Juan Herrera

Assignment 1

1a)

This is the output that is received when I ran the program it looks like it takes in an argument which represents the number of threads desired which are then created.

```
juan@DESKTOP-QCQU6MF:/mnt/c/Users/juane/code/csci551/assignment1$ ./a.out 5
Main program thread will now create all threads requested ...
Hello from OMP thread 0 of 5
Hello from OMP thread 3 of 5
Hello from OMP thread 1 of 5
Hello from OMP thread 4 of 5
Hello from OMP thread 2 of 5
All threads now done, main program proceeding to exit
juan@DESKTOP-QCQU6MF:/mnt/c/Users/juane/code/csci551/assignment1$
```

The pragma function looks a lot like the hello world function shown in the Chapter 5 Pacheco reading.

The reason the threads are not being displayed in order is because each thread is competing for access to stdout so there is no guarantee that the output will be in the assumed numerical order.

```
20  #pragma omp parallel for num_threads(thread_count)
21      for(int i=0; i<16;i++)
22      {
23          Hello_thread();
24      }
25
```

PROBLEMS 4 OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```
juan@DESKTOP-QCQU6MF:/mnt/c/Users/juane/code/csci551/assignment1$ ./a.out 4
Main program thread will now create all threads requested ...
Hello from OMP thread 1 of 4
Hello from OMP thread 1 of 4
Hello from OMP thread 1 of 4
Hello from OMP thread 1 of 4
Hello from OMP thread 0 of 4
Hello from OMP thread 0 of 4
Hello from OMP thread 0 of 4
Hello from OMP thread 0 of 4
Hello from OMP thread 2 of 4
Hello from OMP thread 2 of 4
Hello from OMP thread 2 of 4
Hello from OMP thread 2 of 4
Hello from OMP thread 3 of 4
Hello from OMP thread 3 of 4
Hello from OMP thread 3 of 4
Hello from OMP thread 3 of 4
All threads now done, main program proceeding to exit
juan@DESKTOP-QCQU6MF:/mnt/c/Users/juane/code/csci551/assignment1$
```

Yes there are alternative ways to use the omp directive for programs that call functions in loops simply add the *parallel for* directive and follow up by using a for loop immediately which will then divide the task up between all the available threads as even as it can

In order to print out the message 16 times you must make the for loop iterate 16 times which is done here

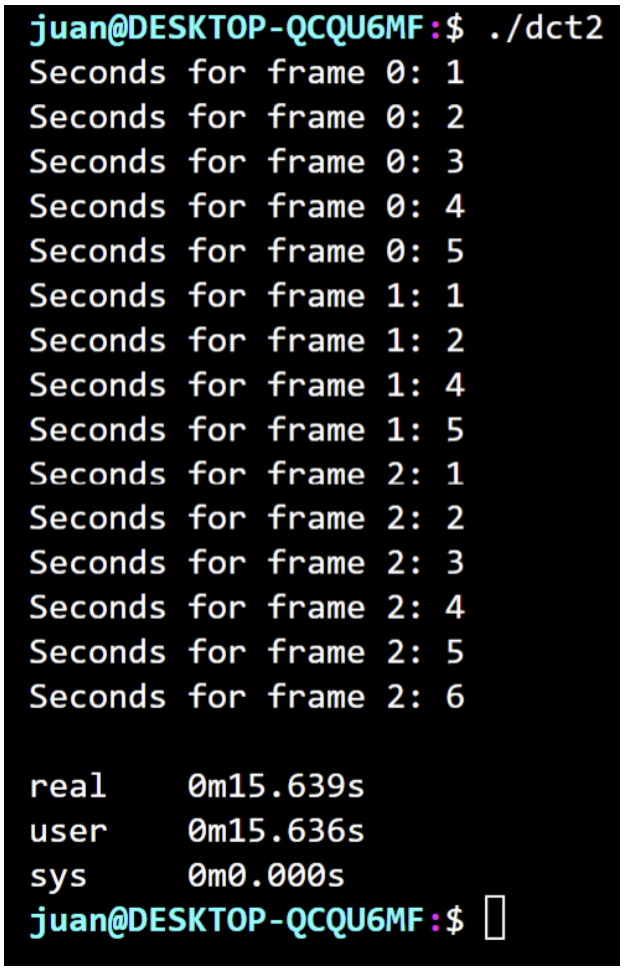
1b)

These are the results that I got when I built and ran it for the first time with the time function.

