Multithreaded Image Compression using DFT

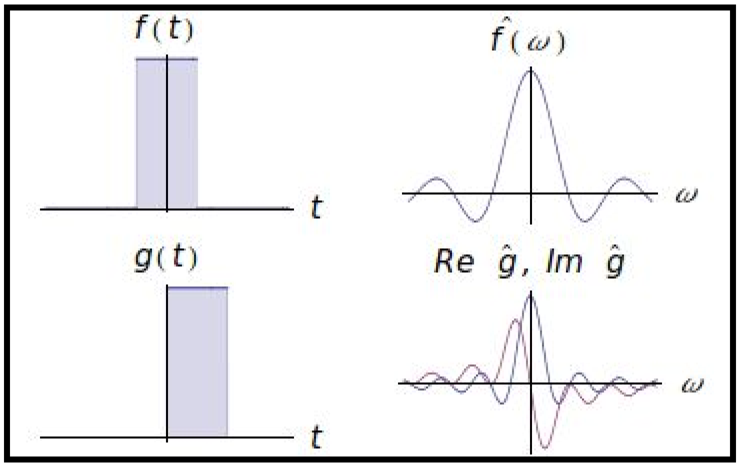
COM301T: Operating Systems Project

Group Number: 26
Nehemiae George Roy (EVD18I018)

DFT Theory

Image compression is a widely used technique in many image file formats to efficiently store images, while retaining their observable visual properties. Many lossy file formats like jpeg, jpg, etc. are based on this principle, and apply many image transformation techniques to remove redundant information, or to remove information that is practically useless - since they contribute nothing to the observable image. This latter technique is grounded in analysing the frequency domain version of the image, using Discrete Fourier Transforms (DFT).

The Fourier Transform is an important image processing tool that allows the image to be represented in the frequency domain. The fundamental basis of the Continuous Fourier Transform is that any continuous periodic signal can be represented by a combination of multiple sinusoids. This is especially noticeable for square wave pulses, where multiple sinc waves can be seen in its corresponding frequency domain. The Discrete Fourier Transform is similar to this approach, except that now we work now on discrete form of data, and hence we look only at a set of samples that qualify to describe the image



Continuous Fourier Transform of a square pulse

The Discrete Fourier Transform is given by,

$$F(k,l) = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} f(i,j)e^{-i2\pi(\frac{ki}{N}+\frac{lj}{N})}$$

where $f(a,b)$ is the image of size $N \times N$ in the spatial domain
$F(k,l)$ is the image of size $N \times N$ in frequency domain
The exponential is also called the twiddle factor.

The Inverse Discrete Fourier Transform is given by,

$$f(a,b) = \frac{1}{N^2}\sum_{k=0}^{N-1}\sum_{l=0}^{N-1} F(k,l)e^{i2\pi(\frac{ka}{N}+\frac{lb}{N})}$$

The important things to note are that the value of each point $F(k,l)$ is obtained by multiplying the spatial image with the corresponding twiddle factor and summing the result. The twiddle factors are sine and cosine waves with increasing frequencies, i.e. $F(0,0)$ represents the DC component of the image which corresponds to the average brightness and $F(N-1, N-1)$ represents the highest frequency.

