# Assignment 1

## Question 1

For both programs, the main performance bottleneck was in the Gaussian Blur portions, and in the final image assembly/writing. Because of Amdahl’s law, these limited the speedup and so I first optimized these portions.

**Threads:**

For the threads program, I parallelized the Gaussian Blur for the Red, Green and Blue channels, since they operated on data that was entirely separate. I created two threads to work on the Red and Blue arrays, while the main thread worked on the green channel. Joining the other two threads after this parallel section ensures that the destination buffers already hold the correct image data.  
Then, I parallelized the assembly of the destination buffers into a single image buffer, by splitting up the image into rows and allowing each thread to fill up a few rows, thus dramatically speeding up this process. The writing section was refactored to use the image buffer in a single fwrite operation, to reduce incurring syscall overheads, as was the case in the unoptimized, program, where fwrite would be called once for each pixel-channel.

Refer to the table in 2a. If we consider as the best sequential algorithm, we can see that the threads program has a maximum speedup of .

**OpenMP:**

Similarly, for blur\_omp, the Red, Green and Blue channels were parallelized to run on multiple OpenMP processes, thus reducing the time required to complete the Gaussian blur section. Because assembly of the channels into a destination buffer could deal exclusively with blocks of rows that operated independently from each other (unlike the Gaussian blur process), this segment could easily be loop-parallelized with the OpenMP parallel for construct.

Again referring to the table in 2a and using as the best sequential algorithm, the OMP program has a maximum speedup of

## Question 2

1. Table below: The data is available in results.txt, or by running make 2a  
   Timing is measured using the Unix time utility, and the program is run with

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Number of Cores | Execution Time | | Speedup | | |
| Threads | OpenMP | Sequential | Threads | OMP |
| 1 | Seq: 2.29  Threads: 2.30 | 2.89 | 1 | 1 | 1 |
| 2 | 1.25 | 1.27 | 1.83 | 1.84 | 2.27 |
| 4 | 0.92 | 0.94 | 2.49 | 2.5 | 3.07 |
| 6 | 0.92 | 0.94 | 2.49 | 2.5 | 3.07 |
| 8 | 0.93 | 0.94 | 2.46 | 2.47 | 3.07 |

1. There is a difference, likely due to the overhead of threads/OpenMP. On a single core, all the code is executed sequentially, but there are additional costs associated with context-switching.   
   Refer to Appendix A, for the results of a single core. We can see that the sequential program incurs 1 context-switch and uses 0.04s of kernel time. The threaded program uses 5 context-switches (on the same order of magnitude as 1), and also uses 0.04s. The OpenMP program uses a huge number (204) of context switches, and likely also uses more system calls, resulting in a higher kernel time.
2. Using to compute a speedup is not likely to be as useful as using , as the OpenMP program will incur the overheads of multithreading even with a single core. The result is as shown in the last column, where we might appear to get a superlinear speedup.   
   Using allows us to more accurately assess our parallel solution, as it also reflects the cost of parallelization. We realize, here, that while our program does result in a speedup, it does not scale proportionately with the number of cores.
3. Using an ideal program, we could use a total of eight threads, with one thread executing concurrently on each of the eight cores without thread-switching. This assumes that the data can be partitioned eight ways such that no significant synchronization is necessary, which does not apply in the program above (only a three-way partition of the data is done in blur\_threads.c)  
   An experiment would be to first design a variable-partition program, and then run it with various thread counts and measure the wall-clock time for each of these programs. Comparing the (repeated and averaged) results should give an idea of an optimal thread count.
4. An experiment could spawn a large number of threads, creating and joining each thread in sequence, as outlined below, and comparing it with another program that directly invoked the same function.

|  |  |
| --- | --- |
| start\_time = wall\_clock\_time();  for i from 0 to 1000 { | |
| Sequential Program: | Threaded program: |
| do\_nothing(); | thread\_create(t, do\_nothing)  thread\_join(t) |
| }  return wall\_clock\_time – start\_time; | |

The differences in execution time will be due to the overhead of threads, and dividing the differences in the timing of the two by 1000 will yield an approximate average per-thread overhead time.

## Appendix A: results.txt

====USING 1 CORES ===

====SEQ====

Real Time 2.29

CPU Time 2.25

Kernel Time 0.04

Context Switches 1

====THREAD====

Real Time 2.30

CPU Time 2.26

Kernel Time 0.04

Context Switches 5

====OMP====

Real Time 2.89

CPU Time 2.23

Kernel Time 0.05

Context Switches 204

====USING 2 CORES ===

====SEQ====

Real Time 2.30

CPU Time 2.25

Kernel Time 0.05

Context Switches 1

====THREAD====

Real Time 1.25

CPU Time 2.38

Kernel Time 0.05

Context Switches 4

====OMP====

Real Time 1.29

CPU Time 2.45

Kernel Time 0.04

Context Switches 199

====USING 4 CORES ===

====SEQ====

Real Time 2.32

CPU Time 2.27

Kernel Time 0.04

Context Switches 1

====THREAD====

Real Time 0.92

CPU Time 2.60

Kernel Time 0.05

Context Switches 5

====OMP====

Real Time 0.94

CPU Time 2.60

Kernel Time 0.06

Context Switches 197

====USING 6 CORES ===

====SEQ====

Real Time 2.40

CPU Time 2.34

Kernel Time 0.05

Context Switches 1

====THREAD====

Real Time 0.92

CPU Time 2.61

Kernel Time 0.04

Context Switches 8

====OMP====

Real Time 0.94

CPU Time 2.62

Kernel Time 0.05

Context Switches 199

====USING 8 CORES ===

====SEQ====

Real Time 2.30

CPU Time 2.26

Kernel Time 0.04

Context Switches 1

====THREAD====

Real Time 0.93

CPU Time 2.65

Kernel Time 0.05

Context Switches 7

====OMP====

Real Time 0.94

CPU Time 2.63

Kernel Time 0.05

Context Switches 207