CSCI4110: High Performance Computing

Assignment 4: Implement Parallel Image Processing in C++

Skills you will learn

- Loading images into arrays
- Manipulating images
- Parallel Processing using OpenMP

Task 1: Problem Description

You will be writing a 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 an 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:

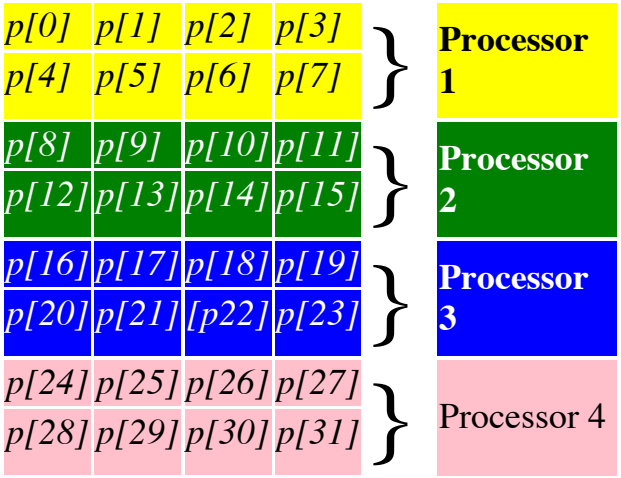
*gray_value[i]= 0.21 * pixel[i].red + 0.72 * pixel[i].green + 0.07 * pixel[i].blue*

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.

Implementation

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.

Tools

You will need to use the following tools to complete your assignment:

- An editor.
- A C++ compiler that is OpenMP capable (such as the gnu C++ compiler).
    - Note: shell.csc.tntech.edu has everything you need. You can use the vim for nano editor and the g++ compiler.
- Some image files such as ttu.ppm ttu_tile.ppm
- A program that can display PPM image files (for example, a browser).

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 you 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 postion 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 compies 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 that 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.

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).

Task2: Problem Description

Manipulating Images

In this lab, you will be writing programs to manipulate images. The two image processing techniques that you will implement are grey-scaling and flipping an image. Below is an example of an image that has been gray-scaled, flipped, and then both gray-scaled and flipped.