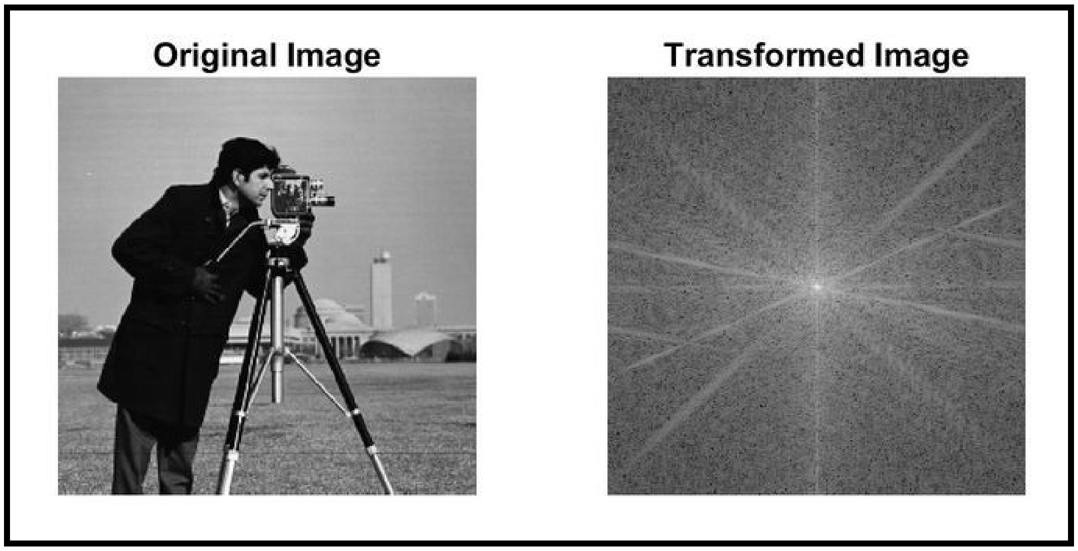


Image in spatial and frequency domain (The center of the frequency domain is $F(0,0)$)

The fundamental basis of this project relies on this fact. An image is broken down into multiple sub-images (preferably squares), and the DFT coefficients of the pixels are calculated. Those coefficients that fall below a tolerance value are dropped - i.e. set to zero - and the inverse DFT of the sub-image is taken and replaces the old sub-image. In effect, the new image contains the pixel values corresponding to the frequency components above the set threshold. Thus, the new image contains pixel values that are more averaged out, and hence this image will have a lower file size than the original.

Another interesting fact is that the DFT can be calculated using FFT, which can be parallelized in software by virtue of the procedure. However, the intention of this project is to not perform FFT by multithreading, and hence this is implemented using existing libraries.

Aim of Project

1. To build an image compressor executable that accepts the input and output file name, along with a compression factor between 1 to 10.
2. To implement the aforementioned program using multithreading, where:
    a. Thread 0: Copying of pixels into a new image file.
    b. Thread 1 to N: Accessing 16x16 sub-images and calculating the DFT of these pixels (by multithreaded FFT), and updating these sub-images.

   The value N will also be passed to the executable.
3. To compare this multithreaded version, with one without using any threads and analyse the speedup.
4. To observe the varying degrees of compression with the specified compression factor.

The entire program was implemented using C with GCC compiler, along with the FFTW library.

The Code - `img_prj_threaded.c`

We will now look at a brief description of the code. The user enters the number of threads the program must use, followed by the compression factor, and followed by the input and output file names. This is shown in the syntax:

```
./a.out <num_threads> <compress_factor [1-10]> <input_img.jpg>
<output_img.jpg>
```

The procedure to use the FFTW library is very lengthy, but in short, we initialize the threads required for this using `fftw_init_threads()` and `fftw_plan_with_nthreads(3)` and then create a plan for the FFT that must be calculated for the sub-images. The worst case scenario for any sized image, is where there can be 4 possible DFT calculations, shown also in the below image:

1. (Red) DFT of sub-images of size 16x16.
2. (Yellow) DFT of sub-images along the extreme column, where the image width is not divisible by 16 and hence. The height of these sub-images is 16.
3. (Orange) DFT of sub-images along the extreme row, where the image height is not divisible by 16 and hence. The width of these sub-images is 16.
4. (White) DFT of sub-image at the bottom right corner, where both the image height and image width is not divisible by 16.

