’s say we are trying to drain a startd. Can we make remaining jobs run faster?
    - As jobs finish up, can we add CPUs to those still running?
  - CMSSW’s threading framework *could* be modified to dynamically change the number of active processing threads.
- *Dynamic* resizable jobs: the next frontier?

Questions?