```
juan@DESKTOP-QCQU6MF:/mnt/c/Users/juane/code/csci551/assignment1/openmp_dct2/openmp_dct2$ time ./dct2

real    0m15.708s
user    0m15.704s
sys     0m0.000s
juan@DESKTOP-QCQU6MF:/mnt/c/Users/juane/code/csci551/assignment1/openmp_dct2/openmp_dct2$
```

It seems that every 1280x960 frame gets processed about every 5 seconds meaning that we can process .2 1280x960 frames per second.



I managed to get this calculation by making a counter variable that would keep track of the iterations the for loop that emulates a 1280x960 resolution image with one color channel. It was mentioned to us that every 30 seconds could be considered one second

```
// E.g. Since video is most often 30 Hz, or 30 frames/sec, 30 iterations is therefor like 1 second of video.
//      Adjust the iterations as is reasonable for your system!
```

Knowing this I simply kept track of every 30th iteration and made a separate seconds counter which kept track of this and incremented that counter whenever we hit a 30th iteration

```
// adding this counter variable in order to keep track of the iterations in the for loop in order to keep track of the seconds that have gone by
int count=0;

for(int frame_idx=0; frame_idx < MAX_ITERATIONS; frame_idx++)
{
    int seconds=0;

    // Emulate a 1280x960 resolutioon image with one color channel - gray
    for(int block_col_idx=0; block_col_idx < 160; block_col_idx++)
    {
        count++;//keeping track of the # of iterations
        //every 30 iterations will be 1 second
        if(count%30==0)
        {
            seconds++;
            printf("Seconds for frame %i: %i\n", frame_idx, seconds);

        }
```

1c)

These are the results when ompdct2 was built and ran for the first time

```
juan@DESKTOP-QCQU6MF:$ time ./ompdct2

real    0m5.831s
user    0m17.020s
sys     0m0.000s
juan@DESKTOP-QCQU6MF:$
```

I was able to determine that we are able to process about 1.9 1280x960 frames per second given that the real time in order to execute this was 5.700 seconds and we have 3 frames that need to be processed

```
juan@DESKTOP-QCQU6MF:$ time ./ompdct2
Seconds for frame 1: 1
Seconds for frame 1: 2
Seconds for frame 1: 3
Seconds for frame 1: 4
Seconds for frame 2: 1
Seconds for frame 1: 5
Seconds for frame 0: 1
Seconds for frame 2: 2
Seconds for frame 1: 6
Seconds for frame 0: 2
Seconds for frame 2: 3
Seconds for frame 2: 4
Seconds for frame 2: 5
Seconds for frame 2: 6
Seconds for frame 2: 7

real    0m5.700s
user    0m16.815s
sys     0m0.000s
juan@DESKTOP-QCQU6MF:$
```

I was able to determine this similarly to how I did the previous program I simply just made a couple of counters that would allow me to track every 30th iteration. This would show me how long it would take each frame to complete individually and since we are doing this in parallel we can see that it was more efficient

