ITESM Campus Puebla

Implementación de redes amplia y servicios distribuidos
(Grupo 201)

Procesamiento distribuido

Ituriel Mejía Garita - A01730875

Alejandro Castro Reus - A01731065

José Antonio Bobadilla García - A01734433

26 de abril de 2023

Descripción del experimento

El propósito de este informe es documentar el experimento realizado sobre el procesamiento distribuido con adición de una interfaz gráfica desarrollada en Python. El objetivo principal del experimento fue evaluar el desempeño de un conjunto de computadoras trabajando juntas en una tarea compartida utilizando una interfaz gráfica para poder escoger la imagen a procesar.
Para lograr esto, se desarrolló la interfaz usando el lenguaje de programación Python utilizando la librería de PYUIC.

La tarea que se realizó con el procesamiento distribuido, fue dar un efecto de desenfoque con 40 máscaras distintas, es decir, a 40 grados diferentes de desenfoque. Para probar la eficiencia del procesamiento distribuido, se le asignó cada máscara a un hilo de procesamiento diferente. El código para realizarlo se escribió en C y se usó MPI para distribuir las tareas entre los distintos hilos de procesamiento. Con esta metodología, se analizará si se logró una asignación eficiente de tareas y verificar que se haya logrado reducir el tiempo de procesamiento significativamente.

Condiciones de operación del experimento

El programa recibió como entrada imágenes BMP de cualquier dimensión, siempre y cuando cumplieran con los siguientes requisitos: formato BMP, profundidad de bits de 24 y sin compresión. La salida del programa generó una nueva imagen BMP con efecto de desenfoque.

Para llevar a cabo la ejecución del programa, se utilizaron los recursos de tres computadoras conectadas en red.

Computadora 1 (Master)

- Sistema operativo: Windows 10
- Memoria RAM: 16GB
- Procesador: Intel Core i5-10700K 3.80GHz
- Máquina Virtual:
    - Sistema Operativo: Linux Mint 21.1
    - Memoria Base: 4096 MB
    - Procesadores: 4 CPUs

Computadora 2 (Slave)

- Sistema operativo: Windows 10
- Memoria RAM: 8GB
- Procesador: Intel(R) Core(TM) i5-8265U CPU 1.60GHz

- Máquina Virtual:
    - Sistema Operativo: Linux Mint 21.1
    - Memoria Base: 4096 MB
    - Procesadores: 3 CPUs

Computadora 3 (Slave)

- Sistema operativo: Windows 10
- Memoria RAM: 8GB
- Procesador: Intel(R) Core(TM) i7-9750H CPU 2.60GHz
- Máquina Virtual:
    - Sistema Operativo: Linux Mint 21.1
    - Memoria Base: 4212 MB
    - Procesadores: 4 CPUs

Para el multiprocesamiento entre las tres computadoras, se emplearon las siguientes librerías: nfs-kernel-server 2.6.1, openssh-server 8.9, gcc 11.3.0, mpiexec 4.1.2, mpich 4.1.2, pyuic.

Además, se configuraron las IPs estáticas IPV4 de cada computadora: Ituriel con 192.168.1.5, Alejandro con 192.168.1.7 y Antonio con 192.168.1.6. Las tres computadoras se conectaron a la misma red LAN a través de un switch para realizar la conexión entre ellas.

Código en Python

Para instalar la libreria pyuic se utilizó el siguiente comando:

```
pip install pyuic5-tool
```

También se utilizaron las librerías de openCV las cuales se instalan con el siguiente comando:

```
pip install opencv-python
```

Al momento de compilar la aplicación en el usuario mpiu se necesitó realizar configuraciones con dicho usuario para poder utilizar el display de la computadora y poder mostrar la ventana con la interfaz gráfica.

Primeramente se utilizó el *comando xhost* + el cuál notifica al servidor X de que cualquier aplicación que se esté ejecutando en hostremoto tiene permiso para realizar peticiones a la misma.

