UTEM
UNIVERSIDAD TECNOLÓGICA METROPOLITANA

Proyecto Procesamiento de imágenes

Integrantes

Gonzalo Salinas

Ignacio Valdes

Camila Carrasco

Asignatura

Computación paralela y distribuida

Docente

Sebastian Salazar

Entrega

Miércoles, 12 de Agosto de 2020

Introducción

En la actualidad con el uso de nuevas tecnologías surge la necesidad de poder implementar un concepto que garantice que los recursos permanecerán siempre disponibles, así como también que la carga de trabajo en el sistema sea balanceada.

Una solución a esta necesidad es la implementación de cluster de computadoras, la cual consiste en un grupo de sistemas independientes, conocidos como nodos que trabajan como un sistema único conectado mediante una red de alta velocidad. Los clusters son usualmente empleados para mejorar el rendimiento y/o la disponibilidad por encima de la que es provista por un solo computador.

Esta tecnología es aplicada en diversas áreas relacionadas al cálculo numérico, simulación, problemas científicos y procesamiento de imágenes. Estos procesos requieren de muchos recursos de cómputo, principalmente de tiempo y espacio. Por ello, el repartir el proceso en una máquina paralela de memoria distribuida, pretende una reducción de estos tiempos.

La información contenida dentro de este informe describe el desarrollo de tres operaciones de tratamiento de imágenes mediante la utilización de algoritmos paralelos. El primero consiste en la difuminación de imágenes con el uso de filtro gaussiano, la segunda consiste en la conversión de una imagen en color a escala de grises y finalmente la operación de escalado de imagen.

Las operaciones descritas anteriormente son posible gracias al desarrollo de los lenguajes de programación que soportan la utilización de hilos y procesos, nos referimos especialmente el lenguaje C y C++ que a través de las librerías MPI permite crear aplicaciones paralelas con uso de memoria distribuida.

Tecnología utilizada

Para el desarrollo del proyecto se realizó el diseño e implementación de un cluster de procesamiento de imágenes con el uso de MPI, el cual proporciona una librería de funciones para C y C++, las que son empleadas en los programas para comunicar datos entre procesos. Popularmente al momento de realizar cómputo en paralelo y con el uso de cluster se tiende a la utilización de esta librería, ya que permite que diversos procesos puedan correr en paralelo en varios nodos dentro de nuestro cluster creado.

El standard MPI permite definir tanto la sintaxis como la semántica del conjunto de rutinas que pueden ser utilizadas en la implementación de programas que usen pasaje de mensajes.

Para lograr el paso de mensajes es fundamental el uso de dos funciones, la primera será la encargada de enviar el mensaje a un proceso determinado y la segunda recibe el mensaje en el proceso. El entorno del mensaje debe contar principalmente con el identificador del proceso receptor del mensaje, el identificador del proceso emisor del mensaje, una etiqueta y un comunicador
Se destaca principalmente de MPI el estándar de comportamiento de implementaciones de manera concisa, la más importante hace referencia a la garantía de seguridad en la transmisión de mensajes, lo cual libera al usuario de comprobar si los mensajes son recibidos correctamente.

Instalación de Requerimientos

Para la utilización del software es necesario el compilador GCC empacado de serie para C++, el cual se incluye en la instalación de las siguientes librerías:

Linux:

```
sudo apt update
sudo apt upgrade
apt-get install openjdk-8-jdk git-core build-essential libopenmpi-dev openssh-server
sudo apt install libopencv-dev python3-opencv
sudo apt-get install mpi-default-dev
```

En caso de requerir make (opcional):

Si por algún motivo se necesita realizar una compilación del **main.cpp**, sitúese en la carpeta **paralelo** e ingrese el siguiente comando:

```
make clean && make
```

Manual de uso

Para la utilización del programa, usted deberá colocar la imagen a la cual desea aplicar los cambios en la carpeta **paralelo/Imagenes**. Además de la carga de la imagen, también es necesaria la información de las máquinas que trabajarán en conjunto en el Cluster, para esto existe un archivo llamado **maquinas.txt** en el cual se debe ingresar dicha información de la siguiente manera (ejemplo):

```
192.168.0.2 slots=3
```