```
229     int count=0;
230 #pragma omp parallel for num_threads(thread_count)
231     for(int frame_idx=0; frame_idx < MAX_ITERATIONS; frame_idx++)
232     {
233         int seconds=0;
234         // Emulate a 1280x960 resolutioon image with one color channel - gray
235         for(int block_col_idx=0; block_col_idx < 160; block_col_idx++)
236         {
237             count++;//keeping track of the # of iterations
238             //every 30 iterations will be 1 second
239             if(count%30==0)
240             {
241                 seconds++;
242                 printf("Seconds for frame %i: %i\n", frame_idx, seconds);
243
244             }
```

1d)

Timing using 1 thread: it seems like this is doing exactly what the regular dct2 program is doing, this would make sense because it doesn't have extra threads to divide the workload essentially making it look like a sequential execute. TIME TO EXECUTE 15.987s

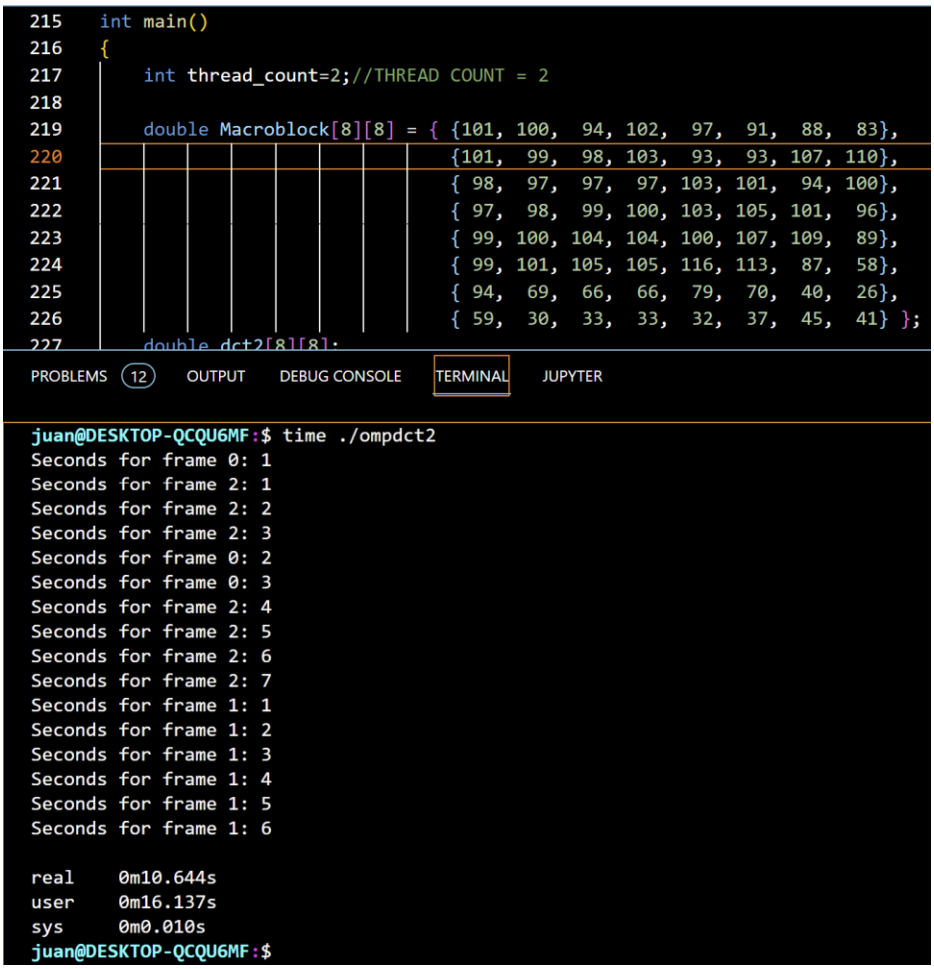
```
214
215   int main()
216   {
217       int thread_count=1;//THREAD COUNT = 1
218
219       double Macroblock[8][8] = { {101, 100,  94, 102,  97,  91,  88,  83},
220                                   {101,  99,  98, 103,  93,  93, 107, 110},
221                                   { 98,  97,  97,  97, 103, 101,  94, 100},
222                                   { 97,  98,  99, 100, 103, 105, 101,  96},
223                                   { 99, 100, 104, 104, 100, 107, 109,  89},
224                                   { 99, 101, 105, 105, 116, 113,  87,  58},
225                                   { 94,  69,  66,  66,  79,  70,  40,  26},
226                                   { 59  30  33  33  32  37  45  41} };
```

PROBLEMS 12 OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```
juan@DESKTOP-QCQU6MF:$ ./ompdct2
Seconds for frame 0: 1
Seconds for frame 0: 2
Seconds for frame 0: 3
Seconds for frame 0: 4
Seconds for frame 0: 5
Seconds for frame 1: 5
Seconds for frame 2: 1
Seconds for frame 2: 2
Seconds for frame 2: 3
Seconds for frame 2: 4
Seconds for frame 2: 5
Seconds for frame 2: 6

real    0m15.987s
user    0m15.983s
sys     0m0.000s
juan@DESKTOP-QCQU6MF:$
```

Timing using 2 threads: This was definitely faster than 1 thread by about 5 seconds: TIME TO EXECUTE 10.644s



