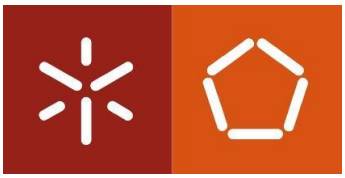

Universidade do Minho

Departamento de Informática

Trabalho Prático
Computação Avançada I

Filipe Monteiro (a80229) Leonardo Silva (pg39261)

21 de Janeiro de 2020

Conteúdo

1 Introdução

Neste projeto foi requerido que utilizássemos os conhecimentos adquiridos durante o semestre da cadeira de **Computação Avançada I** sobre computação utilizando *clusters*, com a submissão de *Jobs* utilizando o **HTCondor**. Neste particular exercício era pedido a criação de um *Job* capaz de diminuir a resolução de um video, submetendo assim ao nosso *cluster*.

2 Arquitetura do Cluster

O *cluster* foi desenvolvido utilizando o programa **HTCondor**. Com este, foi necessário definir IP's fixos para todas as máquinas e definir o tipo de políticas e responsabilidades de cada uma. Utilizando as configurações disponibilizadas pelo professor, construimos 1 *Master* e 2 *Slaves*. O *Master* é o responsável por gerir os recursos e *Jobs*, podendo submeter e também executar processamentos. Este possui 1 CPU e 2Gb de RAM. Os *Slaves* são recursos disponíveis para executar processamentos, possuindo cada um 1 CPU e 1 Gb de RAM. Estes estão todos ligados por cabo (virtual), com IP's fixos para poderem ser acedidos/usados. Isto tudo foi cirado usando o *CentOs7* em várias máquinas virtuais.

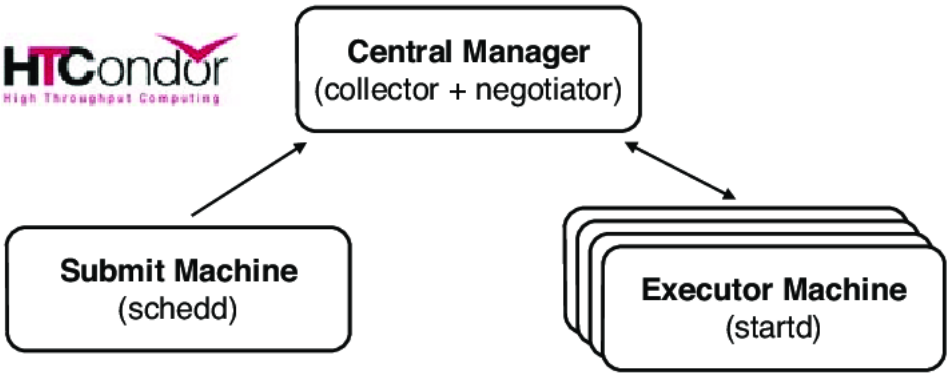

Figura 1: Estrutura do **HTCondor**

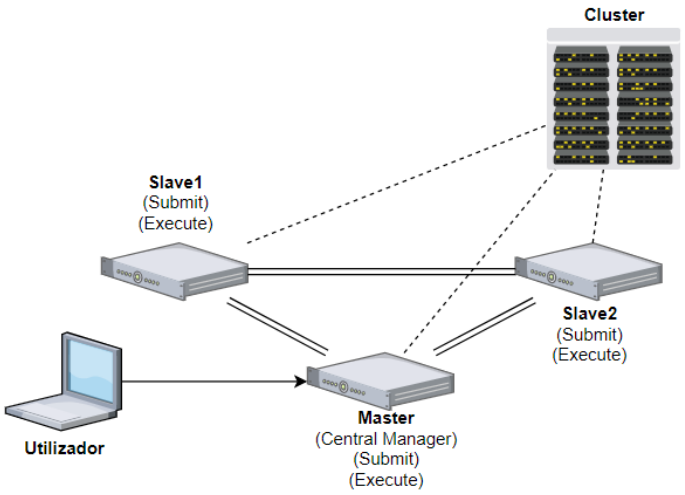

Figura 2: Arquitetura do *cluster*, utilizando o *HTCondor*

3 *Scaling* de vídeos

Para tirar partido dos recursos disponibilizados pelo *cluster* decidimos dividir o vídeo inicial em segmentos mais pequenos, distribui-los pelos recursos para converter e no fim, após recepção de todos estes, juntar todos eles para criar um vídeo, agora em menor resolução. O ponto de corte para cada segmento é 1 segundo de conteúdo, ficando no fim, em teoria, com N segmentos (com um vídeo de duração N segundos). Isto tudo usando o programa *ffmpeg*.