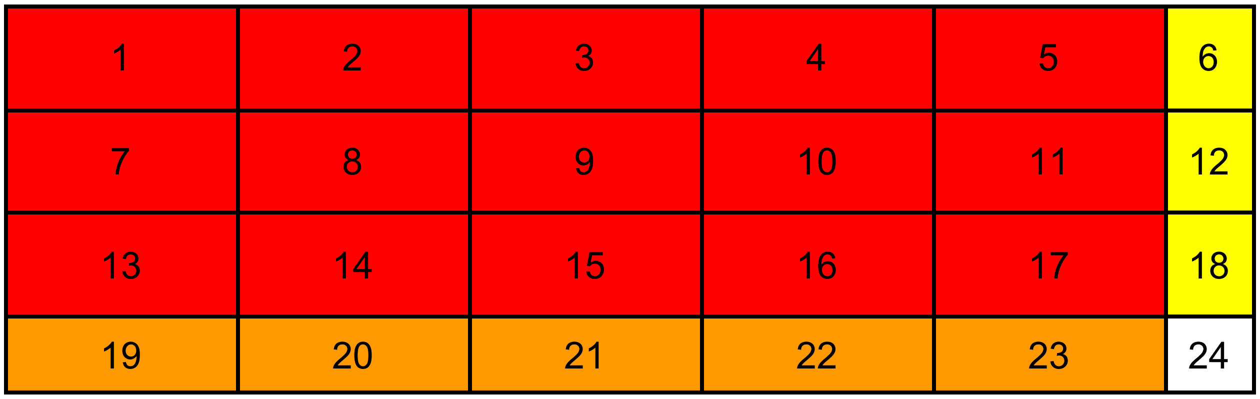


Image that is broken into sub-images

Thus, we set up the DFT calculation for these 4 mentioned cases and their corresponding inverse DFTs, using `fftw_plan_dft_2d()`. Inverse DFT is obtained by specifying `FFTW_BACKWARD`.

As mentioned before we will use 1 thread for copying the pixels between the old image file and the new image file, and N threads for the DFT calculations. Hence, let's look at the `copy_runner()`. Two pointers `p` and `pg` run across the input and output image respectively, and copy the pixel values. Also we must remember that some images may have 3 channels if they are RGB or one channel if they are grayscale, and hence the pointers need to jump in steps of `img_channels`. The most part here, is accessing the `all_sq` array through the `all_sq_ptr` pointer. This 2 dimensional array has as many elements as the required number of sub-images in an image (same as the previous image shown above). It holds three possible values:

1. 0 - The corresponding sub-image has not yet been copied from the original image by `copy_runner()`.
2. 1 - The pixel values have been copied and calculations of DFT and replacement of pixels is to be done.
3. 2 - Calculations are completed by `calc_runner()`.

Clearly, this array must be shared between N+1 threads and this becomes a resource that could lead to potential race conditions. To avoid this we make use of a single mutex, named `mutex`, which any thread must gain the lock to read or modify the `all_sq` array. Thus, when a row of sub-images has been copied, the `copy_runner()` acquires the mutex and then sets the corresponding elements of the row to 1.

Next, let's move to the `calc_runner()`. Here, we are set into an infinite loop, where we first attempt to acquire the mutex, and then scan through the `all_sq` array, searching for any sub-image that is set to 1. And if we find such an element, we set it to 2 and then break out of the loop. Also we keep checking if any element is 0, to ensure that the thread must not die but continue in the loop. When we exit the loop, if this condition is not met, meaning that all the squares are currently completed or being worked on by the other threads, then the thread exits the infinite loop and dies gracefully. The variable `req_sq` holds the square index that we must now work on, and the first thing we do is to set the required sizes of the rows and columns of the sub-image based on the 4 possibilities mentioned. Then the complex pointer `input` is set to a memory area of appropriate size. Again we must run our loops as many times as there are image channels, and the `p` pointer will point to the starting location of the original image file plus the total rows to reach the row of the sub-image, plus the total number of columns of size of the sub-image to reach the starting point of our sub-image. The similar logic is used for the remaining pointer references. We then copy the contents from the original image into the created memory area. Then based on the size of the sub-image, we calculate its corresponding DFT using `fftw_execute_dft()`, calling the corresponding planner. Then, we run across the sub-image and set every element that is lesser than the tolerance value to 0, and then re-convert the DFT to the required pixel values, using the appropriate planners. In the next set of loops, the pointer `pg` runs across the sub-image in the new memory location, and the new pixel values are copied to the destination image, after dividing by the number of elements in the sub-image. Once, this is done, we free the memory area pointed to by `input`.

