DVA336 - Project Part 2

André Caldegren acn15007@student.mdh.se
Anton Roslund ard15003@student.mdh.se

January 12, 2018

1 Introduction

A mean image filter is used to give an image a blur effect. It can be used to clean up noise in an image at the cost of sharpness.

A mean image filter is an embarrassingly parallel problem since the color of each pixel is calculated independently. Since the problem is embarrassingly parallel we expect a performance increase relative to the number of processors used. I will probably not be perfect since there is some overhead related to managing the threads.

2 Sequential solution

We choose loop trogh the pixel in row-major order because that's how they are stored in the bitmap file format [1].

```c
// Lopping through all pixels
for (int y = 0; y < height; y++){
  for (int x = 0; x < width; x++){
    Color color = 0;
    int neighbours = 0;

    // Looping through neighbours, in this case it's 3x3
    for(int i = x-1; i <= x+1; i++){
      for(int j = y-1; j <= y+1; j++){
        // Checking if neighbours are inside of the image
        // Because the first pixel (top left)
        // would only have 3 neighbours + itself.
        if(i >= 0 && j >= 0 && i < width && j < height){
          neighbors++;
          color += image_in[i][j];
        }
      }
    }
    image_out[y][x] = color /= neighbors;
  }
}
```

3 Parallel solution

We have two different solutions to the parallelization of this problem. Both solutions achieve the same goal, but the code complexity differs quite a lot. The first solution is using POSIX threads, while the second solution is using OpenMP. Both solutions have the goal of increasing the speed of execution by dividing up the work that needs to be done among several threads instead of just one.

Since POSIX threads are quite low level more code is needed to make them work compared to OpenMP. We first need to initialize every thread, and all the parameters that are going to be sent to the thread. The area of the image that every specific thread is supposed to do their calculation on need to be "manually" calculated. The input image is divided along the y-axis into as many parts as there are threads. After the allocation is done and execution is under way every thread does their part to complete the whole picture.

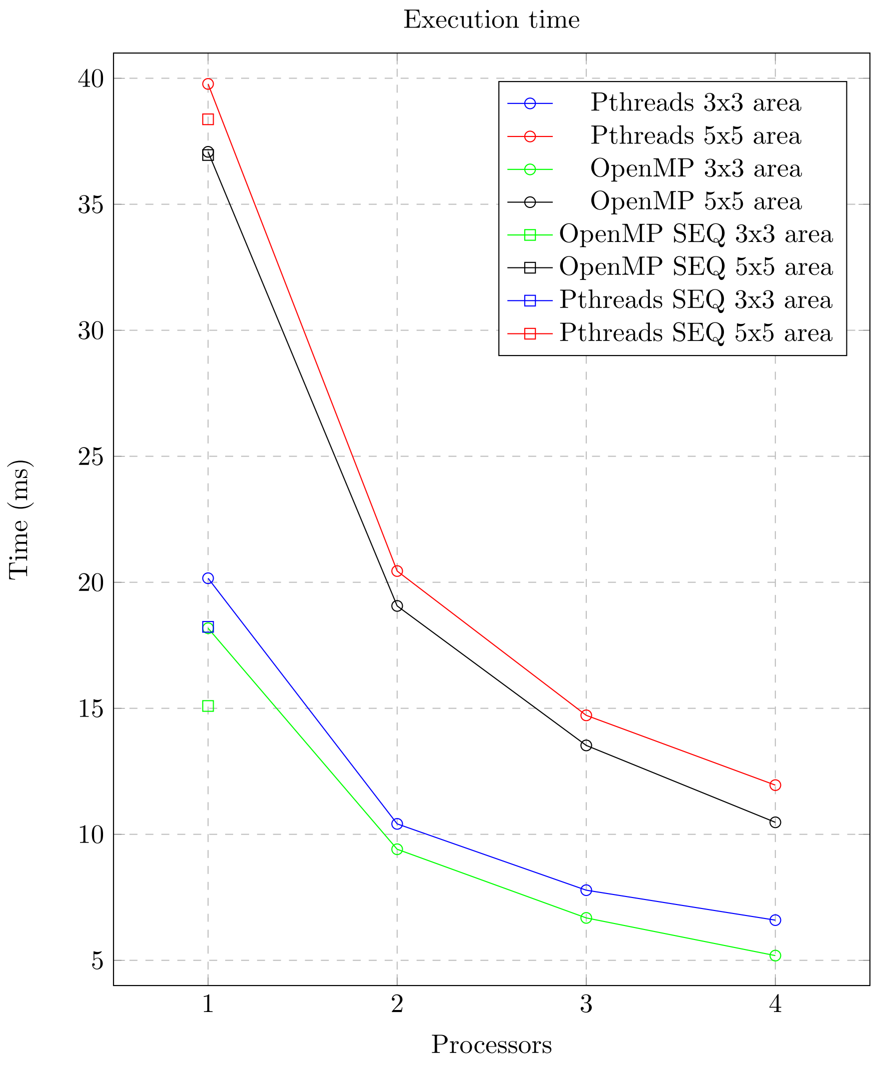
OpenMP on the other hand only requires that one line of code is changed from the sequential algorithm. The library takes care of the rest. There is not really any complexity added in the code. This is nice for the programmer, it makes it a lot easier to focus on coding a good algorithm instead of having to mess around with threads. At least in a problem like this it is not necessary for the programmer to have full control over the threads.

