University of the Witwatersrand

Parallel Computing: Image noise filtering project

Liam Pulles

September 30, 2016

1 Introduction

1.1 Problem statement

1.2 Proposed solution overview

1.3 Background Theory

2 Solution technicalities

2.1 Serial Implementation

The serial implementation works as follows:

1. An image is loaded into memory (a pointer to BMP).
2. The matrix structure is created (we've made a few standard 5x5 and 3x3 matrices).
3. The covolution function is called.
4. The convolution function makes a blank output BMP* structure with the same dimensions as the source image for storing the calculated pixels.
5. The covolution function calls the kernel function for each pixel, using two nested for loops.
6. The kernel function calculates the kernel for a pixel using the edge extrapolation function, the matrix sturucture, and the source image. The final calculation for the pixel is stored in the output BMP* structure directly for speed.

7. Once the whole covolution is complete, the program writes a new file with the filtered image.

2.2 Parallel Implementation

The parallel implementation differs from the serial implementation in that instead of the master thread (the serial program) iterating through each pixel, the parallel implementation has several threads going through different pixels at the same time.

This is achieved by use of an omp for loop structure. We use dynamic scheduling to accomodate for potential differences in complexity in regions of the image. Each iteration of the outer for loop calculates the kernel for a (image height)x([matrix width]x2) sized chunk of the image (See figure 2.1) (Note that the final chunk is truncated to fit with the width of the image). There are a few reasons for this choice of task decomposition:

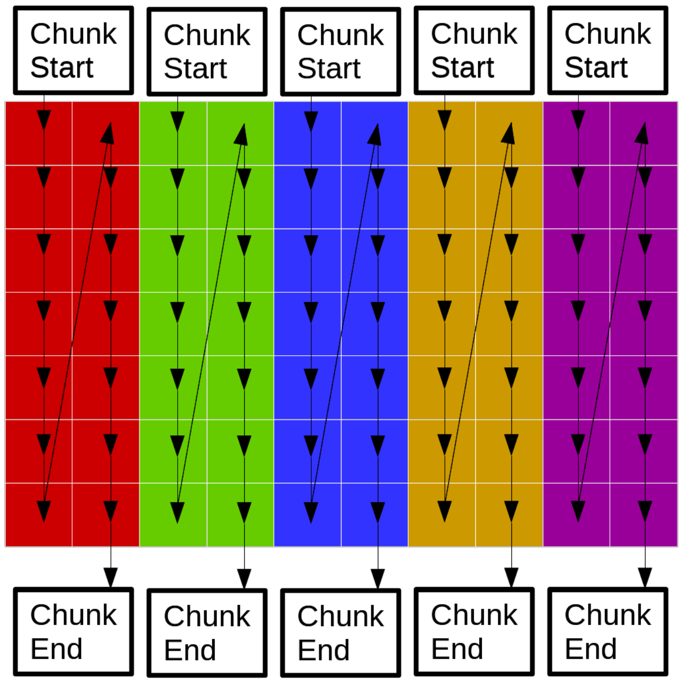

Figure 2.1: The decomposition of the workload of the image into chunks.

- We want to to keep the number of chunks for threads to process low so as to reduce unproductive inter-thread communication time, but we also want there to be enough chunks so that threads that execute their chunks quickly can still have something useful to do while waiting for other threads.

- We want the dimensions of the chunks to minimize potential contention between threads, specifically when we are calculating the kernel for two seperate pixels which are in kernel dimension range (this would result in both the kernel calculations needing to contend for the same pixels for their own calculation, see figure 2.2). Assuming the image is reasonably sized, using (matrix width)*2 width chunks will nearly always keep the kernel calculations far enough away from each other at each moment of execution, even when one thread is running faster than the others. For a 5x5 kernel, a thread would have to have done more than 5 columns of the image before one thread had done a single column for there to be contention issues in this regard.

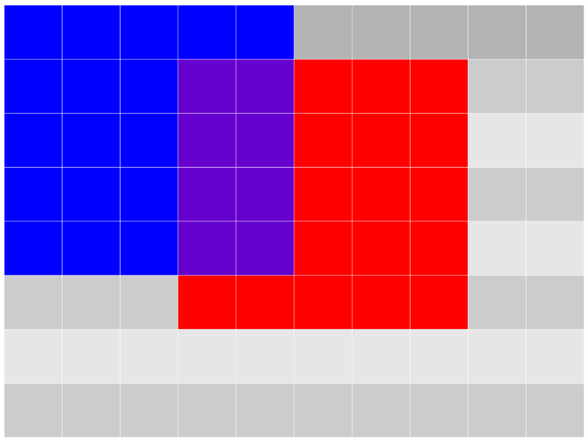

Figure 2.2: An example of contention with nearby simultaneous kernel calculations. The purple region represents potentially contested pixels.

