Mosaic filtering in C with OpenMP

Objective

To implement a C program that can apply a Mosaic filter filter to a PPM image that is either in binary or plain text format. The program should take arguments in a predefined format and reassemble the image to output it as another PPM file

Achieved Implementation

A C program was created that can take the predefined arguments and process a PPM binary image with the mosaic filter. Solutions were coded in both a sequential manner and using openMP to parallelise the image processing. Unfortunately a solution for processing plain text imported PPM images could not be devised.

All compute times are for the Dog2048x2048.PPM image. For sake of comparison, this code was ran on an Intel Pentium 4451U @ 2.3GHz. This CPU is hyper threaded so more substantial performance gains may be seen on computers with a true quad core CPU.

Optimisation

Sequential

As a base, a 2d structure was used to represent the data imported from the PPM image. In this base program the right most index was iterated over first when processing the image to optimise the utilisation of the cache. References to global variables or variables outside the function was minimised however this was simple to implement given the nature of the program. Care was taken to not optimize the initial program too much as this can cause problems and delays with implementation. This initial version of the sequential program took 0.3210s timed using the clock_t method and dividing the CPU time by the CLOCKS_PER_SEC value to convert to wall time.

A second version of the program was developed which switched numerous things now that implementation was finished. First of all, the "2D" representation of the image data using an array of pointers was scrapped in favour of a flattened 1D array. The usual column index * column length + row index approach was used, still iterating over the second or "row" index to optimise cache utilisation. An unorthodox version of loop jamming was also implemented during the image processing and then assigning process. the loops cannot be properly jammed as we do need to iterate over all the indexes first to calculate the average of said pixels before we then iterate over them to assign the average into them. To optimise this, the indexes of the values are assigned into a local array and then re used which in our case helps because we are having to calculate the indexes of the pixels. This second iteration of the code ran in 0.2320s

OpenMP

The initial version of OpenMP implementation uses a OMP parallel section before the OMP parallel for loop. This allows us to store the values needed to calculate local average of the pixels locally which reduces the usage of critical sections which increases the parallelisation of the code.

The second iteration of the code used the OMP for collapse function. As per the OpenMP documentation, parallel efficiency increases as workload over the CPU cores increases (Essentially Amdahl's Law). The only change that had to be made to implement this was to move all declarations of variables into the second level of the for loop as the documentation for this feature states the nesting of the loops must be "pure" meaning the for loops are directly nested with no other lines of codes between them. This optimisation took the run time of the code from 0.2210s to 0.1940s.

Release Mode

After finishing the implementations that I could achieve, the compile mode was switched from Debug to Release in order to allow the compiler to integrate it's own optimisations. The times of 0.2320s and 0.1940s for Sequential and OpenMP respectively were reduced to 0.0940s and 0.1000s. Again this is not being run on the most thread friendly machine however this overtaking of efficiency by the sequential program indicates a poor implementation of OpenMP.

Issues

As stated above, implementation of reading plain text PPM images was not achieved. Using the Visual Studio debugging tool, it was found that whilst the data seemingly imported fine, when processing, the pixel values were inexplicably set to 0. This issue was likely due to some sort of casting issue however no solution was devised in time.

Evaluation

The missing bits of implementation are disappointing however with the time and support available the issues could not be investigated further meaning suggestions for further improvements are extremely limited. The disappointing efficiency of OpenMP in release mode would perhaps improve on a machine with more threads available and it is not known if a machine with 4 "true" threads would perform better as a suitable machine to test this could not be found.