What is common among both solutions are that all the threads operate on the same input image and output images. This means that there is no stitching of different, smaller images, necessary when the threads are done with their calculations. Both the POSIX and OpenMP algorithm is basically the same as the sequential. The only difference is for POSIX threads where the loops that loop through the image needs to be restricted.

4 Performance

The performance was measured on a school computer with 4 cores. The code was compiled with `g++ -Ofast -o main main.cpp`. The Pthreads version used the flag `-pthread` And the time was messaured using an $1000 * 1000$ bmp image. The OpenMP version used the flag `-fopenmp`. Instead of using and image the OpenMP version uses a randomly generated $1000 * 1000$ `unit8_t` matrix, which might re responsible for the time difference between OpenMP and Pthreads.

Measurements are using `gettimeofday()` time for loading and storing the image is exluded, only the calculation is measured. All times are in milliseconds and from an average of 100 runs.



5 Validation

Checking to see if our parallel algorithm gave the correct output image was not all that hard. Analyzing the output image and comparing it to the input image was enough to see a clear distinction. If you wanted to make further analyzations you could compare the output image of the sequential algorithm with the output image from one of the parallel algorithms.

Since the algorithm is basically the same for both the POSIX and OpenMP version there should not be any discrepancies between those outputs. When we compared output images from both versions that is exactly what we found. Both images are the same, and both images are also the same as the output from the sequential code.

One last thing we did to be sure that our output image was correct was to use a third party. We entered the same image into a third party's image filter system. This was done just in case something was horribly wrong with our sequential algorithm that everything was built upon. And then we compare the result of their output to the result of our output. The result was as expected, all good.

6 Instruction

There are two ways of getting this project, either you can use the zip file provided by us, our you can clone the public github repo found at Enari/DVA336Project. After unzipping everything, you can find that we have provided both the source code, a makefile and the library that we use to manipulate images. Since this program uses POSIX threads, you can only run this program on Unix based systems like Linux or Mac.

To run the program all you must do is open a terminal, go to the folder where the makefile is and type make. An executable will then be generated for you which you then can run from the terminal.

To change the input image, you need to change the code itself. The image must be of the BMP format and it needs to be perfectly square. The number of threads and the area of the average can also be found in the code and can be changed. The final image will be call out.bmp and saved in the images folder.

7 Conclusion

In conclusion, we can clearly see that parallelizing this algorithm has had a profound effect on the execution time. By increasing the amount of threads, we can see that our execution time drops like predicted. Of course, you cannot just keep increasing the amount of threads and expect a lower execution time. You need to take the number of processor cores available into account.

References

[1] The bmp file format. `https://en.wikipedia.org/wiki/BMP_file_format`. Accessed: 2018-01-09.