Scalable, Parallel Video Transcoding on Ubuntu

Transcoding video is a very resource intensive process.

It can take many minutes to process a small, 30-second clip, or even hours to process a full movie. There are numerous, excellent, open source video transcoding and processing tools freely available in Ubuntu, including libav-tools, ffmpeg, mencoder, and handbrake. Surprisingly, however, none of those support parallel computing easily or out of the box. And disappointingly, I couldn't find any MPI support readily available either.

I happened to have an Orange Box for a few days recently, so I decided to tackle the problem myself, and develop a scalable, parallel video transcoding solution myself. I'm delighted to share the result with you today!

When it comes to commercial video production, it can take thousands of machines, hundreds of compute hours to render a full movie. I had the distinct privilege some time ago to visit WETA Digital in Wellington, New Zealand and tour the render farm that processed *The Lord of the Rings* triology, *Avatar*, and *The Hobbit*, etc. And just a few weeks ago, I visited another quite visionary, cloud savvy digital film processing firm in Hollywood, called Digital Film Tree.

Windows and Mac OS may be the first platforms that come to mind, when you think about front end video production, Linux is far more widely used for batch video processing, and with Ubuntu, in particular, being extensively at both WETA Digital and Digital Film Tree, among others.

While I could have worked with any of a number of tools, I settled on avconv (the successor(?) of ffmpeg), as it was the first one that I got working well on my laptop, before scaling it out to the cluster.

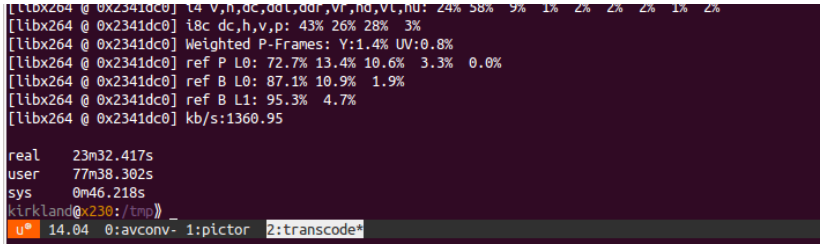
I designed an approach on my whiteboard, in fact quite similar to some work I did parallelizing and scaling the john-the-ripper password quality checker.

At a high level, the algorithm looks like this:

1. Create a shared network filesystem, simultaneously readable and writable by all nodes
2. Have the master node split the work into even sized chunks for each worker
3. Have each worker process their segment of the video, and raise a flag when done
4. Have the master node wait for each of the all-done flags, and then concatenate the result

And that's exactly what I implemented that in a new transcode charm and transcode-cluster bundle.
It provides linear scalability and performance improvements, as you add additional units to the cluster. A transcode job that takes 24 minutes on a single node, is down to 3 minutes on 8 worker nodes in the Orange Box, using Juju and MAAS against physical hardware nodes.

For the curious, the real magic is in the *config-changed* hook, which has decent inline documentation.



The trick, for anyone who might make their way into this by way of various StackExchange questions and (incorrect) answers, is in the command that splits up the original video (around

line 54):

```
avconv -ss $start_time -i $filename -t $length -s $size -vcodec libx264 -acodec aac -
bsf:v h264_mp4toannexb -f mpegts -strict experimental -y
${filename}.part${current_node}.ts
```

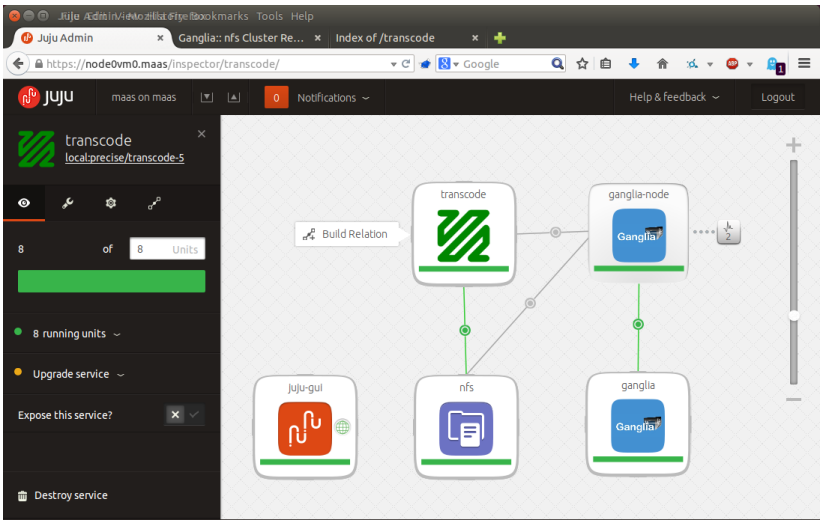
And the one that puts it back together (around line 72):

```
avconv -i concat:"$concat" -c copy -bsf:a aac_adtstoasc -y
${filename}_${size}_x264_aac.${format}
```

I found this post and this documentation particularly helpful in understanding and solving the problem.