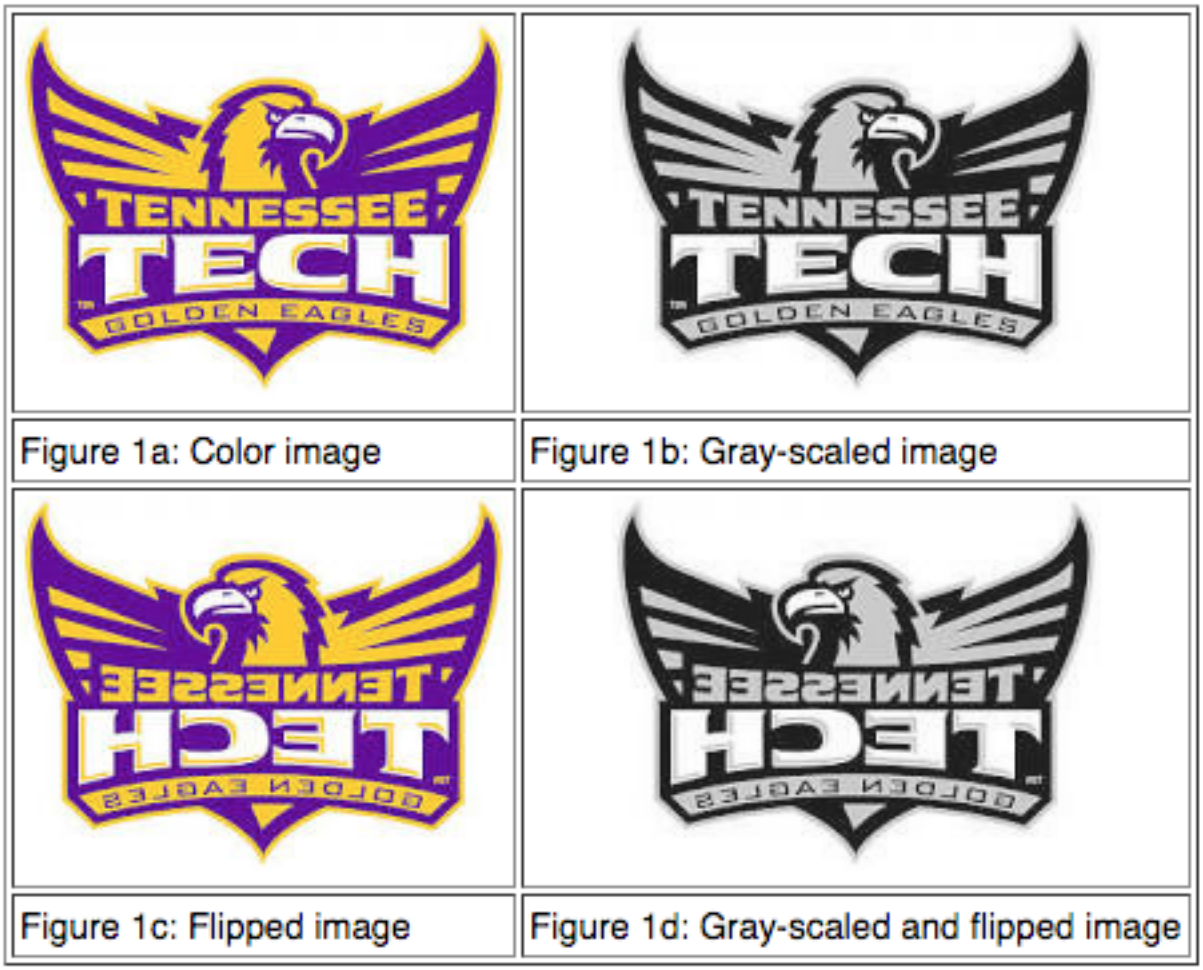


Figure 1a: Color image

Figure 1b: Gray-scaled image

Figure 1c: Flipped image

Figure 1d: 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

Pipelining and Parallelism

You will be using an image processing technique called *Filtering*. Filtering is simply applying a sequence of operations to an image to achieve a desired outcome. For example, if you wanted to show an image as if it were depicted as shown in a mirror on an old black-and-white TV, you would apply the gray-scale and flip filters. Fortunately, for many image filters, such as gray-scale and flipping, you can accomplish the filtering in parallel and thus reduce the computation time. One method is *pipelining*, which is a form of *functional decomposition*. Functional decomposition decomposes a problem according to the functions that are used to accomplish the computations. Functional decomposition is in contrast to *domain decomposition* that decomposes a problem according the to input data.

As shown in Figure 2, Pipelining is a form of functional decomposition in which a series of filters, or functions, manipulates a portion of the input data, and then passes the data onto the next function, which begins processing the data. However, the first function simultaneously begins computing of the next portion of input data, so that both functions are computing in parallel. Note that the technique can be applied to arbitrarily many functions.

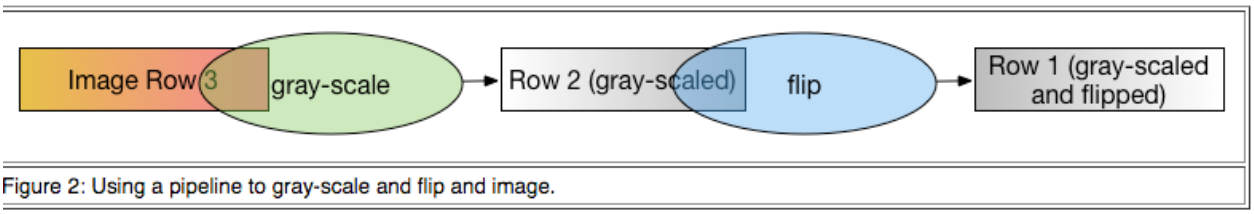


Figure 2: Using a pipeline to gray-scale and flip and image.