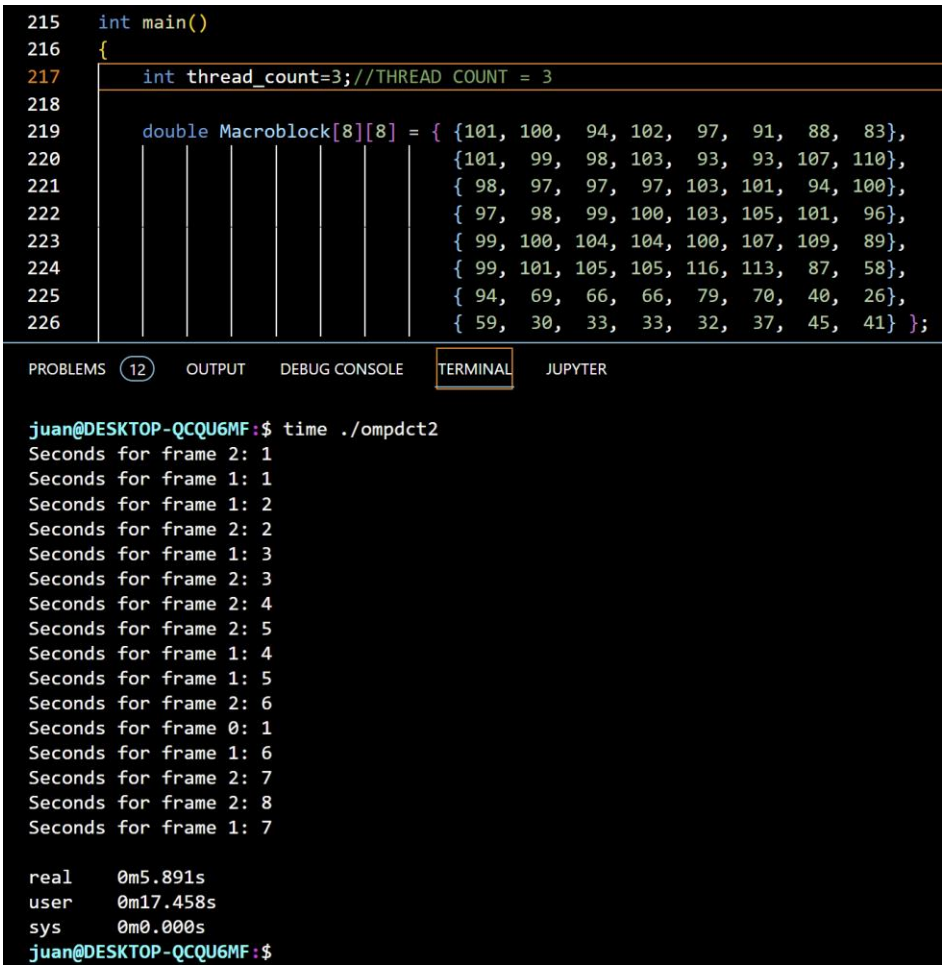
```
215  int main()
216  {
217      int thread_count=2;//THREAD COUNT = 2
218
219      double Macroblock[8][8] = { {101, 100,  94, 102,  97,  91,  88,  83},
220                                  {101,  99,  98, 103,  93,  93, 107, 110},
221                                  { 98,  97,  97,  97, 103, 101,  94, 100},
222                                  { 97,  98,  99, 100, 103, 105, 101,  96},
223                                  { 99, 100, 104, 104, 100, 107, 109,  89},
224                                  { 99, 101, 105, 105, 116, 113,  87,  58},
225                                  { 94,  69,  66,  66,  79,  70,  40,  26},
226                                  { 59,  30,  33,  33,  32,  37,  45,  41} };
227      double dct2[8][8];
```

PROBLEMS 12 OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```
juan@DESKTOP-QCQU6MF:$ time ./ompdct2
Seconds for frame 0: 1
Seconds for frame 2: 1
Seconds for frame 2: 2
Seconds for frame 2: 3
Seconds for frame 0: 2
Seconds for frame 0: 3
Seconds for frame 2: 4
Seconds for frame 2: 5
Seconds for frame 2: 6
Seconds for frame 2: 7
Seconds for frame 1: 1
Seconds for frame 1: 2
Seconds for frame 1: 3
Seconds for frame 1: 4
Seconds for frame 1: 5
Seconds for frame 1: 6

real    0m10.644s
user    0m16.137s
sys     0m0.010s
juan@DESKTOP-QCQU6MF:$
```

Timing using 3 threads: TIME TO EXECUTE 5.891s



