Gradient and Phase Images using Pthreads and OpenMP

David Duy Ngo

High Performance Computing, University of the Pacific

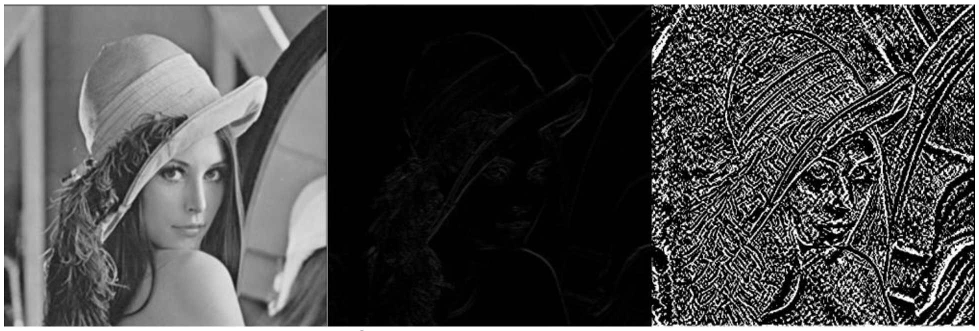
Figure 1 From Left to Right: Original image, Magnitude, Phase

Introduction

This project is a continuation of Project 2: Finding Horizontal and Vertical Intensity Gradients with the addition of two new operations: Magnitude and Phase. I will also test the parallelization using both *OpenMP* and *pthreads* and compare the effort required to program them with their respective speed-ups.

In terms of the serial program, the only addition I made was the *calculate_magphase()* function.

```
void calculate_magphase(Image horiz, Image vert, float* mag, float* phase){
        int width = horiz.width, height = horiz.height, i, j;

        for (i = 0; i < height; i++){
                for (j = 0; j < width; j++){
                        mag[i*width + j] = sqrt(pow(vert.image[i*width +j], 2) + pow(horiz.image[i*width +j], 2));
                phase[i*width + j] = atan2(vert.image[i*width + j], horiz.image[i*width + j]);
                }
        }

}
```

Parallelization

Pthreads

The code from Project 2 remains the same except for a few tweaks. I have written another thread function *thread_magphase()* that calls the *calculate_magphase()* function for each thread. In the *main* function, there is a totally of three loops to create and join the pthreads.

OpenMP

I used the serial program as the base for the OpenMP program. For the most part, the base code remains the same. In the main function, I initiated the OpenMP threads:

```
int num_threads = omp_get_num_procs();
omp_set_num_threads(num_threads);
```

The only next two additions I have made were the *#pragma* statements in both the *convolution()* and *calculate_magphase()* functions. Otherwise, the whole serial program remains the same.

Results

The system I am using is a *VMware* running Ubuntu with 2 allocated cores and 1 GB of RAM. I will also be running the program on the cluster for comparison.

Local Machine

	512x512	1024x1024	2048x2048	3072x3072	4096x4096
Serial	349	1400	5852	13155	23579
OpenMP (2 Threads)	263	842	3494	7322	13905
Pthreads (2 Threads)	267	864	3392	7622	13362

The OpenMP program can only detect 2 threads in my system. Therefore, I based this test with only two threads. It seems that although there is some speedup between the serial and parallel programs, there is virtually no difference between the OpenMP and Pthread programs.

Cluster

	512x512	1024x1024	2048x2048	3072x3072	4096x4096
Serial	244	973	3210	7189	12377
OpenMP (48 Threads)	38	77	280	532	873
Pthreads (48 Threads)	99	296	896	2012	3898

On the cluster, the OpenMP program detected a total of 48 threads. Therefore, I loaded 48 threads in the pthread program as well. The OpenMP program clearly performed much faster than the pthread program with around a 3x speedup. This was a drastic difference when compared to the program on my local machine. The only explanation I have for this result is probably that I would need a higher thread count than just two threads.

Conclusion

It takes much less effort to write the OpenMP program versus the pthread program. With the pthread, I needed to write additional thread functions and calculate the indices that each thread would need to be in charge of. Not just for this particular project, but also with project two I needed to initialize pthreads and allocated memory for each one. Such tasks (including debugging and rewriting) can take me hours to accomplish.

The OpenMP program is a whole lot more forgiving. I only needed to initialize OpenMP and the library does the rest. The only two additions I have made to the program were the *#pragma* statements to specify which for loops needed to be parallelized and which variables are private to each thread. Such tasks can require at most 1 hour to figure out. That and also considering that the OpenMP program performed near 3x faster than the pthread program makes OpenMP the better method of parallelization.