Como se ve en la línea anterior, en el txt se debe incluir la ip de la máquina y los slots o procesadores que utilizara de esta, si se desea, se pueden incluir más máquinas esclavas. Luego, debe situar la consola de comandos en la carpeta **paralelo,** y ejecutar los siguientes comandos, dependiendo de la operación que desee realizar:

Para el difuminado de la imagen:

```
mpirun --hostfile maquinas.txt ./dist/programa 1 Imagenes/imagen.jpg
#O en su defecto, Sin mpi:
./dist/programa 1 Imagenes/imagen.jpg
```

Para realizar escalado de grises:

```
mpirun --hostfile maquinas.txt ./dist/programa 2 Imagenes/imagen.jpg
#O en su defecto, Sin mpi:
./dist/programa 2 Imagenes/imagen.jpg
```

Para escalar la imagen al doble del tamaño:

```
mpirun --hostfile maquinas.txt ./dist/programa 3 Imagenes/imagen.jpg
#O en su defecto, Sin mpi:
./dist/programa 3 Imagenes/imagen.jpg
```

El programa realizará la operación correspondiente y retornará la imagen resultante en la carpeta **paralelo** en formato png.

Lógica interna y funcionamiento

Primero se explicarán las distintas funciones que utiliza el programa principal.

Función guardarImagenResultado:

```
204   void guardarImagenResultado(Mat image, string operation){
205       time_t rawtime;
206       struct tm * timeinfo;
207       char buffer[80];
208       time (&rawtime);
209       timeinfo = localtime(&rawtime);
210       strftime(buffer,sizeof(buffer),"%Y%m%d%I%M%S",timeinfo);
211       std::string str(buffer);
212       imwrite("operacion_"+operation+"_"+str+".png",image);
213   }
```

Esta función realiza el guardado de la imagen procesada por el programa en la carpeta **paralelo**. En las líneas 205 a 208 se designan todas las variables necesarias, luego, en la línea 210 se guarda la hora en el buffer con el formato indicado en el string "%Y%m%d%I%M%S". La línea 211 simplemente convierte la información del buffer a string y finalmente, la línea 212 guarda la imagen con toda la información previamente preparada.

Función copiarTrozoImagen:

```
215   void copiarTrozoImagen(Mat ientrante, Mat idestino, int primx, int primy, int ultix, int ultiy)
216   {
217       for(int x = 0; x<ultix-primx; x++){
218           for(int y = 0; y<ultiy-primy; y++){
219               idestino.at<Vec4b>(y,x)[0] = ientrante.at<Vec3b>(y,x+primx)[0];
220               idestino.at<Vec4b>(y,x)[1] = ientrante.at<Vec3b>(y,x+primx)[1];
221               idestino.at<Vec4b>(y,x)[2] = ientrante.at<Vec3b>(y,x+primx)[2];
222               idestino.at<Vec4b>(y,x)[3] = 255;
223           }
224       }
225   }
```

Como su nombre lo sugiere, esta función realiza una copia de una parte de la imagen y la guarda en una variable. Para realizar esta operación, es necesario utilizar un bucle de dos For, con la intención de recorrer toda la imagen en base a los datos de entrada. Los datos de entrada describen la posición inicial y final de la copia de la imagen, en base a las filas y columnas en el archivo Mat "ientrante". Los datos de entrada "primx" y "ultix" seleccionan el rango de columnas a copiar mientras que "primy" y "ultiy" el rango de filas. En las líneas 219 a 222 del código, se copian los datos de cada canal de la imagen original a la de destino.