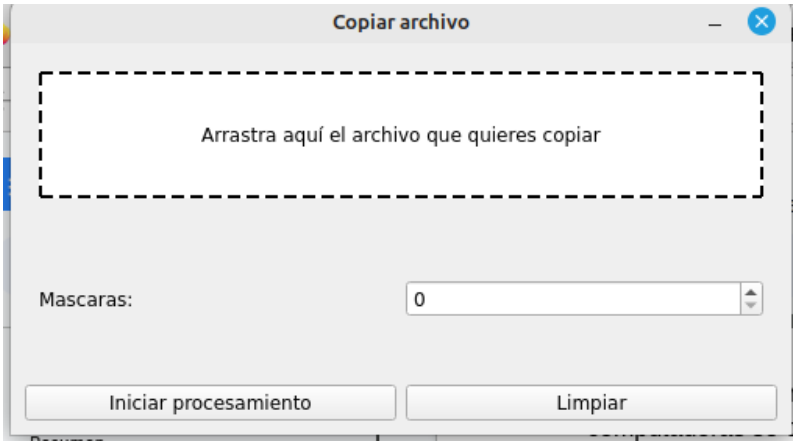
Posteriormente una vez dentro del usuario MPIU se setéo la variable DISPLAY con valor de :0, el cuál hace referencia al valor del Display de la computadora.

Una vez que se realizó esto se pudo compilar la interfaz en usuario mpiu y pode realizar el procesamiento distribuido

Para ejecutar el siguiente script de Python se utiliza el siguiente comando:

```
python3 script.py
```

Este comando mostrará la interfaz gráfica desarrollada con PyQt5:



En esta interfaz se puede arrastrar una imagen y definir la cantidad de máscaras para el programa. Esta interfaz copiará la imagen al directorio /mirror para poder realizar el procesamiento de dicha imagen.

```python
import sys
import os
import shutil
from PyQt5 import QtWidgets, QtCore
import subprocess


# PING

#Lista de direcciones IP de los hosts a los que se les va a hacer ping
HOSTS = ["192.168.1.6", "192.168.1.7", "192.168.1.5"]

class ExecuteBlurring:
    def __init__(self) -> None:
        pass

    #Obtener hosts disponibles
    def getHosts(self):
```

```
        # #Testing
        # hosts = [1,1,1]
        # num_hosts_available = sum(1 for value in hosts if value)
        # return num_hosts_available, hosts

        #Inicializar una lista que indique si los hosts están disponibles o no
        available_hosts = [0,0,0]

        #Iterar por cada host y hacer ping
        for i in range(len(HOSTS)):
            host = HOSTS[i]
            ping_output = subprocess.run(["ping", "-c", "1", "-W", "1", "-i",
"0.2", host], capture_output=True)
            if ping_output.returncode == 0: #Si el resultado del ping es 0,
significa que el host está disponible
                available_hosts[i] = 1
            else: #Si el resultado del ping es diferente de 0, significa que
el host no está disponible
                available_hosts[i] = 0

        #Contar cuántos hosts están disponibles
        num_hosts_available = sum(1 for value in available_hosts if value)
        return num_hosts_available, available_hosts

    #Obtener la distribucion de los procesos
    def getWeights(self,nMasks):
        n, hostsAvailable = self.getHosts()
        nMasks = int(nMasks)
        n = int(n)
        nMasksItu = (nMasks // n) * hostsAvailable[2] #Dividir las máscaras
entre los hosts disponibles y asignarlas proporcionalmente
        nMasksReus = (nMasks // n) * hostsAvailable[1]
        nMasksBoba = nMasks - nMasksItu - nMasksReus #Las máscaras que sobren
se asignan al tercer host disponible
        return [nMasksBoba, nMasksReus, nMasksItu]

    #Actualizar el writefile, archivo que contendra la distribucion de
procesos
    def writefile(self, nMasks):
        masksAssignment = self.getWeights(nMasks)
        with open("machinefile", "w") as file:
            for i in range(len(masksAssignment)):
```

```
                slots = masksAssignment[i] # Slots asignados
                if slots != 0:
                    # Escribir en el archivo un nuevo host con el formato: ub0
slots=14 max_slots=14
                    file.write(f'ub{i} slots={slots} max_slots={slots}\n')
                    print(f'ub{i} slots={slots} max_slots={slots}')

    # Comando para ejecutar con el writefile actualizado
    def execute(self,nMasks):
        self.writefile(nMasks) #Actualizar writefile



# GUI
class DragDropWidget(QtWidgets.QWidget):
    def __init__(self, parent=None):
        super().__init__(parent)
        # Se establece que el widget puede aceptar elementos que se arrastren
y suelten.
        self.setAcceptDrops(True)
        # Se define el estilo del widget.
        self.setStyleSheet("background-color: white; border: 2px dashed
black;")
        # Se agrega un QLabel que indica al usuario que arrastre y suelte el
archivo en el widget.
        self.label = QtWidgets.QLabel("Arrastra aquí el archivo que quieres
copiar", self)
        self.label.setAlignment(QtCore.Qt.AlignCenter)
        # Se agrega un QVBoxLayout al widget para ubicar el QLabel en el
centro del widget.
        self.layout = QtWidgets.QVBoxLayout()
        self.layout.addWidget(self.label)
        self.setLayout(self.layout)

    # Este método se llama cuando un elemento se arrastra y entra en el
widget.
    # Se verifica si se está arrastrando una URL (archivo) y se acepta la
acción propuesta.
    def dragEnterEvent(self, event):
        if event.mimeData().hasUrls():
            event.acceptProposedAction()
```

```
    # Este método se llama cuando un elemento se suelta dentro del widget.
    # Se obtiene la ruta del archivo y se actualiza el texto del QLabel con la
ruta.
    def dropEvent(self, event):
        file_path = event.mimeData().urls()[0].toLocalFile()
        self.label.setText(file_path)
        self.file_path = file_path

# Se define la clase MainWindow que hereda de QMainWindow.
class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()
        # Se establece el título y el tamaño fijo de la ventana principal.
        self.setWindowTitle("Copiar archivo")
        self.setFixedSize(500, 250)
        # Se define un QWidget que será el widget central de la ventana
principal.
        central_widget = QtWidgets.QWidget(self)
        self.setCentralWidget(central_widget)
        # Se agrega un QVBoxLayout al widget central para ubicar los elementos
en orden vertical.
        main_layout = QtWidgets.QVBoxLayout()
        central_widget.setLayout(main_layout)
        # Se agrega el widget DragDropWidget al QVBoxLayout para que el
usuario pueda seleccionar el archivo a copiar.
        self.drag_drop_widget = DragDropWidget(self)
        main_layout.addWidget(self.drag_drop_widget)

        # Se agrega un QSpinBox al QVBoxLayout para que el usuario pueda
ingresar el número de máscaras a aplicar al archivo.
        self.input_widget = QtWidgets.QWidget()
        input_layout = QtWidgets.QHBoxLayout()
        self.input_widget.setLayout(input_layout)
        self.input_label = QtWidgets.QLabel("Mascaras: ")
        input_layout.addWidget(self.input_label)
        self.input_number = QtWidgets.QSpinBox()
        input_layout.addWidget(self.input_number)
        main_layout.addWidget(self.input_widget)

        # Se agrega un QHBoxLayout al QVBoxLayout para ubicar los botones de
copiar y limpiar.
        button_layout = QtWidgets.QHBoxLayout()
```

```
        main_layout.addLayout(button_layout)
        # Se define un QPushButton para iniciar el proceso de copiado y se
conecta a un método de la clase.
        copy_button = QtWidgets.QPushButton("Iniciar procesamiento")
        copy_button.clicked.connect(self.copy_file)
        button_layout.addWidget(copy_button)
        clear_button = QtWidgets.QPushButton("Limpiar")
        clear_button.clicked.connect(self.clear)
        button_layout.addWidget(clear_button)

def copy_file(self):
    # Verifica que exista el widget para arrastrar y soltar archivos
    if hasattr(self, "drag_drop_widget"):
        # Obtiene la ruta del archivo arrastrado y soltado
        file_path = self.drag_drop_widget.file_path
        # Establece la ruta de destino como la carpeta del archivo actual y el
nombre del archivo arrastrado
        destination_path =
os.path.join(os.path.dirname(os.path.abspath(__file__)),
os.path.basename(file_path))
        # Verifica que haya un archivo seleccionado
        if file_path:
            try:
                # Copia el archivo a la ruta de destino con el comando de
shell 'cp'
                command = "sudo cp "+file_path+" "+destination_path
                os.system(command)
                # Muestra una ventana de diálogo indicando que se copió
correctamente el archivo
                QtWidgets.QMessageBox.information(
                    self,
                    "Copiado",
                    f"Se ha copiado correctamente el archivo {file_path} en la
ruta {destination_path}."
                )
                # Obtiene el número de máscaras seleccionado y lo imprime en
la consola
                print("Mascaras seleccionadas: ",self.input_number.text())
                # Crea un objeto de la clase ExecuteBlurring y ejecuta el
método execute con el número de máscaras como argumento
                exec = ExecuteBlurring()
                exec.execute(self.input_number.text())
```

```
            # Si ocurre una excepción durante la copia del archivo, muestra
una ventana de diálogo indicando el error
            except Exception as e:
                QtWidgets.QMessageBox.critical(
                    self,
                    "Error",
                    f"Error al copiar el archivo. {str(e)}"
                )
        # Si no se ha seleccionado un archivo, muestra una ventana de diálogo
de advertencia
        else:
            QtWidgets.QMessageBox.warning(
                self,
                "Atención",
                "Selecciona un archivo para copiar."
            )

def clear(self):
    # Si existe el widget para arrastrar y soltar archivos, establece el texto
de la etiqueta a su valor inicial y elimina la ruta del archivo seleccionado
    if hasattr(self, "drag_drop_widget"):
        self.drag_drop_widget.label.setText("Arrastra aquí el archivo que
quieres copiar")
        self.drag_drop_widget.file_path = ""
    # Si existe el widget para editar la ruta de destino, establece su texto a
una cadena vacía
    if hasattr(self, "path_edit"):
        self.path_edit.setText("")


if __name__ == "__main__":
    # Crea una aplicación de PyQt5
    app = QtWidgets.QApplication(sys.argv)
    # Crea una ventana principal de la aplicación
    main_window = MainWindow()
    # Muestra la ventana principal
    main_window.show()
    # Inicia la aplicación y espera a que el usuario cierre la ventana
    sys.exit(app.exec_())
```