In any case, once deployed, my cluster bundle looks like this. 8 units of transcoders, all connected to a shared filesystem, and performance monitoring too.

I was able to leverage the *shared-fs* relation provided by the *nfs* charm, as well as the *ganglia* charm to monitor the utilization of the cluster. You can see the spikes in the cpu, disk, and network in the graphs below, during the course of a transcode job.



For my testing, I downloaded the movie *Code Rush,* freely available under the CC-BY-NC-SA 3.0 license. If you haven't seen it, it's an excellent documentary about the open source software around Netscape/Mozilla/Firefox and the dotcom bubble of the late 1990s.

Oddly enough, the stock, 746MB high quality MP4 video doesn't play in Firefox, since it's an mpeg4 stream, rather than H264. Fail. (Yes, of course I could have used *mplayer*, *vlc*, etc., that's not the point ;-)

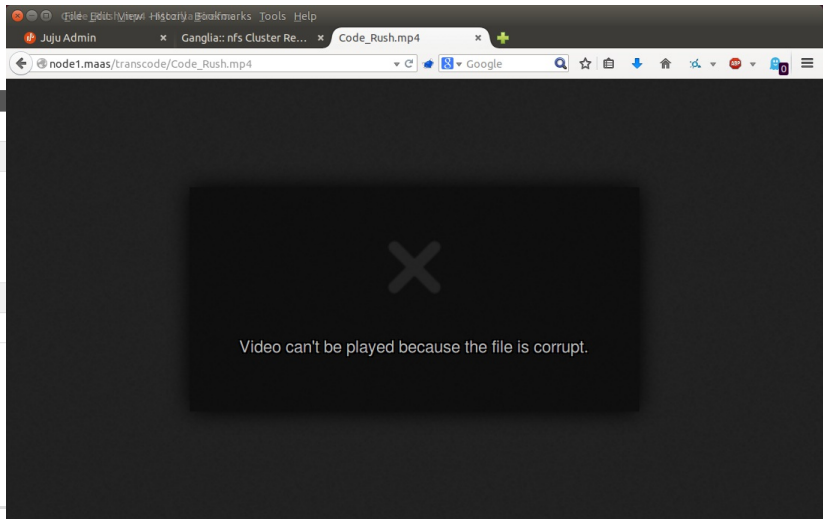
Perhaps one of the most useful, intriguing features of HTML5 is it's support for embedding multimedia, video, and sound into webpages. HTML5 even supports multiple video formats. Sounds nice, right? If it only were that simple... As it turns out, different browsers have, and lack support for the different formats. While there is no one format to rule them all, MP4 is supported by the majority of browsers, including the two that I use (Chromium and Firefox). This matrix from w3schools.com illustrates the mess.

Video Formats and Browser Support

Currently, there are 3 supported video formats for the <video> element: MP4, WebM, and Ogg:

Browser	MP4	WebM
Internet Explorer	YES	NO
Chrome	YES	YES
Firefox	YES **Update 1:** Firefox 21 on Windows and Android now supports MP4 **Update 2:** Firefox 30 on Linux now supports MP4	YES
Safari	YES	NO
Opera	NO	YES

- MP4 = MPEG 4 files with H264 video codec and AAC audio codec
- WebM = WebM files with VP8 video codec and Vorbis audio codec
- Ogg = Ogg files with Theora video codec and Vorbis audio codec