Each thread works column by column. We do not create seperate sub images for each thread to use for source and output because it would require splitting overhead at the beginning and joining overhead at the end. It is mentioned in the EasyBMP documentation (EasyBMP Manual 1.06, pg. 16) that the RGBApixel pointers are stored internally as a 2d array. We also know that this must be stored on the heap, since it has to do deal with varying sized images (and thus, varying sized 2d arrays). Finally, we can deduce that since the size of the array is width x column that these 2d arrays store arrays of rows (the x component of the image).

So what about the contention that arises when threads read from the same source image in memory at approximately the same time, and write to the same output image at approximately the same time. Will contention on the cache line arise from reading/writing the same row?

We compared two version of the parallel implementation - one which sequentially accessed by row and one which sequentially accessed by column. The difference in execution time was negligible.

Why is this? We believe it is because the 2d array stores pixel POINTERS.

Suppose there is contention between 2 threads trying to access far away pixels on the same row, and - we shall presume for argument's sake - the same cache line. Then one thread will have to wait while the other gets the pointer data (which is of size int). But after that contention, mofiying or reading to that RGBApixel - which forms the vast bulk of the process of modifying a pixel - will be independent of modifyng or reading any other RGBApixel, since it is a pointer and can exist anywhere in the heap and thus on seperate cache lines.

So since the delay caused by the contention is so small, it has an almost zero sum effect on the toal execution time. The number of contentions also becomes less over time, as once a contention does arise, one of the threads will move forward while another has to wait. So ultimately, the threads will often be working on both different rows and different columns anyway.

3 Results analysis

3.1 About the code

- **To compile** : make
- **To Run**

  Serial Version : ./main source.bmp output.bmp **s**

  Parallel Version : ./main source.bmp output.bmp **p**

3.2 Analysis

The following results were obtained by using a 5x5 Gaussian blur kernal matrix on the image set stored in TestBMP.
Each .bmp image has a resoloution of 2560x1440 px resulting in a size of 11.1Mb per image. Now, Because of this large resoloution each image was processed 10 times and the average of that processing time was recoreded to promote more accurate results.

However, here we should note that all code for testing (controll for loops) have been removed for the final submission.

The following table illustrates the parallel and serial processing time for each image.

Image Used	Serial Time(S)	Parallel Time(S)	Difference(S)
1.bmp	5.31379	5.13833	0.17546
2.bmp	5.45032	5.11123	0.33909
3.bmp	5.84925	5.46885	0.3804
4.bmp	5.22041	5.20856	0.01185
5.bmp	5.20246	5.13382	0.06864
6.bmp	5.20605	5.11949	0.08656
7.bmp	5.23101	5.08228	0.14873
8.bmp	5.20282	5.1518	0.05102
9.bmp	5.21546	5.1153	0.10016
10.bmp	5.18062	5.11094	0.06968

Using the above table we can see that on average our parallel implentation lowers our processing time by 0.143159 seconds.
The reason for this increase in performance is explaned in section 2.2

The following diagram is a graph which represents the above table in a visual way...

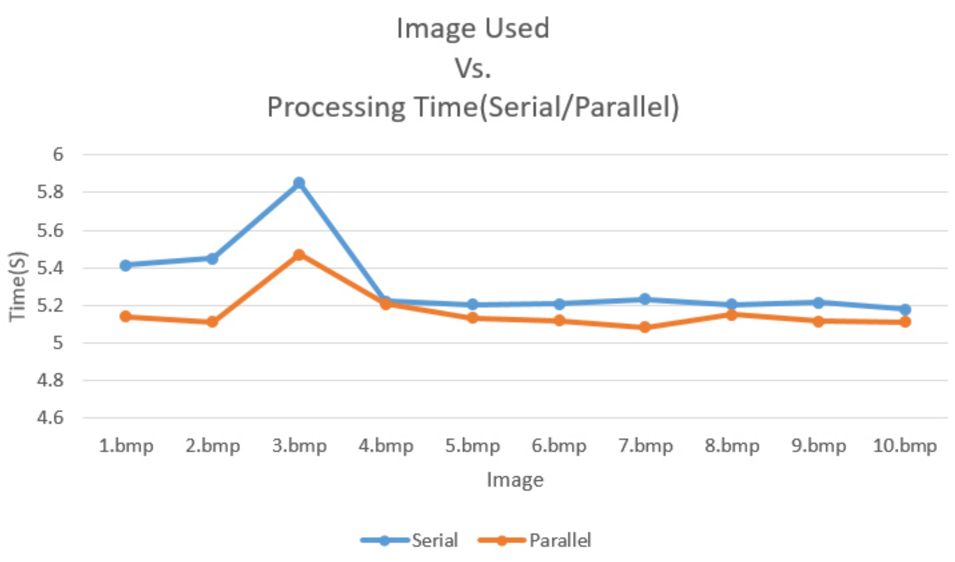

Figure 3.1: Serial Vs Parallel (Lower is better)

Here, using the above graph it becomes easier to see that our parallel implemtation out performs our serial implentation everytime.

4 Conclusion