Código en C

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>


typedef struct {
   unsigned char red, green, blue;
} pixel;


int main(int argc, char *argv[]) {
   int rank, size;
   char processor[MPI_MAX_PROCESSOR_NAME];
   int proc_name_len;

   // Inicializar configuracion de MPI para comenzar proceso en una maquina
distinta
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Get_processor_name(processor, &proc_name_len);



   // Establecer valores iniciales y archivo que se va a leer
   FILE *input, *output, *readings;
   input = fopen("original_image.bmp","rb");
   float red_sum = 0, green_sum = 0, blue_sum = 0;


   int y, x, i, j;
   // Leer encabezado BMP
   unsigned char header[54];
   fread(header, sizeof(unsigned char), 54, input);


   // Extraer dimensiones de la imagen
   int width = *(int*)&header[18];
   int height = *(int*)&header[22];
```

```
// Calcular relleno de la imagen
int padding = 0;
while ((width * 3 + padding) % 4 != 0) {
    padding++;
}


// Asignar memoria para los datos de la imagen
pixel *img_data = (pixel*)malloc(width *  height * sizeof(pixel));
// Leer los datos de la imagen
fread(img_data, sizeof(pixel), width * height, input);
int num_output_files = 2;


// Aplicar el desenfoque a la imagen en paralelo
if (rank < size) {
    int num_procs = size - rank;
    char filename[50];
    sprintf(filename, "blurred_%d.bmp", num_procs);
    output = fopen(filename, "wb");


    // Calcular el tamaño del kernel y su sumatoria
    int kernel_size = 11 + num_procs*2;
    float kernel[kernel_size][kernel_size];
    float kernel_sum = 0;
    for (int i = 0; i < kernel_size; i++) {
        for (int j = 0; j < kernel_size; j++) {
            kernel[i][j] = 1.0 / (float)(kernel_size * kernel_size);
            kernel_sum += kernel[i][j];
        }
    }