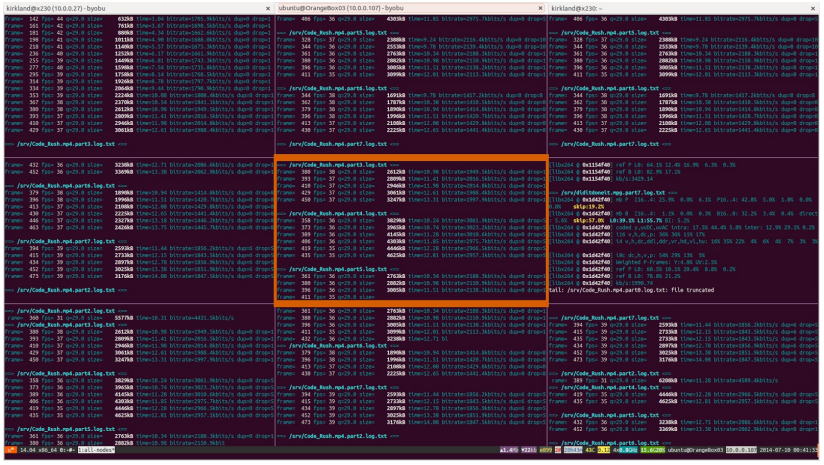
http://www.w3schools.com/html/html5_video.asp

The file format, however, is only half of the story. The audio and video contents within the file also have to be encoded and compressed with very specific codecs, in order to work properly within the browsers. For MP4, the video has to be encoded with H264, and the audio with AAC.

Among the various brands of phones, webcams, digital cameras, etc., the output format and codecs are seriously all over the map. If you've ever wondered what's happening, when you upload a video to YouTube or Facebook, and it's a while before it's ready to be viewed, it's being transcoded and scaled in the background.

In any case, I find it quite useful to transcode my videos to MP4/H264/AAC format. And for that, a scalable, parallel computing approach to video processing would be quite helpful.

During the course of the 3 minute run, I liked watching the avconv log files of all of the nodes, using Byobu and Tmux in a tiled split screen format, like this:



Also, the *transcode* charm installs an Apache2 webserver on each node, so you can expose the service and point a browser to any of the nodes, where you can find the input, output, and intermediary data files, as well as the logs and DONE flags.

Once the job completes, I can simply click on the output file, Code_Rush.mp4_1280x720_x264_aac.mp4, and see that it's now perfectly viewable in the browser!

In case you're curious, I have verified the same charm with a couple of other OGG, AVI, MPEG, and MOV input files, too.

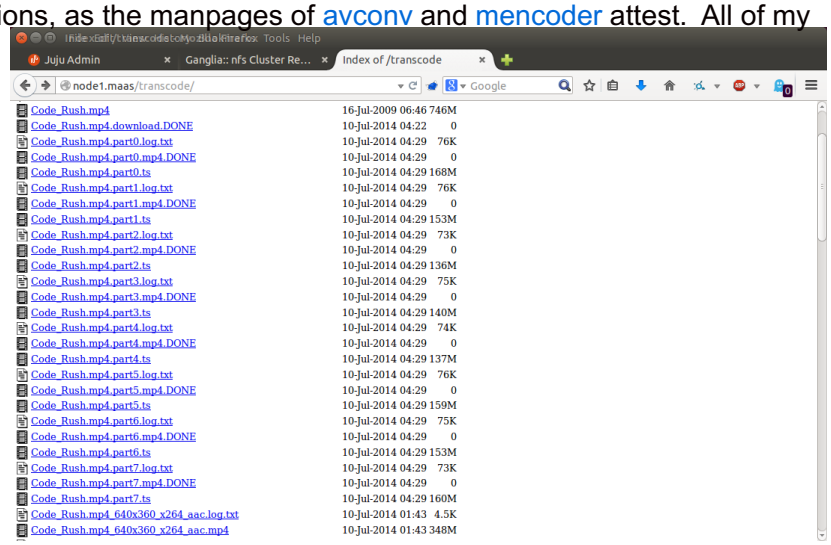
Beyond transcoding the format and codecs, I have also added configuration support within the charm itself to scale the video frame size, too. This is useful to take a larger video, and scale it down to a more appropriate size, perhaps for a phone or tablet. Again, this resource intensive procedure perfectly benefits from additional compute units.

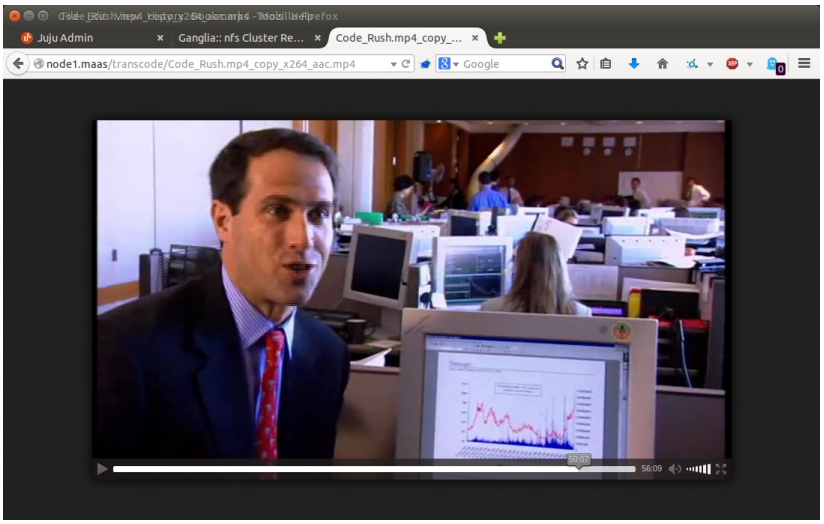
File format, audio/video codec, and frame size changes are hardly the extent of video transcoding workloads. There