Notice that in Figure 2, the rectangles are overlapping with the ovals to represent that the gray-scale function is simultaneously processing image row 3 while flip is processing row 2, which was previously processed by gray-scale.

So, how do you implement a pipeline. Consider Figure 3. A straightforward way to implement pipelining is to use a queue to pass data between the functions. When the first function, which is gray-scale in this example, finishes a row, it passes that row to flip by putting the row in a shared queue. When the flip function is ready to process the next row, it checks the queue. If the queue has a row in it, then the flip function removes the row and processes it.

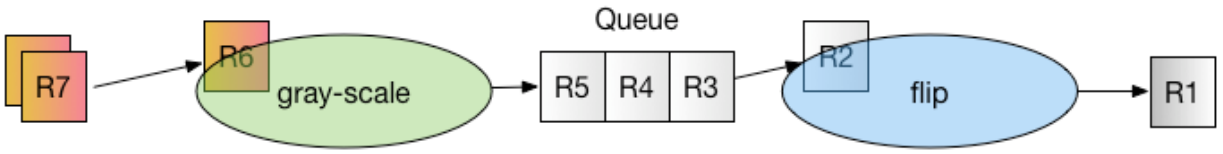


Figure 3: Implementing a pipeline with a queue.

Description

Introduction

For this lab, you will be implementing filters to gray-scale and flip an image. You will then modify your code so that it processes the filters in parallel via a pipeline, thus speeding up the applications of the functions.

Tools

You will need to use the following tools to complete your assignment:

- An editor.
- A C++ compiler that is OpenMP capable (such as the gnu C++ compiler).
  - Note: shell.csc.tntech.edu has everything you need. You can use the vim or nano editor and the g++ compiler.
- The following image files: ttu.ppm ttu_tile.ppm
- A program that can display PPM image files (for example, a browser).
- The following library that contains code to read and write PPM images: libppm.cpp libppm.h

Preliminaries: Reading and Writing a PPM File

To get started, download the libppm.cpp and libppm.h files above, as well as the ttu.ppm and ttu_tile.ppm image files. Next, you will make sure that you can compile and run a simple program that uses the functions in libppm.cpp to read and write PPM files. Create a new file in your editor called ppm_lab.cpp. Put the following code in ppm_lab.cpp:

```
#include <iostream>
#include <fstream>
#include <stdexcept>
#include <sstream>
#include <sys/time.h>

#include "libppm.h"


int main(int argc, char *argv[]) {

  if (argc != 3) {
    std::cerr << "Usage: " << argv[0] << " in_ppm_file out_ppm_file" <<
std::endl;
    return 1;
  }

  PPM_header img_header;

  try {

    std::ifstream ifs(argv[1], std::ios::binary);
    if (!ifs) {
      throw std::runtime_error("Cannot open input file");
    }
    PPM_read_header(ifs, img_header);

    std::cout << img_header << std::endl;

    RGB_8 *img = new RGB_8[img_header.height * img_header.width];
    PPM_read_rgb_8(ifs, img_header.width, img_header.height, (RGB_8 *) img);

    std::ofstream ofs(argv[2], std::ios::binary);
    if (!ofs) {
      throw std::runtime_error("Cannot open output file");
    }
    PPM_write_header_8(ofs, img_header.width, img_header.height);
    PPM_write_rgb_8(ofs, img_header.width, img_header.height, (RGB_8 *) img);

    ifs.close();
    ofs.close();
```

```
  } catch (std::runtime_error &re) {
    std::cout << re.what() << std::endl;
    return 2;
  }
  return 0;
}
```

ote that the above code simply reads the image file given as the first parameter on the command line, and then saves the image to the file given as the second parameter on the command line.

