Parallel Image Processing

Course Level:
CS1

PDC Concepts Covered:

PDC Concept	Bloom Level
Concurrency	C
Data parallel	A

Programming Skill Covered:
- Loading images into arrays
- Manipulating images

Programming Knowledge Prerequisites:
Basic programming knowledge in C is required for this lab. More specifically, the below skills are sufficient to complete the coding assignment.
- Arrays
- Structures
- Arrays of structures
- Pointers
- ifstream/ofstream

Tools Required:
- A C++ compiler that is OpenMP capable (such as the gnu C++ compiler).
- The following image files: ttu.ppm ttu_tile.ppm
- A program that can display PPM image files (for example, a browser).

Prolog and Review

Unlike scalar variable, such as variables of type int, double, and char, array variable can hold more than one value. Consider the following variable declarations.

```
1 int i = 5;
2 int numbers[3] = {10, 20, 30};
```

The variable declaration on line 1 of the above code declares a variable called ***i*** that holds the number 5, and that variable can only hold one number. However, the variable declared at line 2 can hold 3 integers, and is initialized to hold the integers 10, 20, and 30. Accessing the integers is as simple as indicating the correct index. In C++, arrays are zero based, so the first item in the array always starts as index 0. So, in the above example, numbers [0] holds the integer 10, numbers [1] holds the integer 20, and number[2] holds the integer 30.

You are no limited to storing scalar values, such as integers, into arrays. You can also declare arrays that hold structures. Recall that a structure is a data type that can hold a set of specified variables. For example, if you want to create a new data type that holds a person's name, age, and salary, you could declare that data type as follows.

```cpp
struct person_t {
  char name[30];
  int age;
  float salary;
};
```

Then, creating a variable of that type is as simple as the following.

```cpp
person_t joe = {"Joe Schmoe", 22, 20345};
std::cout << joe.name << "'s age is " << joe.age << std::endl;
```

Notice that after the variable joe is assigned a name, age, and salary, the above code accesses joe's name and age using dot notation. Creating and accessing arrays of structures simply combines array syntax with structure syntax. Consider the following.

```cpp
1 person_t people[2] = { {"Joe Schmoe", 22, 20345},
2                        {"Jane Doe",   23, 45323} };
3 std::cout << people[0].name << "'s age is " << people[0].age << std::end;
4 std::cout << people[1].name << "'s age is " << people[1].age << std::endl;
```

As you can see, the above code declares an array that can hold two structures of type person_t. The array is initialized with the name, age, and salary of each of the two people. Then, at line 3, the code prints the name and age of the person_t at index 0 of the array. Line 4 then prints the name and age of the person_t at index 1 of the array.

Problem Description

This lab you will be writing program to manipulate images. You will write a serial version of the program and time it. Then you will write a parallel version of the program, time it, and then compare the speed of the two programs.

The two image processing techniques that you will implement are grey-scaling and flipping an image. Below is an example of image that has been gray-scaled, flipped, and then both gray-scaled and flipped.

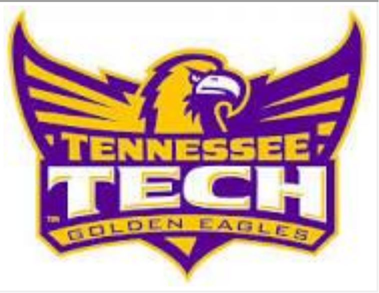
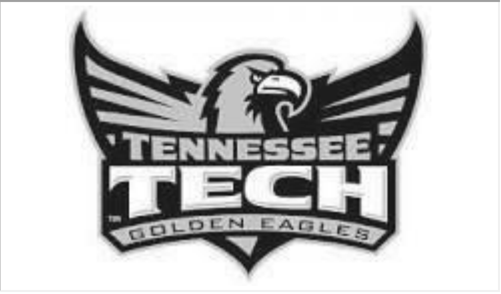
	