    // Escribir el encabezado BMP
    for (int i = 0; i < 54; i++) {
        fputc(header[i], output);
    }

    // Con los primeros dos loops, se recorre toda la imagen
    // No se contemplan valores que estan en los extremos
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
```

```
                total_r = 0, total_g = 0, total_b = 0;
                // Con los otros dos loops, se recorre la mascara
                // Se multiplicaran todos los valores de la mascara,
                // por todos los pixeles que rodean el pixel que estamos analizando
                for (int i = 0; i < kernel_size; i++) {
                    for (int j = 0; j < kernel_size; j++) {
                        int nx = x - (kernel_size/2) + j;
                        int ny = y - (kernel_size/2) + i;
                        if (nx >= 0 && nx < width && ny >= 0 && ny < height) {
                            //Multiplicacion de los pixeles con el kernel
                            int index = ny * width + nx;
                            float kernel_value = kernel[i][j] / kernel_sum;
                            total_r += kernel_value * image_data[index].r;
                            total_g += kernel_value * image_data[index].g;
                            total_b += kernel_value * image_data[index].b;
                        }
                    }
                }
                // Al final, en la copia de la imagen (el resultado), se cambiara el
pixel original, por la suma calculada
                pixel blurred_pixel = {
                    (unsigned char)total_b,
                    (unsigned char)total_g,
                    (unsigned char)total_r
                };
                fputc(blurred_pixel.b, output_file);
                fputc(blurred_pixel.g, output_file);
                fputc(blurred_pixel.r, output_file);



            }
            for (int i = 0; i < padding; i++) {
                fputc(0, output_file);
            }

            //Se imprime el progreso que lleva cada proceso
            if ((y + 1) % 100 == 0) {
                printf("Processed %d rows from processor %d with a total of %d
processors and image=%d from pc %s\n", y + 1, myrank, nprocs, n, processor_name);
            }
        }
        fclose(output_file);
    }
        // Clean up
        free(image_data);
```

```
        fclose(input_file);
        MPI_Finalize();
    return 0;
}
```

Resultados

Los resultados del experimento indicaron que el procesamiento distribuido con la adición de una interfaz gráfica desarrollada en Python fue efectivo en la tarea de aplicar el efecto de desenfoque a 40 imágenes diferentes utilizando 40 máscaras distintas.

La interfaz gráfica desarrollada nos permitió seleccionar fácilmente la imagen que se debe procesar, lo que facilitó la tarea de administrar las tareas de procesamiento. El uso de MPI para distribuir las tareas entre los distintos hilos de procesamiento permitió una asignación eficiente de tareas, lo que resultó en una reducción significativa del tiempo de procesamiento en comparación con el procesamiento en una sola máquina. Esto demuestra que el procesamiento distribuido y la interfaz gráfica desarrollada en Python fueron eficaces para mejorar el rendimiento en la tarea de aplicar el efecto de desenfoque a un conjunto de imágenes utilizando múltiples máscaras.

Conclusiones

En conclusión, el uso de un procesamiento distribuido con una interfaz gráfica desarrollada en Python es una estrategia efectiva para mejorar significativamente el rendimiento en aplicaciones de procesamiento de imágenes que requieren tareas intensivas en cómputo y procesamiento y la experiencia de usuario.

La implementación de una estrategia de procesamiento distribuido con una interfaz gráfica desarrollada en Python puede ser una herramienta muy valiosa para la investigación y el desarrollo de aplicaciones de procesamiento de imágenes. Esto puede llevar a una mayor eficiencia en la gestión y procesamiento de datos, lo que a su vez puede reducir significativamente los tiempos de procesamiento en comparación con los métodos tradicionales.

En resumen, la combinación de procesamiento distribuido y una interfaz gráfica desarrollada en Python puede ser una estrategia muy efectiva para mejorar el rendimiento y la eficiencia en aplicaciones de procesamiento de imágenes, lo que puede llevar a resultados más rápidos y precisos en la investigación y el desarrollo de aplicaciones de procesamiento de imágenes.