```
215  int main()
216  {
217      int thread_count=3;//THREAD COUNT = 3
218
219      double Macroblock[8][8] = { {101, 100,  94, 102,  97,  91,  88,  83},
220                                  {101,  99,  98, 103,  93,  93, 107, 110},
221                                  { 98,  97,  97,  97, 103, 101,  94, 100},
222                                  { 97,  98,  99, 100, 103, 105, 101,  96},
223                                  { 99, 100, 104, 104, 100, 107, 109,  89},
224                                  { 99, 101, 105, 105, 116, 113,  87,  58},
225                                  { 94,  69,  66,  66,  79,  70,  40,  26},
226                                  { 59,  30,  33,  33,  32,  37,  45,  41} };
```

```
PROBLEMS 12   OUTPUT   DEBUG CONSOLE   TERMINAL   JUPYTER

juan@DESKTOP-QCQU6MF:$ time ./ompdct2
Seconds for frame 2: 1
Seconds for frame 1: 1
Seconds for frame 1: 2
Seconds for frame 2: 2
Seconds for frame 1: 3
Seconds for frame 2: 3
Seconds for frame 2: 4
Seconds for frame 2: 5
Seconds for frame 1: 4
Seconds for frame 1: 5
Seconds for frame 2: 6
Seconds for frame 0: 1
Seconds for frame 1: 6
Seconds for frame 2: 7
Seconds for frame 2: 8
Seconds for frame 1: 7

real    0m5.891s
user    0m17.458s
sys     0m0.000s
juan@DESKTOP-QCQU6MF:$
```

Timing using 4 threads: TIME TO EXECUTE 5.722

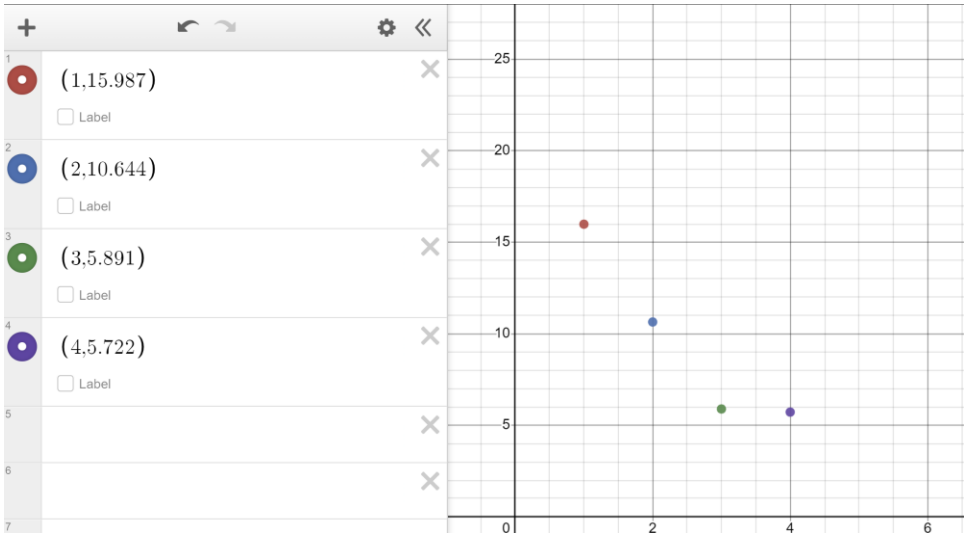
```
215  int main()
216  {
217      int thread_count=4;//THREAD COUNT = 4
218
219      double Macroblock[8][8] = { {101, 100,  94, 102,  97,  91,  88,  83},
220                                  {101,  99,  98, 103,  93,  93, 107, 110},
221                                  { 98,  97,  97,  97, 103, 101,  94, 100},
222                                  { 97,  98,  99, 100, 103, 105, 101,  96},
223                                  { 99, 100, 104, 104, 100, 107, 109,  89},
224                                  { 99, 101, 105, 105, 116, 113,  87,  58},
225                                  { 94,  69,  66,  66,  79,  70,  40,  26},
226                                  { 59,  30,  33,  33,  32,  37,  45,  41} };
```

PROBLEMS 12 OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```
juan@DESKTOP-QCQU6MF:$ time ./ompdct2
Seconds for frame 2: 1
Seconds for frame 0: 1
Seconds for frame 1: 1
Seconds for frame 2: 2
Seconds for frame 0: 2
Seconds for frame 0: 3
Seconds for frame 2: 3
Seconds for frame 2: 4
Seconds for frame 0: 4
Seconds for frame 0: 5
Seconds for frame 0: 6
Seconds for frame 0: 7
Seconds for frame 1: 2
Seconds for frame 1: 3
Seconds for frame 2: 5

real    0m5.722s
user    0m16.912s
sys     0m0.000s
juan@DESKTOP-QCQU6MF:$
```

