Technology has improved in leaps and bounds for decades. With machines having multicore CPUs, it is possible to have threads which run in parallel. This report will be discussing the use of multiple threads to perform a task to blur images concurrently.

There are multiple ways which an image can be partitioned in order to perform the blur in parallel. This report will compare the time it takes for each of them to completely blur the image. With a sequential case as the base line, the partition which will be compared to the base line includes a separate thread per pixel, a thread per row, a thread per column, and a thread for each square with width of ten, twenty, thirty, forty, and fifty percent of the average between row and column. The functions have been named as parallel\_blur\_*type* where the *type* is the type of partition which each thread runs. In the case when the limit on the number of possible threads have been reached, it has been designed such that the threads will join before continuing. The thread limit has been set at forty thousand threads or lower if the limit which the operating system sets is lower than forty thousand.

 BlurExprmt file takes in two arguments, one being the path to the image to blur and the other being the path to store the blurred image. The program will start off by blurring the image and store it. Then, it will then go through each of the blur types, including sequential, two hundred times each and stores the time it takes to loop through and blur the whole image in result.csv. The result in this report will be from running the program on Ray machine at Imperial College London with CPU’s specification being HP EliteDesk 800 G3 TWR, Nvidia GPUs (Geforce 1080 8GB) Intel Core i7-7700K 4.20GHz and 16 GB ram on Linux.

Blurred image of the original.

Size: 640 x 640.

Original image used for this experiment.

Size: 640 x 640.

More information on the data can be seen in the appendix. Since each of them is executed two hundred times to reduce random error, the data generated has been represented using the box plot below. The plot for Sequential and Pixel has been removed since they are less significant.

|  |  |  |
| --- | --- | --- |
|  | Mean (ms) | Median (ms) |
| Sequential | 203.9307 | 205.5962 |
| Pixel | 8907.544 | 9116.535 |
| Row | 82.30187 | 82.07296 |
| Column | 78.17755 | 79.13713 |
| Sector10 | 45.88437 | 44.85326 |
| Sector20 | 42.37549 | 41.72079 |
| Sector30 | 43.76171 | 43.26958 |
| Sector40 | 47.88398 | 46.67729 |
| Sector50 | 63.97401 | 68.81808 |

During the implementation of pixel by pixel, there were not enough threads for every pixel thus when joining the thread which assumes the that all threads were created successfully, it causes errors and a very long run time. The fix was to change from joining all threads at the end, it joins whenever no new threads can be created before continuing to create new threads. Another issue was that the threads overwrite what was not supposed to be overwritten which causes the blurring to not be completely correct. This was fixed through the use of mallocs to store important parts of the program.

Figure 1

Figure 2

The results came out as expected. What was surprising was how long it took for the pixel by pixel case to complete the task as compared to doing it sequentially as shown in Figure 1. This is due to multiple reasons. Firstly, it takes time to initialize data to pass into pthread\_create whether its assigning value or assigning space to it via malloc. Secondly, it requires time to create new threads which is created for every single pixel in this method. Additionally, joining also takes time since it will wait until every thread is finished before continuing. Coupled with the final reason where there are not enough threads for every pixel, this requires multiple joining to be completed before being able to blur the whole picture.

In Figure 2, it can be seen that every single method of partitioning is faster than doing it sequentially. Furthermore, every sector partitioning is faster than row and column partitioning. This is due to likely to be due to caching and spatial locality and doing it in sectors allows more hits than in rows and columns. Similarly, as the size of the sector increases, it may cause more misses whereas if the size of the sector is too small, the cache will not be fully utilized as shown how sector20 is faster than sector10. However, since the sector is calculated in percentage, the result can greatly differ for larger images as the cache size is the same hence more experiments will be required. Additionally, as images get larger, it may not be possible for threads to get created and the time required may change this more data will be needed.

In conclusion, blurring images in parallel can reduce the execution time. However, whether it will truly reduce the time needed as well as how much depends on how the images are partitioned. If not done correctly, it can increase the time needed by more than forty times compared to doing it sequentially. Thus, it is recommended to try out different sector sizes for different machines and image sizes to fully optimize blurring the images.

**Appendix**