Grayscaling and Flipping

Now, you will write two functions. The first function is called grayscale. The prototype for the function is given below:

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

Converting an image into grayscale is simple. You will use the luminosity method, which typically gives good results for most situations. The luminosity method 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 is as follows:

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

Once you have this function written, test it. Modify 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
}
```

Test your new flip function by modifying your main so that is calls flip() after calling to_grayscale(). Make sure that your code works by viewing the image (it should be both grayscaled and flipped).

After you know your code works, time it. Download the ttu_tiled.ppm file at the link provided above (its a large file). Next add the following code immediately 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 you call the flip() 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.

Pipelining

Now, you are going to modify your code so the the image filters will be applied in a pipeline. Note that once a row of pixels is finished being grayscaled, the row can be immediately flipped. Time can be saved by flipping an already greyscaled row, and, at the same time, grayscaling the next row in the image. Unfortunately, you have to modify your code to take advantage of this parallelism. Fortunately, you have some help. OpenMP is a tool that simplifies taking advatage of parallism.

You will modify your code so that the grayscale() function enqueues the row that it justs finished grayscaling. Then, you will modify your flip routine by removing the outer for loop that loops through the rows and replacing it with a call to dequeue the next row to flip. Using a queue in this way implements a pipeline between the two functions.

So, now for the details. First, declare your queue at the top of your C++ file. The declaration should look like so:

```
std::queue<RGB_8 *> pipeline;
```

Add the following code to your C++ file. This code accomplishes two goals. First, the queue is a shared resource between the grayscale() function and the flip() function, so it must be protected from concurrent access (Note the OpenMP critical pragma). Second, the flip() function can only flip a row when a row is available. Thus the dequeue() function must check the availability of a row in a loop.

```
RGB_8 *dequeue(std::queue<RGB_8 *> &q) {
   RGB_8 *image_row;
  bool keep_checking = true;
  while (keep_checking) {
#pragma omp critical (pipeline)
    {
      if (!pipeline.empty()) {
        image_row = pipeline.front();
        pipeline.pop();
        keep_checking = false;
      }
    }
#pragma omp taskyield
  }
  return image_row;
}

void enqueue(std::queue<RGB_8 *> &q, RGB_8 *row) {
#pragma omp critical (pipeline)
```

```
  {
    pipeline.push(row);
  }
}
```

Next, modify your grayscale code such that, at the end of processing each row, that row is enqueued. So the algorithm changes to the following:

```
void to_grayscale(RGB_8 *img, int width, int height) {
     for each row in img
       for each rgb color value in row (denote as row[i])
         set temp to 0.21 * row[i].r + 0.72 * row[i].g +
0.07 * row[i].b
         set row[i].r to temp
         set row[i].g to temp
         set row[i].b to temp
       end for
       enqueue(row)
     end for
     enqueue(0)
}
```

Note the enqueue(0) at the very end. The sentinel value of 0 is used to signal the flip() function that no more rows are available.

Next, modify the flip funciton. Your flip function should not use an outer for loop to determine the next row. Instead, it should get the next row from the queue. Following is the pseudocode:

```
void  flip(int width) {
     do
       set row to dequeue(pipleline)
       if row is not 0
         for each rgb color value in row (denote at
row[i])
           swap row[i] with row[width-i-1]
         end for
       end if
     while row is not 0
}
```

Finally, modify your main so that OpenMP will spawn the functions in two different threads. So, replace your calls to grayscale() and flip() in the main with the following

code:

```
#pragma omp parallel num_threads(2)
  {
    if (omp_get_thread_num() == 0) {
      to_grayscale(image, width, height);
    } else {
      flip(width);
    }
  }
```

Note that the above code runs to_grayscale() in the thread with id (thread number) 0, and the flip() function in the other thread.

Compile your program. Add the "-fopenmp" option to your compiler command (for example: `g++ -fopenmp -o ppm_lab ppm_lab.cpp libppm.cpp`). Test it to make sure that it works.

Now, run your code twenty times on the ttu_tile.ppm image and calculate the average time.

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. In the Methodology section above, pipelining, which is a method of functional decomposition, is described. Describe how you would implement the solution using only domain decomposition.