Here is a graph for for the times where X is the number of threads and y is the time required to execute



1e)

The parallel portion of the code took 15.933594s. I was able to calculate this using the clock_gettime function.

```
Parallel portion took 15.933594
real    0m15.937s
user    0m15.934s
sys     0m0.000s
```

In order to determine the parallel and sequential portions of the code we would need to use Amdahl's Law which is equal to 1/(1-P)+P/S where 1-P is the sequential portion and P being the Parallel portion

$$\text{Amdahl's Law} = \frac{1}{(1-P)+\frac{P}{S}}$$

I was able to determine P by taking 15.933/15.937 which is .99 which means S is 1 - .99 which equals .01

2a)

In order to sharpen the image 3 times I had to change the number of iterations in the for loop provided, It was originally 90 but in order to do it 3 times I changed it to 270 for both files

```
//Changing the # of iterations to 90x3 to sharpen the image 3 times
#define ITERATIONS (270)
```

In order to keep track of the frame rate info I had to print to stdout the current frame and get the time for that specific frame. I did this in iterations of 30 frames.

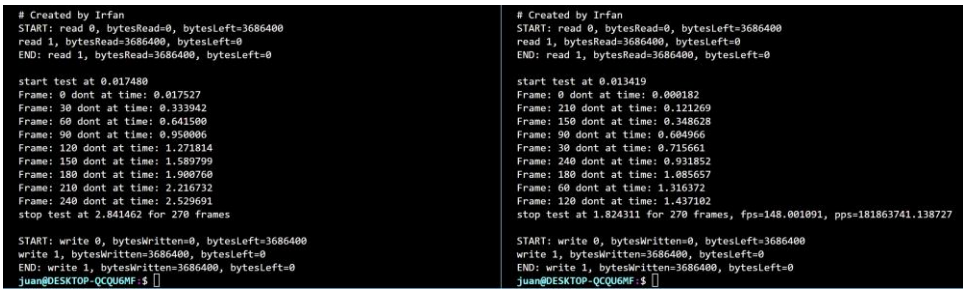
Left is non omp right side has the omp implementation

```
146
147   for(iter=0; iter < ITERATIONS; iter++)
148   {
149       //This is for reporting frame rate every 30 frames will be poste
150       if(iter%30==0)
151       {
152           clock_gettime(CLOCK_MONOTONIC, &now);
153           fnow = (FLOAT)now.tv_sec  + (FLOAT)now.tv_nsec / 1000000000.
154           printf("Frame: %i dont at time: %lf\n", iter, fnow-fstart);
155       }
```

```
149   #pragma omp parallel for num_threads(thread_count)
150       for(iter=0; iter < ITERATIONS; iter++)
151       {
152           //This is for reporting frame rate every 30 frames will be poste
153           if(iter%30==0)
154           {
155               clock_gettime(CLOCK_MONOTONIC, &now);
156               fnow = (FLOAT)now.tv_sec  + (FLOAT)now.tv_nsec / 1000000000.
157               printf("Frame: %i dont at time: %lf\n", iter, fnow-fstart);
158           }
```

These were the results:

Left is non omp Right side has omp implementation



The PSF illustrates the system impulse response which is similar to what the human eye does for example the first layer of the human retina transforms an image provided by the light entering the eye as a pattern of nerve impulses. The second layer would process these impulses and passes it to the third layer projects it onto the retina.