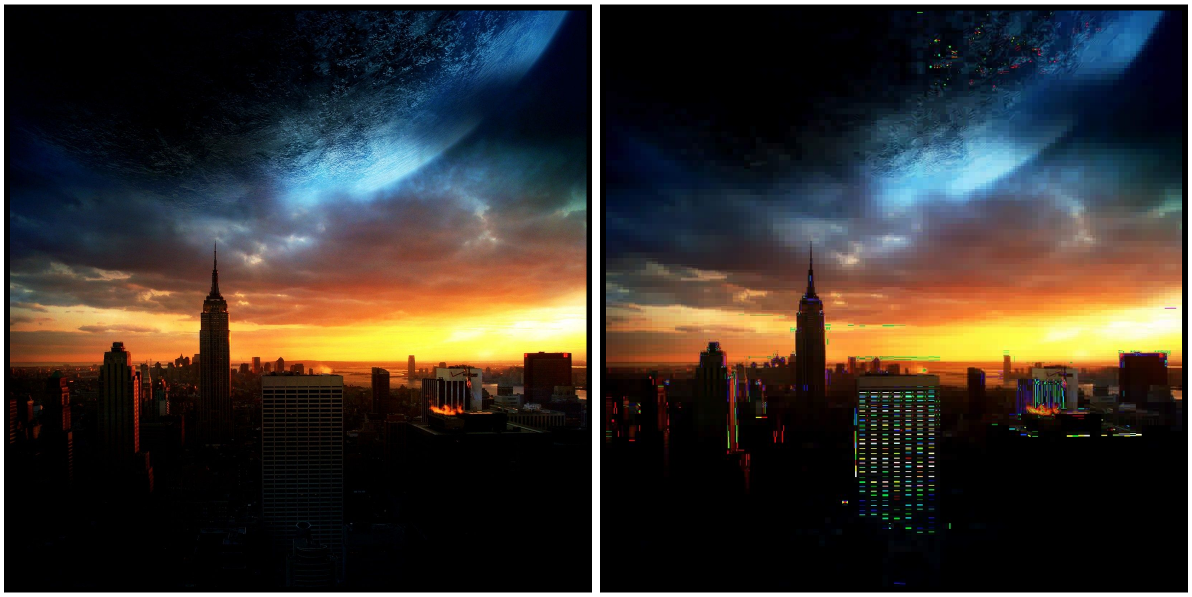
Back in `main()`, we are waiting for all the threads to complete and then we write the new file to the current directory, and free up the memory locations of the image files in memory.

Code Test-drive

The following images show some unique cases, which prove that the code works for all possible combinations.

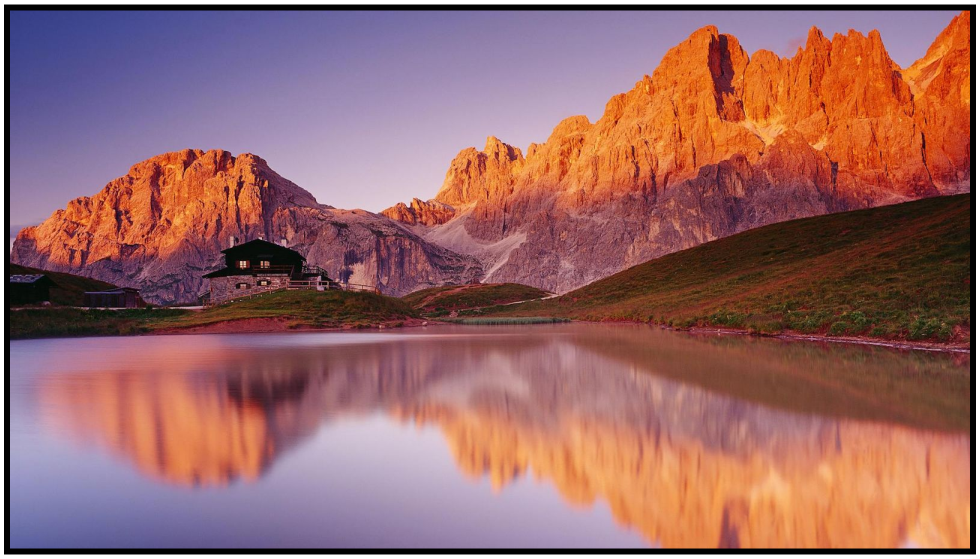
The call from the terminal is shown below.