Figure 4a: Color image	Figure 4b: Gray-scaled image

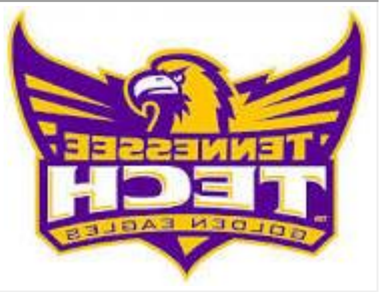
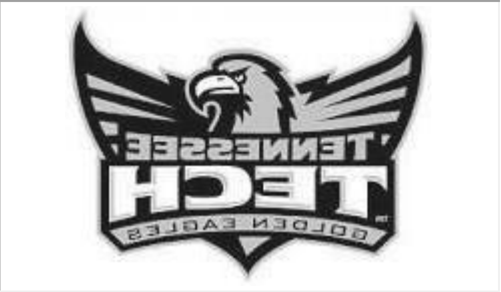

	

Figure 4c: Flipped image	Figure 4d: Gray-scaled and flipped image

Images are represented as pixels. You can think of a pixel and an individual color "dot" on your monitor screen. The color of the pixel is represented as a mixture of intensities of the colors red, green, and blue. Each intensity is represented as an 8-bit number in the ranging from 0 to 255. For example, the values (0,0,0) represents the color black, the values (255,0,0) represent the color red, and the values (255,255,0) represent the color yellow. We call these intensities RGB values (for red, green, and blue).

Gray scaling an image represented as a series of RGB values is easy. Different methods exist, but an effective method is called the luminosity method. Given the *i*th pixel, you gray-scale that pixel with the following formula:

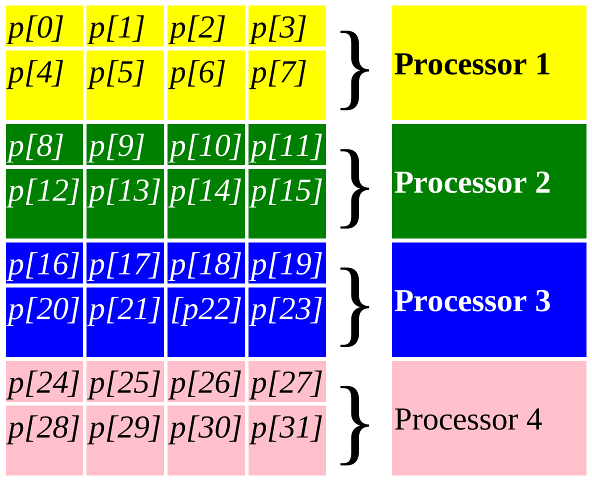
*gray_value[i]= 0.21 * pixel[i].red + 0.72 * pixel[i].green + 0.07 * pixel[i].blue*

Then, for each *i*, set the red, green, and blue component of *pixel[i]* to *gray_value[i]* to gray-scale the image.

Flipping the image is also easy. Flip an image by flipping the 1st pixel with the last pixel, the 2nd pixel with the next-to-last pixel, the 3rd pixel with the second-to-last pixel, and so on.

Methodology

Given the overview of the gray-scale and flipping algorithms above, how do you gray-scale and flip an image in parallel? Consider the following as a representation of an image:



An image has both a height and a width. The array of pixels shown above can be considered as occurring in rows, where each row is a line of pixels that would appear across the screen. The size of each line of pixels is equal to the image's width, and the number of lines is equal to the images height.

When writing a parallel application, you first must determine how to divide the problem among the available processors. Dividing the problem requires determining 1) how much of the problem each processor should compute, and 2) determining where, in the input data, the processor should begin and end its computations. In general, when dividing the rows among processors, you should divide the work equally. So, if the image consists of ***n*** rows, and there are ***p*** processors available, then each processor should get roughly ***n/p*** rows.

A natural division for an image is to divide the image into chunks where each chunk consists of number of rows of pixels. Then you assign each chunk of rows to a processor. So, in the example above, if you have four processors, you would give each processor two rows. Processor 1 would work on the 1st two rows (those rows in yellow) starting at index 0 and finishing with index 7, processor 2 would work on the third and fourth rows (those rows colored green) starting at index 8 and finishing at index 15, and so on.

***Note to the instructor:** The instructor can illustrate **Data parallel** by pointing to that different processors are doing the same computation but on different pieces of data. Thus, different processors work simultaneously, which is an example of **Concurrency**.

Implementation

For this lab, you will be implementing functions to read, write and grayscale image files. You will then modify your code to that it processes the image files in parallel, thus speeding up the grayscale function.