Función enviarImagen:

```
227  void enviarImagen(Mat imagenParaEnvio, int idestino){
228      int sizes[3];
229
230      sizes[2] = imagenParaEnvio.elemSize();
231      Size s = imagenParaEnvio.size();
232      sizes[0] = s.height;
233      sizes[1] = s.width;
234      MPI_Send( sizes, 3, MPI_INT,idestino,0,MPI_COMM_WORLD);
235      MPI_Send( imagenParaEnvio.data, sizes[0]*sizes[1]*4, MPI_CHAR,idestino,1, MPI_COMM_WORLD);
236  }
```

Función orientada a la comunicación entre los procesadores que trabajan en la misma operación con la imagen. Como el nombre lo sugiere, esta función realiza la comunicación de envío de datos a un destinatario, el cual se indica en el parámetro de entrada "idestino". En las líneas 228 a 233 se definen los datos del arreglo sizes que describen la imagen a enviar en lo que a dimensiones se refiere, luego esta información es usada en la línea 234 del código, donde se envía mediante la función MPI_Send. La línea 235 del código realiza el envío de la data de la imagen, de esta forma, se realiza el envío completo de la imagen en dos operaciones.

Función recibirImagen:

```
238  void recibirImagen(Mat &imagenParaRecibir,int ientrante){
239      MPI_Status estado;
240      int sizes[3];
241      MPI_Recv( sizes,3, MPI_INT,ientrante,0, MPI_COMM_WORLD, &estado);
242      imagenParaRecibir.create(sizes[0], sizes[1], CV_8UC4);
243      MPI_Recv( imagenParaRecibir.data, sizes[0] * sizes[1] * 4, MPI_CHAR, ientrante, 1, MPI_COMM_WORLD, &estado);
244  }
```

Esta función realiza la operación contraria a la anterior, recibiendo los dos datos entregados (líneas de código 241 y 243) y guardándolas en la variable de entrada de la imagen que se usa de recipiente (puntero).

Programa principal:

```
72   int main(int argc, char** argv ){
73
74       if(argc > 2)///Detecta los argumentos de ingreso,
75       {
76           //Definicion de variables, rank sera el numero
77
78           int rank, procesadores;
79           Mat imagen;
80           //Se inicia el MPI
81           MPI_Init(&argc, &argv);
82           MPI_Comm_rank(MPI_COMM_WORLD, &rank);
83           MPI_Comm_size(MPI_COMM_WORLD, &procesadores);
84           string opcion(argv[1]);
```

En la imagen anterior, las líneas 78 a 79 es donde se realiza la creación de las variables principales que utilizara el programa, en rank se guarda la identificación del procesador actual y en procesadores el número de estos. La variable imagen se utilizará para guardar la imagen que ingrese el usuario al programa. En las líneas 81 a 83 se realiza la inicialización de mpi y en la línea 84 se guarda la opción ingresada por el usuario de la operación a realizar.

```
87      if(rank == 0)
88      {
89          imagen = imread(argv[2],IMREAD_COLOR); //Guarda en la variable tipo mat "imagen", la lectura
90          int dimensionParticion = imagen.cols / procesadores; //Divide la imagen a procesar en partes
91          int bordeSuperior = dimensionParticion; //Limite superior inicial para trabajar la imagen.
92          int bordeInferior = 0; //Limite inferior inicial para trabajar la imagen.
93          trozoImagen.create(imagen.rows,dimensionParticion, CV_8UC4);//Se crea una trozo de imagen ini
94          copiarTrozoImagen(imagen,trozoImagen,0,0,dimensionParticion,imagen.rows);//Se copia del trozo
95
96          for(int proce = 1; proce < procesadores; proce++)//Bucle destinado a repartir el trabajo entr
97          {
98              bordeInferior = (dimensionParticion * proce); // En base al numero del procesador se desi
99              bordeSuperior = (dimensionParticion * (proce + 1));
100             if(proce+1 == procesadores)
101             {
102                 bordeSuperior = imagen.cols;
103             }
104             int diferencia = bordeSuperior - bordeInferior;
105             Mat imagenParaEnvio(Size(diferencia, imagen.rows), CV_8UC4); /// Se realiza el la creacio
106             copiarTrozoImagen(imagen, imagenParaEnvio, bordeInferior, 0, bordeSuperior, imagen.rows);
107             enviarImagen(imagenParaEnvio, proce);// Se envia la imagen para su procesamiento al proce
108         }
109     }
110     else
111     {
112         recibirImagen(trozoImagen,0);// Recepcion del trabajo o los trozos de la imagen a procesar po
113
114     }
```