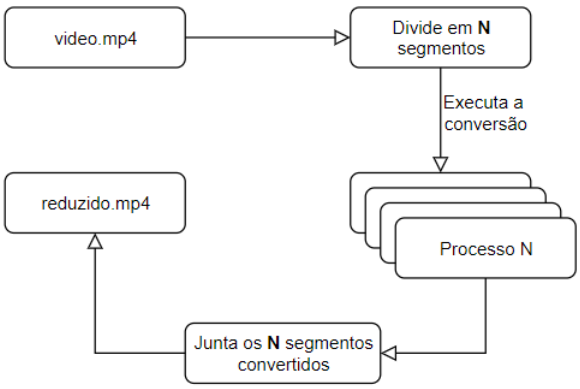

Figura 3: *Flow* do programa.

Contudo, devido à forma de corte utilizada, em vez de N segmentos, são gerados menos, devido ao facto de o corte ser feito de forma crua (sem *decode e recode*). Assim, alguns contêm 1 segundo e outros poderão ter mais uns segundos, dependendo do número de *frames*.

O *Job* é submetido através de um DAG, ou seja, um conjunto de computações que exigem computações dependentes de outras. Neste caso usamos um simples grafo para definir o *Job*:

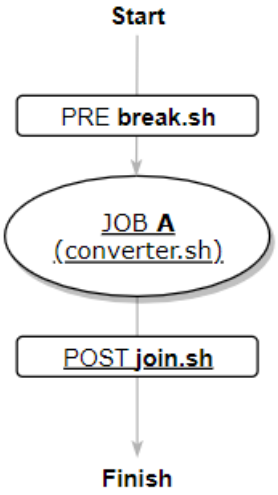

Figura 4: Grafo DAG do projeto.

```
JOB A converter.condor

SCRIPT PRE A break.sh video10m.mp4
SCRIPT POST A join.sh mp4 video10m720.mp4
```

Listing 1: fulljob.dag

Aqui verificámos que o *JOB A* (reponsável por converter o ficheiro) tem 1 *script* para executar antes deste e outro para executar no fim. O primeiro, break.sh é *script* em *shell* que divide o video de N segundos em N segmentos diferentes, começados por "0.xxx", "1.xxx", ..., (xxx é a extensão do ficheiro original) sendo que os números de cada um será importante mais tarde para a atribuição de cada ficheiro ao *JOB A*. Devido ao problema descrito em cima, criámos ainda outro *script* (duration.sh) que calcula o número de segmentos que serão gerados (é importante na nossa *approach* ao problema saber o número de ficheiros para *input* que serão gerados) assim como o tempo em segundos que tem.

```
#!/bin/sh

IFS='.' read -r -a array <<< "${1}"

mkdir logs
COM="ffmpeg -i ${1} -c copy -map 0 -segment_time 5 -f segment -reset_timestamps 1
    %01d.${array[1]}"
exec $COM
```

Listing 2: break.sh

De seguida, o *JOB A* executa um programa em *shell* (converter.sh) constituido de N processos iguais de diferente *input*: cada processo terá um respetivo *input* (0.xxx por exempl) e dará como resultado um outro ficheiro - por exemplo, **out0.xxx.ts** - que corresponde ao segmento passado, de menor qualidade e convertido num formato para mais tarde facilitar a junção de todos eles para formar o ficheiro final.

```
Universe = vanilla
Executable = converter.sh
Arguments = $(Process).mp4 720
Log = logs/converter.$(Process).log
Output = logs/converter.$(Process).out
Error = logs/converter.$(Process).error

should_transfer_files = yes
when_to_transfer_output= ON_EXIT
transfer_input_files = $(Process).mp4
transfer_output_files = out$(Process).mp4.ts

periodic_remove = (CommittedTime - CommittedSuspensionTime) > 150
request_cpus = 1
request_memory = 100 MB
request_disk = 50 MB
Queue 122
```

Listing 3: converter.condor (*JOB* principal)

Para além dos campos *standards* utilizados na descrição do *Job*, adicionamos alguns requisitos como obrigar a ter 1 CPU, 50 MB de memória livre e 50 MB de espaço no disco (para caso

seja fornecido um ficheiro demasiado grande). Repare-se que será através do **$(Process)** que se especifica os *inputs*, *logs* e ficheiros a receber de cada processo.

```
#!/bin/sh

COM="ffmpeg -y -i $1 -vf scale=-2:$2,setsar=1:1 -c:v libx264 -c:a copy"
OUT=" out${1}"
FIRST=$COM$OUT

SND="ffmpeg -i out${1} -c copy -bsf:v h264_mp4toannexb out${1}.ts"

#ffmpeg -i "out${1}" -c copy -bsf:v h264_mp4toannexb "out${1}.ts"

eval $FIRST
eval $SND
```

Listing 4: converter.sh

Em relação à conversão de cada segmento, de forma a minimizar a perda, utilizámos o *codec* **libx264** com valores *default* na precisão da conversão, resultando numa conversão fidedígna com um bom balanço no tamanho do ficheiro. Para selecionar a resolução utilizámos o argumento *-vf scale* onde indicamos a altura como, por exemplo, 720 no caso de um vídeo **HD**, e o comprimento é ajustado de forma a manter um *aspect ratio* correto. Por fim, para não haver erros futuros na agregação dos ficheiros, estes são convertidos para um ficheiro intermédio .ts.