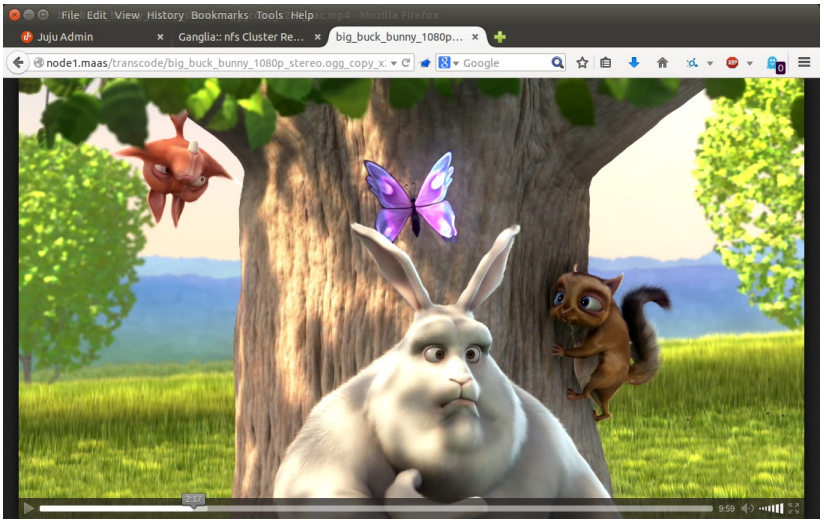
are hundreds of options and thousands of combinations, as the manpages of avconv and mencoder attest. All of my scripts and configurations are free software, open source. Your contributions and extensions are certainly welcome!

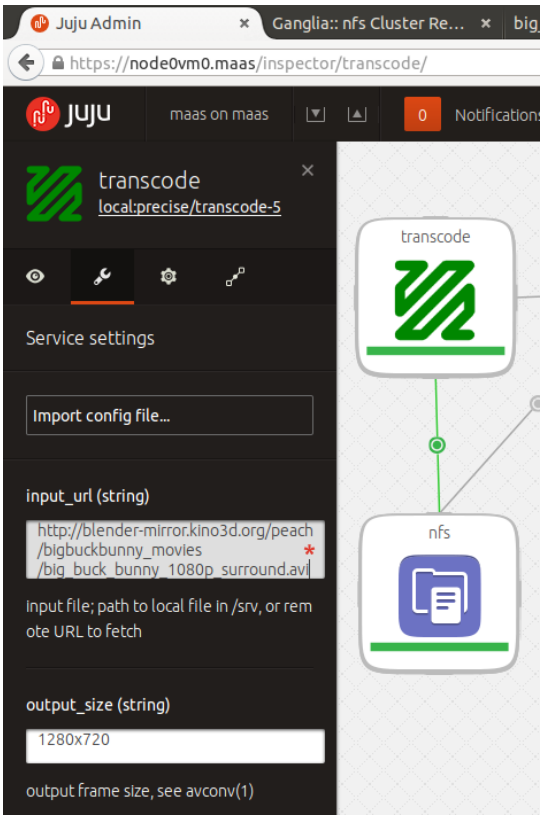
In the mean time, I hope you'll take a look at this charm and consider using it, if you have the need to scale up your own video transcoding ;-)

Cheers,
Dustin








Juju Admin
Ganglia:: nfs Cluster Re...
https://node0vm0.maas/inspector/transcode/
juju
maas on maas
0
Notifications
transcode
local:precise/transcode-5
Service settings
Import config file...
input_url (string)
http://blender-mirror.kino3d.org/peach
/bigbuckbunny_movies
/big_buck_bunny_1080p_surround.avi
input file; path to local file in /srv, or rem
ote URL to fetch
output_size (string)
1280x720
output frame size, see avconv(1)
transcode
nfs