En la imagen anterior se muestra la parte del programa principal destinada a la repartición de la imagen a procesar en los distintos procesadores que trabajan en la operación. Primero se debe calcular y dividir los trozos de la imagen, esto se realiza en las líneas 90 a 94 para el maestro y en las líneas 96 a 107 para los esclavos. El else final de la imagen está destinado a los esclavos, los cuales estarán esperando recibir el trabajo del maestro.

```
118         if(opcion == "1" || opcion == "2")
119         {
120             resultadoParcial = trozoImagen.clone();
121         }
122         else if(opcion == "3"){
123             int columnasAgregadas = trozoImagen.cols * 2.0;
124             int filasAgregadas = trozoImagen.rows * 2.0;
125             resultadoParcial.create(filasAgregadas, columnasAgregadas, CV_8UC4);
126         }
```

Luego, se debe crear el “espacio de trabajo” de cada procesador o trabajador, el cual se realiza en la imagen anterior, según la opción del tipo de trabajo a realizar.

```
129         if(opcion=="1")
130         {
131             int factorh = round((7*trozoImagen.rows)/600);//Factor de difuminado de la imagen en base al tamaño vertical de esta.
132             int factorw = round((7*trozoImagen.cols)/600);//Factor de difuminado de la imagen en base al tamaño horizontal de esta.
133             int factor = round((factorh+factorw)/2)+1; // Calculo del factor final para el kernel de difuminado.
134             if(factor % 2 == 0)// Impide que el factor sea par, con el fin de evitar errores en la funcion GaussianBlur
135             {
136                 factor = factor+1;
137             }
138             GaussianBlur(trozoImagen, resultadoParcial, Size(factor,factor), 0,0,1); //Funcion maravillosa que realiza el difuminado de l
139         }
140         if(opcion=="2")
141         {
142             for(int x=0; x<trozoImagen.cols; x++) // Bucle de doble for que se encarga de obtener el escalado de grises para la imagen, e
143             {
144                 for(int y=0; y<trozoImagen.rows; y++)
145                 {
146                     float valores_promedio = 0;
147                     valores_promedio = (trozoImagen.at<Vec4b>(y,x)[0] + trozoImagen.at<Vec4b>(y,x)[1] + trozoImagen.at<Vec4b>(y,x)[2])/3;
148                     resultadoParcial.at<Vec4b>(y,x)[0] = valores_promedio;//Se iguala al promedio, que equivale a la tonalidad en grises
149                     resultadoParcial.at<Vec4b>(y,x)[1] = valores_promedio;//Se iguala al promedio, que equivale a la tonalidad en grises
150                     resultadoParcial.at<Vec4b>(y,x)[2] = valores_promedio;//Se iguala al promedio, que equivale a la tonalidad en grises
151                 }
152             }
153         }
154         if(opcion=="3")
155         {
156             resize(trozoImagen,resultadoParcial,cv::Size(0,0),2,2, INTER_NEAREST);// Re-escalado de la imagen o trozo al doble del tamaño
157         }
```