Por fim, após executar os processos todos, terminando assim o *JOB A*, é executado o *script* join.sh. Este, começa por chamar um *script python* que gera uma lista com todos os segmentos, convertidos, ordenados pelo número do corte para ser usada pelo *ffmpeg* no momento de os juntar todos e fazer o video final.

```
#!/bin/sh

mkdir output

eval "python join.py ${1}"

ffmpeg -f concat -safe 0 -i list.txt -c copy -bsf:a aac_adtstoasc "output/${2}"
```

Listing 5: join.sh

```
from os import listdir
from os.path import isfile, join
import sys

dirs = listdir(".")
onlyfiles=[]

for file in dirs:
   if isfile(join(".",file)) and ".ts" in file:
      onlyfiles.append(file)

final=[]
for i in range(0,len(onlyfiles)):
   final.append("file 'out"+str(i)+"."+sys.argv[1]+".ts'")
```

```
with open('list.txt','w') as f:
   for item in final:
      print >> f, item
```

Listing 6: join.py

4 Exemplos

4.1 fragmento.mp4, FullHD, 10s

Utilizando como primeiro exemplo o vídeo fornecido pelo professor, numa resolução 1080p (**FullHD**) e de duração 10 segundos, executámos a conversão do documento para 720p (**HD**) utilizando apenas a máquina que submete o trabalho (localmente) e depois com o *cluster*.
Convertendo localmente conseguimos finalizar à volta de 54 segundos. Contudo, com a utilização do *cluster*, este subiu para 57-59 segundos. Era de esperar que tal acontecesse, visto que o processo de corte, troca de ficheiros entre máquinas, conversões e junção dos ficheiros, num vídeo tão pequeno, colocam um *overhead* demasiado grande na tarefa.

4.2 video5m,mp4, FULLHD, 5m

Neste exemplo utilizamos 1 video de duração 5 minutos, em **FullHD**, e o objetivo era reduzir a resolução deste para 720p (**HD**). Executando apenas na máquina *Master*, este demorou entre 25 a 26 minutos a concluir a conversão, enquanto que utilizando o *Cluster*, reduziu para mais ao menos metade do tempo (9-12 minutos). Aqui, comparando com o exemplo anterior, verifica-se um grande benefício da utilização do *cluster*, devido à distribuição da carga por várias máquinas, de forma atómica ao problema em causa. No caso do *cluster*, foram gerados 64 processos diferentes distribuidos pelas 3 máquinas.

Um facto interessante é que, o *master* era quem tinha sempre mais carga comparadamente com o resto das máquinas, talvez pela rapidez com que executava os seus processos, não exigindo tantas trocas dos dados, pela rede.

5 Conclusão

Este trabalho proporcionou nos a capacidade de aprender a criar, gerir e utilizar sistemas *cluster* para fins de processamento de ficheiros de grande dimensão para reduções temporais. Verificamos que a possibilidade de colocar programas a serem executados por várias máquinas, consoante os recursos disponíveis no sistema, é incrível e aconselhada. O facto de os programas não serem corridos na nossa máquina, tem vários aspectos positivos, sendo que, entre estes, devido ao processamento ser feito noutras máquinas, a nossa fica liberta para outras acções, sendo que assim, o processamento é realizado num ambiente bem mais poderoso e eficaz, mais rápido e sem bloquear a minha máquina. Outra parte importante é a possibilidade do programa puder executar por períodos de tempo extensos mas sem nunca bloquear os recursos, pois estes podem ir rodando conforme decisão da máquina *Master* do *cluster*. A tarefa proposta concluímos ser um bom *use case* para a utilização de um sistema *cluster*, onde quanto maior for o ficheiro a converter, mais rápido será comparativamente ao uso de apenas recursos locais. E apesar da estruturação do programa para se adequar ao uso do *cluster* ter sido complicada no início, ao fim de alguma prática esta tornou-se trivial. Contudo, nós apenas utilizámos a superfície das capacidades da totalidade do *HTCondor*. Por fim, para concluir, pensámos que numa tentativa futura de melhorar a *performance* do problema em questão seria, por exemplo, fornecer a todos os processos o mesmo vídeo de *input*, indicando que parte devem cortar (segmentar), para converterem e retornar, tirando a tarefa de segmentar o vídeo da máquina que submete o trabalho.