Angel G. Carrillo Laguna 801141100 CIIC 4030-086

**Table: Performance**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| size(pixels) |  | 200 | 400 | 1000 | 5000 |
| time(ms) | SEQ | 0.09375 | 0.34375 | 2.0625 | 94.98438 |
| OMP | 0.078125 | 0.328125 | 1.984375 | 91.14063 |

To execute the median filter, it must be executed via linux terminal and/or gcc.

**./executableSEQ input.png output.png -lm -lpng**

**./executableOMP input.png output.png -lm -lpng -fopenmp**

The code was made using libpng for gcc, which is a library for image processing of png files. The algorithm will open the file and handle all possible exceptions caused by opening and reading the image. There are a few different if statements since images can be very different from one another for example opening a black and white picture is not the same as opening a colored picture. After opening the image, the algorithm will separate all the pixels in three different values: red, green and blue (RGB). Therefore, I made a header file where there is a type definition of a structure for RGB. After getting all the values, it will create three arrays to separate the RGB values into three separate arrays, one per color, and sort it. I could have used merge sort since it is simple to implement but I found that there is a function for quick-sort inside one of the standard libraries in C, so I used that. Also, quick sort is much faster which is good for performance. After sorting it will get the median of each array and depending on the variable ‘append’, which is a counter for how many pixels can change their locations, the algorithm will either replace a pixel hole with neighbor pixel or merge values to create a pixel and draw it on the holes. Then after rearranging the fixed pixels inside the array, the algorithm recreates the picture in a new file. This out put file is the filtered image.

The sequential code was much faster than expected. As it is read in milliseconds, the numbers for the execution time grow a lot faster than expected. The real sizes for the pictures used are , thus they are bigger than listed so it is understandable why the 5000pixels image took 94ms while the 1000 only took 2ms for the Sequential algorithm. On the other hand, as expected, the algorithm using OpenMP is much faster. OpenMP is a library for thread handling, used to enable parallelism in programs so that any process has lower execution time. Thus, by using OMP to parallelize the heaviest loops in the program, it can make it faster, unless you activate too many threads and thus it will be slower. In overall, the program using OMP was much faster, but it is mostly noticeable for bigger files as you can see the difference between 200pixels and 5000pixels.

PS. If you want to create noise filled images, I uploaded the make\_noise.m file which is for Matlab