En la imagen anterior se muestra la parte del programa principal donde se realiza la operación sobre el trozo de imagen a trabajar que tiene cada procesador, en la opción uno que corresponde al difuminado de la imagen se puede ver la utilización de la función GaussianBlur, la cual hace el difuminado en base al método de gauss. La opción dos corresponde al escalado de grises la cual se realiza por cada pixel, en base al promedio de los colores de cada canal. La opción tres realiza el reescalado de la imagen utilizando la función resize, utilizando el método inter_nearest.

```
160         if(opcion == "1" || opcion == "2" || opcion == "3")
161         {
162             if(rank == 0)// Si se trata del maestro, realiza las operaciones de union de los t
163             {
164                 Mat imagenFinal(imagen.rows, imagen.cols, CV_8UC4);
165                 imagenFinal = resultadoParcial.clone();//Debido a que siempre sera el maestro
166 
167                 for(int p = 1; p < procesadores; p++)//Luego de lo anterior, recolecta el trab
168                 {
169                     Mat imagenTemporal(resultadoParcial.rows, resultadoParcial.cols, CV_8UC4);
170                     recibirImagen(imagenTemporal, p);
171                     hconcat(imagenFinal, imagenTemporal, imagenFinal);//Operacion clave, la en
172                 }
173                 guardarImagenResultado(imagenFinal, opcion);//Guarda la imagen procesada y com
174             }
175             else
176             {
177                 enviarImagen(resultadoParcial, 0);// Si nos e trata del maestro, envia la imag
178             }
179         }
```

Finalmente, en la imagen anterior, se puede ver el proceso de armado de la imagen resultante, el cual se divide en dos tipos de tareas principales definidas según el procesador, en el caso del maestro, este es el encargado del ensamblaje de toda la imagen, y para el caso de un esclavo, este debe simplemente enviar al maestro su imagen trabajada. Dentro de las tareas que debe hacer el maestro, en primera instancia se encuentra el generar la imagen final con el trozo que él mismo trabajó de la imagen original, el cual pega en la variable Mat imagenFinal en la línea de código 165, luego, debe hacer lo mismo con las imágenes de los esclavos, proceso que se lleva a cabo en las líneas 167 a 172, principalmente en la función hconcat, la cual une las imágenes hacia la derecha para formar finalmente la imagen total de

resultado. Finalmente, en la línea de código 173, guarda la imagen completada usando la función correspondiente.

Conclusión

La utilización de librerías como MPI permiten el desarrollo de aplicaciones que buscan una mejora en el rendimiento. Esto es posible debido a poseen una gran cantidad de componentes, los cuales son fáciles de implementar y utilizar. Destaca principalmente su utilidad y potencialidad del procesamiento paralelo en un cluster en lo que al trato de imágenes se refiere.

Desde el punto de vista de los procesos realizados, el difuminado de imagen resulta altamente complejo vista desde una perspectiva matemática, sin embargo el proceso se ve enormemente facilitado gracias a la utilización de la librería OPENCV. Esta librería posee un método definido llamado "GaussianBlur()", que permite realizar el proceso de una manera más sencilla.

Para la segunda operación se utilizó, en un primer momento, un método perteneciente a la librería OPENCV, el cual posee varias limitantes para trabajar ya que generaba problemas con los formatos de las imágenes, debido a esto se decidió llevar a cabo un método propio, que utiliza el promedio de los colores de cada canal para obtener el tono grises sin la necesidad de cambiar el formato de la imagen.

Finalmente para la tercera operación se utilizó una vez más la librería OPENCV con el método "resize()", el cual permite el escalado de la imagen a través de distintos parámetros de entrada.

Es bastante complejo tratar con imágenes manualmente, entendiéndose por manual, al conjunto de operaciones que se debe realizar por cada pixel y canal de colores trabajando con matrices de vectores que describen la información de toda la imagen, es por esto que se optó por métodos pertenecientes a la librería opencv para realizar la mayoría de las operaciones requeridas, agregando también, el trabajo paralelizado de mpi, siendo éste necesario para procesar con mayor rapidez la enormidad de cálculos que se deben realizar en este tipo de trabajos. En definitiva, la computación paralela es una opción obligatoria para procesos que requieran una cantidad de cómputo elevado si se quiere realizar dicho proceso en un tiempo decente de ejecución. Si bien para el caso actual, si no se paraleliza la operación sigue demorando muy poco, se debe recordar que se está trabajando con una sola imagen, pero, ¿Que pasaría en el caso de pasar una película completa en 4k a escala de grises? ¿O si se quiere re-escalar dicha película a otro tamaño de resolución?, es aquí donde toma fuerza la paralelización de procesos y donde se puede comprender parte de su gran utilidad.