Tarea 5 : Programación Distribuida

Lluch, Pablo
pablo.lluch@gmail.com

Fuentes, Tomás
tafuentesc@gmail.com

7 de diciembre de 2013

1. Solución

Para solucionar el problema planteado, se optó por utilizar el código base en Java entregado a los alumnos. La solución está en dos carpetas distintas, al igual que el código base: En demo se encuentra la clase Executor, que fue ligeramente modificada para poder trabajar con un parámetro extra, que especifica el problema que se quiere resolver (bonus del enunciado). En demo_base se encuentra el código correspondientes a los problemas y trabajadores de la red adhoc.

El problema se carga por reflection y se le pide al usuario por consola, inmediatamente después de que se le pide elegir si quiere cargar una matriz de adyacencia aleatoria o bien una prefabricada. Lo único que se debe especificar es el nombre de la clase que se desea utilizar, el cual debe estar en la misma carpeta que los otros archivos del código de demo_base. El único requisito es que la clase debe derivar de la interfaz Task, la cual contiene declaraciones de métodos genéricos. Como ejemplo, se implementó la clase QuicksortTask, la cual implementa un quicksort distribuido.

Los métodos que requiere la interfaz task son:

1. int getAnswerCount(); // Corresponde al número de respuestas que debería esperar a recibir el proceso principal. En el caso de QuicksortTask, corresponde al número de elementos del arreglo.

2. $publicArrayList < Task > getNextTasks(Object[]currentResults)$; // Es el principal método de ejecución. Recibe como parámetro un arreglo de Objects correspondientes a la solución actual, el cual puede contener cualquier tipo de objeto. En el caso de QuicksortTask, son enteros. Puede tener un número arbitrario de elementos. Retorna un ArrayLists de Tasks que corresponden a los próximo subtasks que deben ejecutar otros procesos en paralelo. Nuevamente, para el caso de QuicksortTask, es un ArrayList de dos elementos correspondientes a las dos llamadas recursivas de Quicksort.

   El resto de los miembros de Task pueden ser a gusto del usuario y la clase del problema puede definir lo necesario para resolverlo. Quicksort define un arreglo arbitrario y algunas cosas como pivotes e índices.

2. Simplificaciones

Para el correcto funcionamiento del programa, se hicieron algunas simplificaciones, no por falta de capacidad pero más bien porque se consideró que no era necesario tener en cuenta todos los formatos posibles de imágenes y videos para tener una buena demostración del funcionamiento.

1. La imagen de prueba está en formato RGB, donde sus elementos están descritos por un float entre 0 y 255
2. El video de prueba tiene audio stereo

3. Algoritmo

El algoritmo consiste en los siguientes pasos:

3.1. Procesamiento del video

Primeramente, se extraen los frames y el sonido del video original de forma separada. Al tener el audio en un archivo a parte, se permite ecualizar tranquilamente el video sin modificar el audio. La extracción de los frames se hace utilizando el programa ffmpeg, invocado por Java externamente, y se asume que se encuentra en el computador del usuario. Mediante diversas líneas de comando, se pueden ejecutar las funcionalidades deseadas. Ellas son:

1. Extracción del audio - /usr/bin/ffmpeg -y -i input/short.mp4 -vn -ac 2 -f mp3 input/short.mp3
2. Extracción de frames - /usr/bin/ffmpeg -y -i input/short.mp4 -r 25.0 -ss 0 -t 40 -q:v 1 input/image- %3d.jpeg
3. Unión de frames (después de la ecualización) - /usr/bin/ffmpeg -start_number 1 -y -i input/image- %3d.jpeg -r 25.0 -vcodec mjpeg -q:v 1 input/equalized.mp4

En general:

- -i = input
- -r = frame rate

- -y = reemplazar
- -ss = tiempo de comienzo
- -t = duración
- -vcodec, -acodec = codecs utilizados
- -f = formato de audio
- -q:v, q:a = calidad del video y audio

3.2. Ecualización

Para la ecualización del video, se utiliza el clásico algoritmo descrito en [**?**]. Clásicamente, con imágenes de tonos grises, este algoritma funciona obteniendo, primero, la frecuencia de aparición de cada nivel de gris en una imagen. Una imagen con buen contraste se obtiene haciendo que el histograma cumulativo de tonos en una imagen sea aproximadamente lineal.

El problema resulta cuando se quiere tratar con imágenes en colores, en donde ecualizar por canal usando RGB, por ejemplo, tiene malos resultados pues la información del contraste viene entremezclada entre los canales. Para solucionar este problema, se decidión utilizar uno de los approach mencionados en [**?**], que consiste en primero transformar la imagen al espacio de colores esférico mencionado en [**?**]. Esto tiene la gracia de que el canal r resume bastante bien la información del contraste, pues reune información de los 3 canales RGB originales. Luego, se puede ecualizar sobre este canal para luego transformar de vuelta a RGB y rearmar el video.

Como nota a parte, debido a problemas de las liberías usadas para trabajar con ciertos rangos de valores en imágenes, al ecualizar imágenes en coordenadas esféricas se escala por un factor para no tener una imagen con rangos de valores exacerbadamente altos. Esto es un detalle interno pero no afecta el algoritmo de ecualización.

3.3. Paralelismo

Por supuesto, la gracia de esto es que casi todos los procedimientos son altamente paralelos. Considerando el algoritmo descrito, en la transformación de espacio de colores, el valor final de un pixel es independiente de aquellos

que lo rodean. Por el otro lado, para el cálculo de histogramas, aunque sí hay mayor dependencia, se puede calcular el histograma por partes. En este caso, se decidió calcular primero el histograma por columnas, dado que generalmente las imágenes cuentan con más columnas que filas. Al final se pueden sumar estos histogramas para obtener el histograma final. Para el cálculo de histogramas y transformación de colores, se utilizaron diversas funciones de kernel con OpenCL. El uso de este mismo se hizo a través de un wrapper de OpenCL para Java, JOCL [**?**], que facilitó la implementación de los diversos algoritmos y maneja buffers, kernels y otros elementos de OpenCL utilizando un approacho orientado a objetos.

3.4. Formación del nuevo video

Para cumplir con esta función, se reintegraron los frames procesados por OpenCL con la metodología descrita anteriormente, utilizando ffmpeg. Una posibilidad habría sido paralelizar también esta parte, aunque la complejidad agregada era mayor pues habría requerido meterse con encodings de video, por lo que se rechazó esta opción. Otro motivo por el cual se decidión no paralelizar este proceso es que usando ffmpeg externamente, el mayor problema resulta ser el input y output con el disco duro, por lo que las ganancias de paralelizar este proceso (incluso via CPU, por ejemplo) habrían sido bastante menores.

Para la integración del audio, simplemente se unen los tracks de audio y video ecualizado a través de los métodos en la clase Video.java, que utilizan, nuevamente, ffmpeg.