Reading and Writing a PPM File

You will be reading and writing one of the simplest graphic file formats. The format is the Netpbm color image format, also known as the PPM format.

Reading the header

A PPM file consists of a header describing the file type, the image width, the image height, and the maximum color value per RGB component. The header is formatted as in the following table:

Description	**Type**	**Size**	**Value**
Magic Number that identifies the file format	Characters	2	"P6"
Whitespace	Characters	Variable	space, tab, CR, or LF
Width	Characters	Variable	Numbers in character form
Whitespace	Characters	Variable	space, tab, CR, or LF
Height	Characters	Variable	Numbers in character form
Whitespace	Characters	Variable	space, tab, CR, or LF
Max color value	Characters	Variable	Numbers in character form
A single Whitespace	Characters	Variable	space, tab, CR, or LF (usually LF)

Write a function that can read the header of a PPM image. The prototype for the function should be as follows:

```
void PPM_read_header(std::ifstream &inp, PPM_header &ppm_header);
```

The PPM_structure should be as follows:

```
struct PPM_header {
    int width;
    int height;
    int max_color;
};
```

Note that the file should be opened for reading and passed to PPM_read_header() as a ifstream object. To make reading the header simple, use the input stream operator (operator>>) to read the Magic Number, Width, Height, and Max color value. Make sure that you check the Magic Number and throw a std::runtime_error if the signature does not match. You should read the last single whitespace character of the header using ifstream::read() instead of the input stream operator, as the stream operator will skip over the whitespace and soak up the next character.

Once your have written the function, write a main() driver that calls the function on the ttu.ppm image and prints out the images width, height, and max color value. For example, consider the following invocation:

```
$ ./ppm_lab ttu.ppm
Width: 184, Height: 140, Max color: 255
```

Reading the image

Next, you will write a function that reads the image data. For the purposes of this lab, you will only be responsible for reading files that have a Max color value of 255. In other words, an image has one byte per RGB component. You should create a structure that represent a pixel. The structure should be as follows:

```
struct RGB_8 {
    uint8_t r;
    uint8_t g;
    uint8_t b;
} __attribute__((packed));
```

Note that the attribute added to the end of the structure is to ensure that the compiler does not pad the structure. In other words, the red, green, and blue values must be formatted exactly as they appear in the structure.

The prototype for your image reading function is as follows:

```
void PPM_read_rgb_8(std::ifstream &inp, int width, int height, RGB_8 *img);
```

This function should be called immediately after reading the header. Therefore, the file pointer of the ifstream object will be in the correct position to begin reading the image data. Note that the width and height parameters are obtained from the header that was read by the PPM_read_header() function.

This function is actually very easy to write. The body of the function should call the inp.read() function, passing the img array and the number of bytes to read. How do you know how many bytes to read? The number of bytes is the size of an RGB_8 times the width times the height of the image.

Next, modify your main() to read the image. Before you call your PPM_read_header() function, you will need to allocate an array to hold the image. To do so, declare a pointer to an RGB_8, and use the new operator to allocate it, like so:

```
RGB_8 *image = new  RGB_8[header.width*header.height];
```

Writing the image

Writing the image is as simple as reading the image. Create a function that has the following prototype:

```
void PPM_write_header_8(std::ofstream &outp, int width, int height);
```

Create the function to write the image header to a file. Note that the format must match the format described in the above *Reading the Header* section. All you will only need the output stream operator (operator>>) to write the header data.

Finally, you can write the function that saves the image data. The prototype for that function is as follows:

```
void PPM_write_rgb_8(std::ofstream &outp, int width, int height, RGB_8 *img);
```

Note that this function is as easy as writing the function that reads the image data. However, it should call outp.write(). You should be able to determine what parameters to pass to outp.write().

