Parallel Image Processing

Course Level:

CS2

PDC Concepts Covered:

PDC Concept	Bloom Level
Concurrency	C
Task parallel	A
Producer-Consumer	A
Synchronization	A

Programming Skill Covered:

- Loading images into arrays
- Manipulating images

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
- Pointer
- 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).
- The following library that contains code to read and write PPM images: libppm.cpp libppm.h

Prolog and Review

For this lab, you should review your CS1 material on arrays, structures, and arrays of structures.

You should also review your CS2 course material on queues. The following is a short review. The queue is a container. In other words, it is used to create items of data much like an array. However, it exports an interface that is different from the array for manipulating the contained items. If you want to add items to the queue, you must append the items to the end of

the queue. If you want to get the value of an item in the queue, you must get the value of the item at the beginning of the queue.

For example, consider the C++ Standard Template Library, or STL, queue. You must include the following line in your C++ code to use the STL queue.

```
#include <queue>
```

Then you can allocate a queue and used the queue's `push()` method to append an item to the end of the queue, as in the following example.

```
1
2  #include <iostream>
3  #include <queue>
4
5  int main() {
6    std::cout << "Appending the numbers 10 and 20 to the queue..." << std::endl;
7    std::queue<int> q;
8
9    q.push(10);
10   q.push(20);
11
12   return 0;
13 }
```

If you want to dequeue an item from the queue to get its value, you must use two methods of the STL queue: `front()` and `pop()`. The STL queue also includes methods for determining if a queue is empty, determining the queue's size, and other helpful methods. Following is a more complete example.

```
1  #include <iostream>
2  #include <queue>
3
4  int main () {
5    std::queue<int> q;
6    int number = 0;
7
8    std::cout << "Please enter some integers (enter 0 to end):\n";
9
10   std::cin >> number;
11   while (number != 0) {
```

```
12      q.push (number);
13      std::cin >> number;
14    }
15
16    std::cout << "The queue is " << (q.empty() ? "Empty" : "Not Empty") <<
17 std::endl;
18    std::cout << "The size of the queue is " << q.size() << std::endl;
19    std::cout << "The queue contains: ";
20    while (!q.empty()) {
21      std::cout << ' ' << q.front();
22      q.pop();
23    }
24    std::cout << std::endl;
25
26    return 0;
27 }
```

The following is what the code prints when it is executed.

```
$ ./queue
Please enter some integers
(enter 0 to end):
1 2 3 4 5 0
The queue is Not Empty
The size of the queue is 5
The queue contains:  1 2 3 4 5
```

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 to the 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.

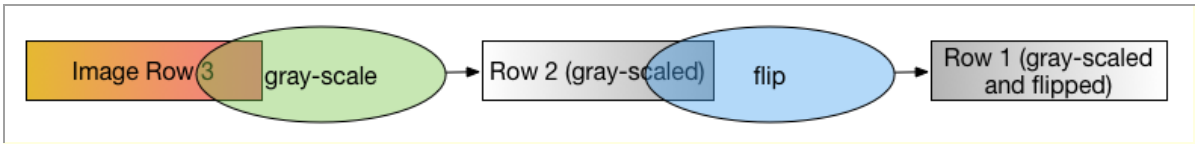

Figure 2: Using a pipeline to gray-scale and flip and image.

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.

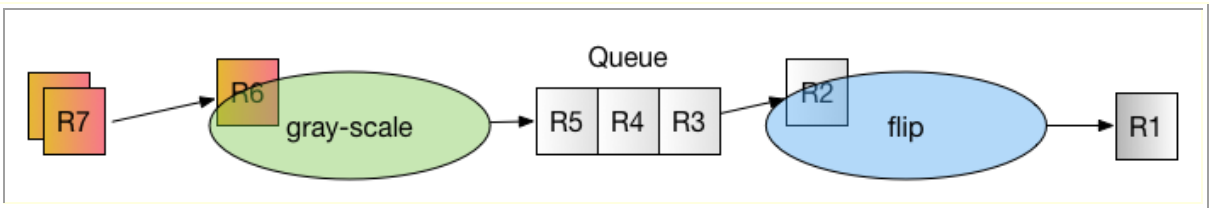

Figure 3: Implementing a pipeline with a queue.

***Note to the instructor:** By the above example of functional decomposition, the instructor can illustrate **Task Parallel** concept.

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.

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:

```
 1 #include <iostream>
 2 #include <fstream>
 3 #include <stdexcept>
 4 #include <sstream>
 5 #include <sys/time.h>
 6
 7 #include "libppm.h"
 8
 9
10 int main(int argc, char *argv[]) {
11
12   if (argc != 3) {
13     std::cerr << "Usage: " << argv[0] << " in_ppm_file out_ppm_file" <<
14 std::endl;
15     return 1;
16   }
17
18   PPM_header img_header;
19
20   try {
21
22     std::ifstream ifs(argv[1], std::ios::binary);
23     if (!ifs) {
24       throw std::runtime_error("Cannot open input file");
25     }
26     PPM_read_header(ifs, img_header);
```

```
27
28      std::cout << img_header << std::endl;
29
30      RGB_8 *img = new RGB_8[img_header.height * img_header.width];
31      PPM_read_rgb_8(ifs, img_header.width, img_header.height, (RGB_8 *)
32  img);
33
34      std::ofstream ofs(argv[2], std::ios::binary);
35      if (!ofs) {
36        throw std::runtime_error("Cannot open output file");
37      }
38      PPM_write_header_8(ofs, img_header.width, img_header.height);
39      PPM_write_rgb_8(ofs, img_header.width, img_header.height, (RGB_8 *)
40  img);
41
42      ifs.close();
43      ofs.close();
44
45    } catch (std::runtime_error &re) {
46      std::cout << re.what() << std::endl;
47      return 2;
      }
      return 0;
    }
```

Note 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 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 advantage of parallelism.

You will modify your code so that the grayscale() function enqueues the row that it just 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.

```
 1 RGB_8 *dequeue(std::queue<RGB_8 *> &q) {
 2    RGB_8 *image_row;
 3   bool keep_checking = true;
 4   while (keep_checking) {
 5 #pragma omp critical (pipeline)
 6     {
 7       if (!pipeline.empty()) {
 8         image_row = pipeline.front();
 9         pipeline.pop();
10         keep_checking = false;
11       }
```

```
12     }
13 #pragma omp taskyield
14   }
15   return image_row;
16 }
17
18 void enqueue(std::queue<RGB_8 *> &q, RGB_8 *row) {
19 #pragma omp critical (pipeline)
20   {
21     pipeline.push(row);
22   }
23 }
```

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.

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. 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.

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 .