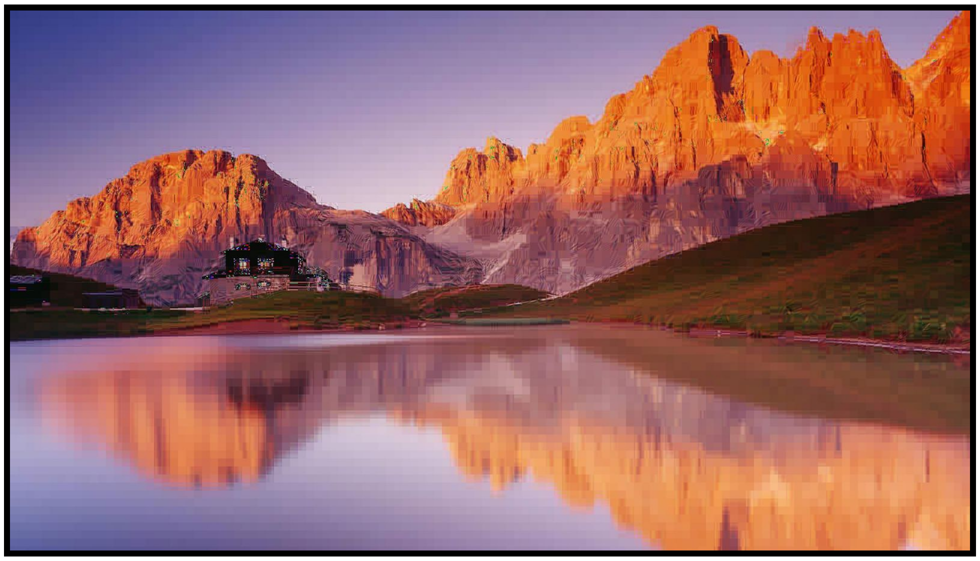
Now, let’s look at some image input combinations. For a square image, with 3 channels, with dimensions divisible by 16.



Original File - city.jpg, 1600x1600, 244.4 KB; New File - city_new.jpg, 1600x1600, Compression Factor 5, 102.4 KB

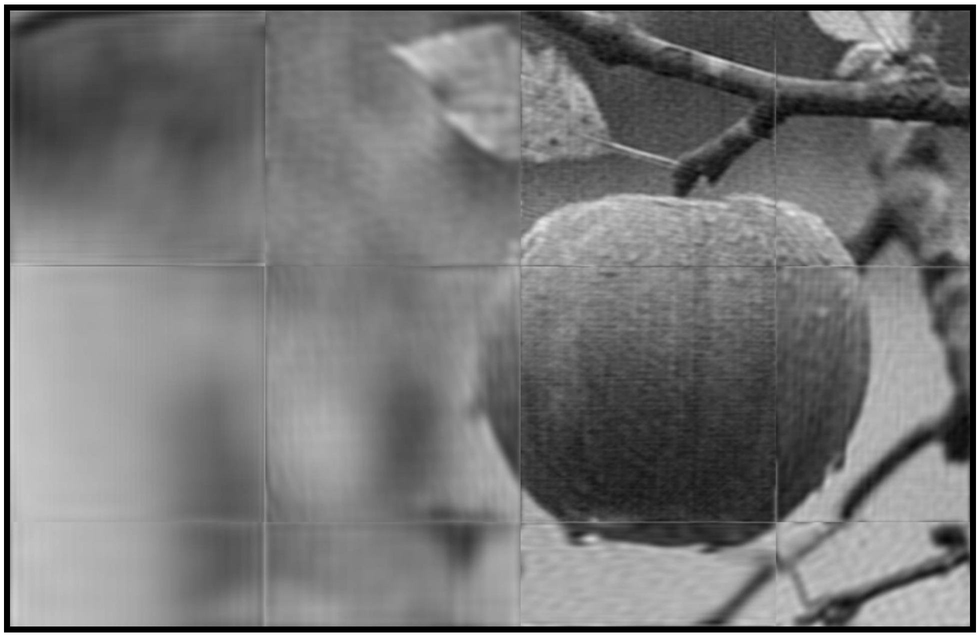
Next, let's look at an image whose dimensions are not divisible by 16, and has 3 channels.


