Finding Horizontal and Vertical Intensity Gradients

David Duy Ngo

High Performance Computing, University of the Pacific

Abstract— **This project is an introduction to parallel programming with the *pthread* library. The core project is an image processing technique using Gaussian convolution. The article explains the parallelization methodology and compares the time results of the multi-threaded program against the serial program.**

I. Introduction

This program is a basic image processing solution that generates a horizontal and vertical intensity gradient image. The serial program takes the input on the terminal in the form of:

```
./serial <file_name> <sigma>
```

In which <*file_name*> is an image file of type *.pgm* and <*sigma*> is a floating point value used to generate the kernel. For this particular experimentation, sigma will be 0.5.

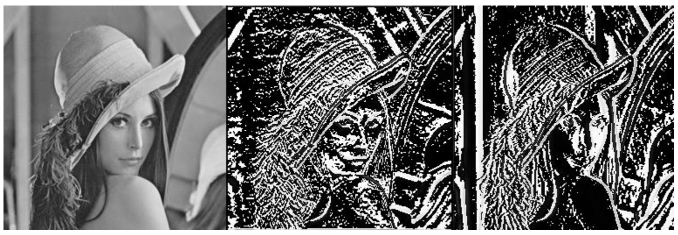

Figure 1. From Left to Right: Original image, Horizontal Gradient, Vertical Gradient

The program begins by generating a kernel mask in which the size is dependent on the *sigma* argument. Both the Gaussian kernel and the Gaussian derivative masks are generated. The program then continues to perform the convolution operation on the reference image for both horizontal and vertical gradients. The convolution operation is a computationally expensive algorithm in which the output image is generated per pixel by calculating the sum of products of each pixel in relation to the masks.

After the convolution operations for both the horizontal and vertical gradients, the images would then be written into new files.

II. Parallelization Methodology

The parallel version of this program uses the *pthread* library. Executing the parallel program from the terminal is as follows:

```
./parallel <file_name> <sigma> <num_threads>
```

A. Preparations

Because of the shared-memory architecture, I defined my global variables as:

```
int num_threads;
float *kernel, *deriv, sigma;
Image temp_image, ref_image;
```

In which the *Image* structure stores the actual 1D image array of size *width*height* and the dimensions of the image. The image structure is as follows:

```
typedef struct {
            float* image;
            int width;
            int height;
} Image;
```

In addition to the Image structure, I have created a *thread_data* structure:

```
typedef struct {
            int thread_id;
            int ker_w;
            int ker_h;
} thread_data;
```

B. Parallelization

The core idea to parallelize this program is to assign each thread its own section of the image to perform the convolution operation on according to its id. Calculation of each thread's domain is as follows:

```
start = data->thread_id*((ref_image.height)/num_threads);
end = start +((ref_image.height)/num_threads);
```

Visually, the assignments would look like this:

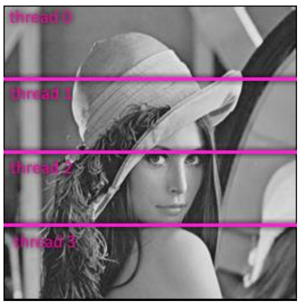

Figure 2

The main thread handles the execution and joining of sub-threads and also handles file read/write to avoid race conditions.

Main Thread:

1. Read/Store Reference Image
2. Generate Gaussian Kernel/Derivatives
3. **Pthread Create**
   a. Sub-threads perform horizontal convolution with both Gaussian kernels and derivative.
4. **Pthread Join**
5. Write *horizontal_derivative.pgm*
6. **Pthread Create**
   a. Sub-threads perform vertical convolution with both Gaussian kernels and derivative.
7. **Pthread Join**
8. Write *vertical_derivative.pgm*
9. **End**

Because of the way the serial program was written, I had to perform the horizontal convolutions and vertical convolutions through separate thread sessions.

III. Results

The system I am using is a *VMware* running Ubuntu with 2 allocated cores and 1 GB of RAM.

A. Runtimes (Local Machine)

Because of the limitations of my hardware, the runtimes of the number of threads exceeding 4 threads are very similar.

	Serial	2 threads	4 threads	8 threads	16 threads
512x512	286	180	385	257	233
1024x1024	1142	668	747	695	750
2048x2048	4434	2574	2585	2494	2521
4096x4096	18178	9583	9304	9243	9498
5120x5120	28764	14373	14562	15204	14525
10240x10240	150072	113244	90405	92498	151212

Figure 3 Runtime Table with times in Milliseconds (local)

According to *Figure 3*, the 2-thread program's runtime is half to that of the serial programs, which is a great sign of strong scalability. However as I increase the number of threads, there is no virtually no change to the runtime. A clear explanation to this issue is because of hardware limitation with my system, which can be tested when I run the program again on the cluster.

However, other factors in my code that would affect the parallel performance include the job of the main thread. The threads are created and joined twice with each session handling a horizontal or vertical convolution. I can further optimize my solution by creating more image buffers (*horiz_image*, *vert_image*) for my threads to handle in one session. That way, I can reduce the number of thread creation and ultimately reduce the overhead time it takes to create and join the *pthreads*.

B. Runtimes (Cluster)

After running the tests on my local machine, I have imported the project files to the cluster.

	Serial	2 threads	4 threads	8 threads	16 threads
512x512	274	273	199	160	40
1024x1024	903	494	290	202	128
2048x2048	2984	1674	1085	653	462
4096x4096	10883	5880	3608	2202	1551
5120x5120	16592	9109	5320	3298	2137
12800x12800	106210	54514	31186	18978	12675

Figure 4a Runtime Table with times in Milliseconds (cluster)

Without the technical limitations of my local machine, there is a clear difference in runtime between threads. Just like before, the runtime for 2 threads is close to half of that of serial. The performance of the program continues to improve as the number of threads increase.

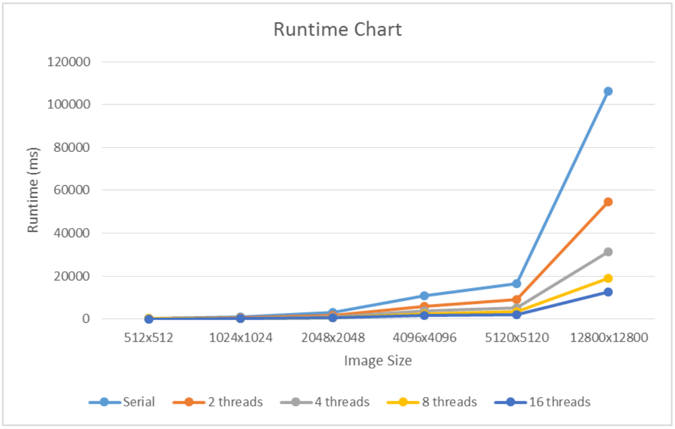

Figure 4b Runtime Chart to show speedup of threads

Although I did stop testing after 16 threads, the current pattern shows that the program will continue to speed up as I continue increasing the thread count. The only factors that would eventually peak the performance would be the read/write times of the file and overhead time when initiating the threads.