To test your functions, add code to your main that simple copies an image file, as in the following example run:

```
$./ppm_lab ttu.ppm ttu_copy.ppm
$ see ttu_copy.ppm                     # works on Linux to view
the file in an image viewer
```

Converting to Grayscale and flipping

Finally, you get to do something interesting with the image. You will write a function to convert the image into grayscale. Converting an image into grayscale is simple. You will use the luminosity method, which typically gives good results for most situations. The luminosity method, as mentioned in the *Methodology* section above, re-calculates the red, green, and blue values according to the following formula:

*grayval = 0.21 * red + 0.72 * green + 0.07 * blue*

The color called *grayval* is repeated as the red, green, and blue component for that pixel in the image. Therefore, the algorithm (not c code!) is as follows:

```
void to_grayscale(RGB_8 *img, int width, int height) {
    for each rgb color value in img (denote as img[i])
      set temp to 0.21 * img[i].r + 0.72 * img[i].g + 0.07 * img[i].b
      set img[i].r to temp
      set img[i].g to temp
      set img[i].b to temp
    end for
}
```

Show that your program works by modifiing your main so that it calls to_grayscale() before it calls PPM_write_header_8() and PPM_write_rgb_8(). Make sure that your code works by viewing the image (it should be, of course, grayscale).

Next, write a function that will flip the image as if it were being viewed in a mirror. The flip function has the following prototype:

```
void flip(RGB_8 *img, int width, int height);
```

The algorithm is pretty straighforward. You will swap the pixel in the ith position of the row with the pixel at the (width - i - 1) position in the row. So, you will swap the ith pixel in the row with the width-1 pixel, the ith-1 pixel with the width-2 pixel, and so on. The algorihtm should look like so:

```
void  flip( RGB_8 *img, int width, int height) {
    for each row in img
      for each rgb color value in row (denote at row[i])
        swap row[i] with row[width-i-1]
      end for
    end for
}
```

Show that your program works by modifiing your main so that it calls flip() after it calls to_grayscale(), but before it calls PPM_write_header_8() and PPM_write_rgb_8(). Make sure that your code works by viewing the image (it should be, of course, grayscale *and* flipped).

Grayscale and Flip in Parallel

Now for the coup de grace. Processing images can be expensive, especially for very large images or processing thousands, or even millions, of images. Fortunately, some image processing can be done in parallel. You have had a lecture about parallelism in your CSC 2100 course. Basically, parallel code can accomplish multiple computations at the same time. Running code is parallel can greatly speed up computations on machines that have more than one processor or more than one core.

Unfortunately, you cannot just run a program on a machine with multiple cores and expect it to be able to execute its code in parallel.   The program has to be restructure (programmed differently) to be able to use the multiple cores on the computer.

However, the OpenMP library has been written to make writing parallel code much easier. In fact, writing code to parallelize simple loops, such as the one in your to_grayscale() and flip() functions is trivial. All you have to do is add the following pragma immediately before the `for` loop in to_grayscale() and the `for` loop in flip().

```
#pragma omp parallel for
```

So, add the pragma, and compile it, adding the following option to your compiler's command line: -fopenmp

So, What's the Difference?

To see how your code with the added OpenMP pragma makes a difference, download the ttu_tiled.ppm file at the link provided above (its a large file). next add the following code before you call your grayscale() function:

```
// Note that  the following code works for g++ on Linux, Mac OS, and Windows
(using MinGW)
struct timeval tv1;
struct timeval tv2;
gettimeofday(&tv1, NULL);
```

Next, add the following code immediately after your grayscale function:

```
gettimeofday(&tv2, NULL);
std::cout << "Total time: " << (double) (tv2.tv_usec - tv1.tv_usec) / 1000000
+
  (double) (tv2.tv_sec - tv1.tv_sec) << std::endl;
```

Now, run your code twenty times on the ttu_tile.ppm image and calculate the average time. Next, comment the OpenMP pragma lines so that the code does not run in parallel. Re-run the program twenty times and record the average. Next, calculate the speedup based on the average times.

Post Lab Questions

1. What is the average time for twenty runs of the serial version of the code (i.e. with pragmas commented)?
2. What is the average time for twenty runs of the parallel version of the code?
3. Calculate the speedup of the parallel version. Is the parallel code significantly faster?
4. The *Methodology* section above described how you decompose the image processing routines to parallelize them. Obviously, OpenMP did all the work for you. How many rows do you think OpenMP assigned to each processor? Hint: have your code print the image's height, and also have you code print out the number of threads in the computation (the function `omp_get_thread_num()` returns the number of threads).

Turn In

Submit a file called README, as a text file, that has the post lab questions and the answers to those questions. Also include your source code for ppm_lab.cpp.