2b)

I compared both the single thread and Pthread PSF sharpen code for 90 iterations and it seems that the Pthread version was much faster

Left side: Sharpen.c Right side: Sharpen_grid.c

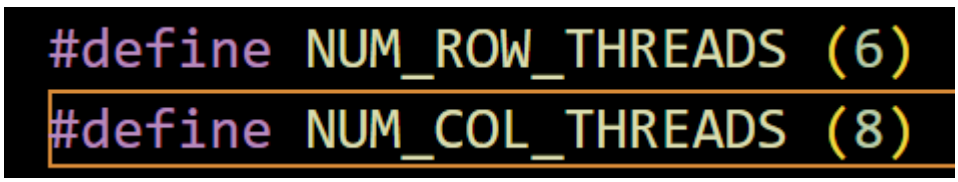
2c)

Comparing both the open mp and the Pthread versions on the PSF Sharpen code it seems to me that the Open MP was slightly faster. 0.436s faster to be exact

Left side:Pthread Right side: OpenMP

2d)

Comparing all of the times together I have decided to run the Pthread and OMP versions running at 14 threads each since my system has 16 cores available to it and in order to maintain the 4x3 aspect ratio on I needed to change the Pthread number of threads for columns and rows to be 8x6 giving me a total of 14 threads

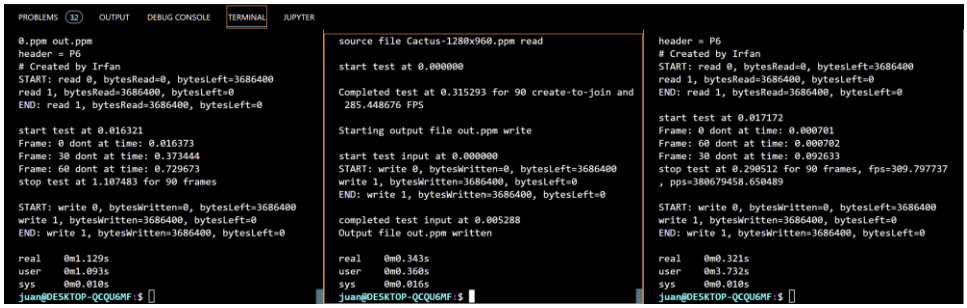


Doing so yielded these results:

Left side: Single Threaded 1.107s/1.129s

Middle: Pthread 0.315s/.343s

Right side: OpenMP .290s/.321s



Speed up for Pthread: P = .315/.343 = .91

$$\frac{1}{(1-.91)+\frac{.91}{14}} = 6.451612903$$

Speed up for OpenMP: P= .290/.321 = .90

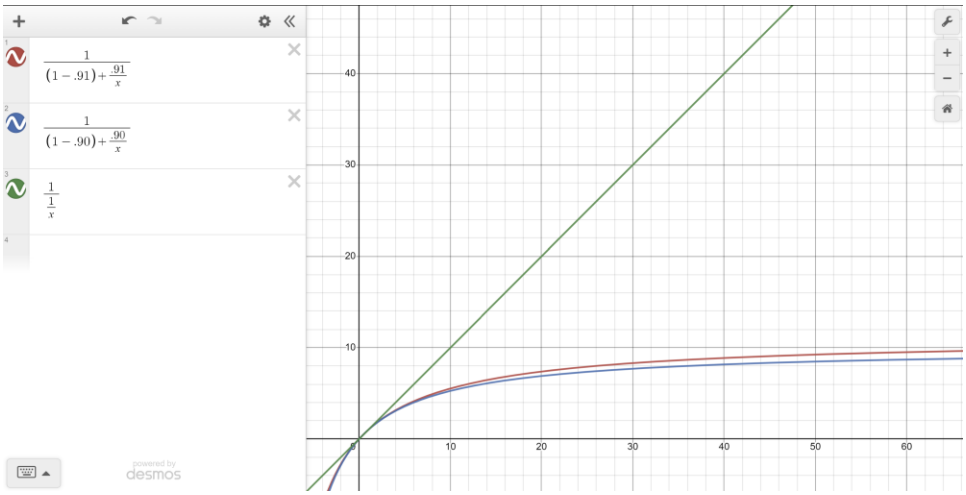
$$\frac{1}{(1-.90)+\frac{.90}{14}} = 6.086956522$$

There would be no speed up for the single threaded version meaning its value would just be 1

So comparing these to the single threaded program the Pthread is 6.45x faster than single

OpenMP is 6.08x faster than single threaded

Now if we compare both of the Amdahl's equations of the Pthread and OpenMP to the ideal linear speed-up it would look like this:


1
$\frac{1}{(1-.91)+\frac{.91}{x}}$
2
$\frac{1}{(1-.90)+\frac{.90}{x}}$
3
$\frac{1}{\frac{1}{x}}$
4
powered by
desmos
40
30
20
10
0
10
20
30
40
50
60