Original File - mountains.jpg, 1920x1080, 288.8 KB


New File - mountains_new.jpg, 1920x1080, Compression Factor 5, 123.5 KB

Finally, we can also give a grayscale image as input. By virtue of the generic nature of the code, the sub-image size 16x16 can be changed by editing the code (two initial `#define` alone) appropriately. Hence, for this image, the sub-image size is 500x500 and the tolerance is scaled up.

Original File - gray.jpeg, 1880x1204, 153.4 KB



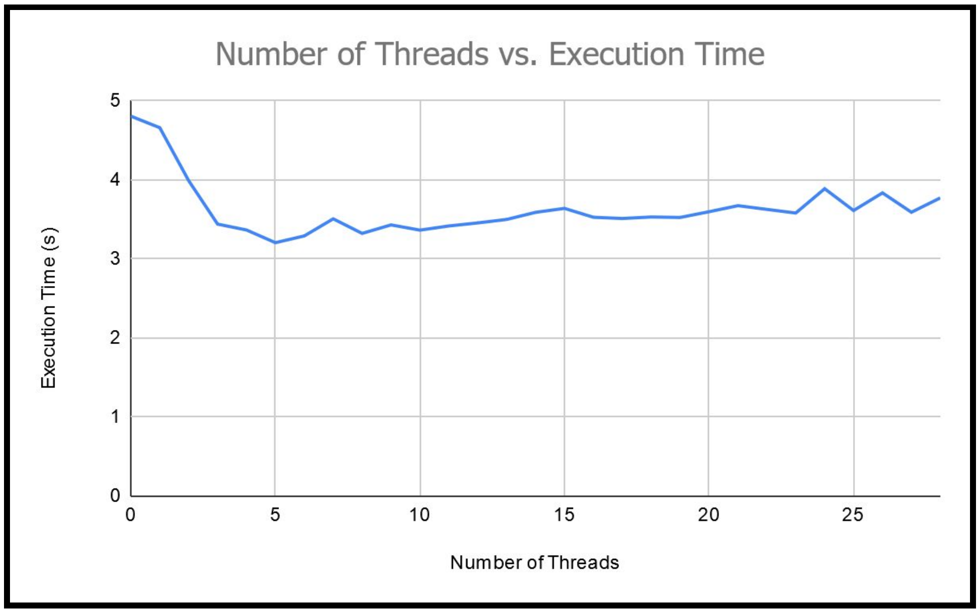
New File - gray_new.jpeg, 1880x1204, Tolerance 50000, 97.7 KB

It is clear that such an implementation is not visually appealing, and hence the sub-image size of 16x16 will be carried onward for the following discussions.

Comparison with no Multithreading

The purpose of this section is to only analyse the performance gains of multithreading as per the implemented code in `img_prj_threaded.c`. Hence, the multithreading of the FFTW library carries forward in this discussion. A separate file called `img_prj_direct.c` can be found with no multithreading implementation. Note this must be confused with passing `num_threads` as 1, as even in this case there will be a total of N+1 - i.e 2 threads - one copying the contents and the other performing `calc_runner()`.

The testing of the times were done on a freshly booted system, with no other applications open, to ensure that the program would have complete access to all the 4 processor cores of the machine. These are summarized in the graph below. The point corresponding to threads as zero, is the code without multithreading.



Graph of number of threads vs. execution time

From this chart we can see a steep decline in the execution time until when threads are equal to 5. After this there is a gradual increase in the execution time, which can be explained by the fact that the system is limited by 4 cores and hence the effective parallelism can be achieved at max by 4 threads. In fact, once the threads cross the total number of sub-images possible for an image, there will be an increase in the execution time due to the mutex gain by these threads - only to realize that all the sub-squares are set to 2, and then they die. Apart from this, for the given system, we would not see much difference after 4 threads as at any rate only 4 threads can be done at a time.

At best, the speed up of the code that can be parallelized is given by,

$$Speed\ Up\ Enhanced\ =\ \frac{Time\ of\ the\ Section\ of\ Code\ on\ 1\ Processor}{Time\ of\ the\ Section\ of\ Code\ on\ 4\ Processors} = \frac{4.852786}{3.206620} = 1.5133$$

Once again we must note that this speed up only refers to the copying and calculating sections of code. The remaining sections are serial, and will limit the possible speed up, even with addition of more processors.

Comparison of Size Compression

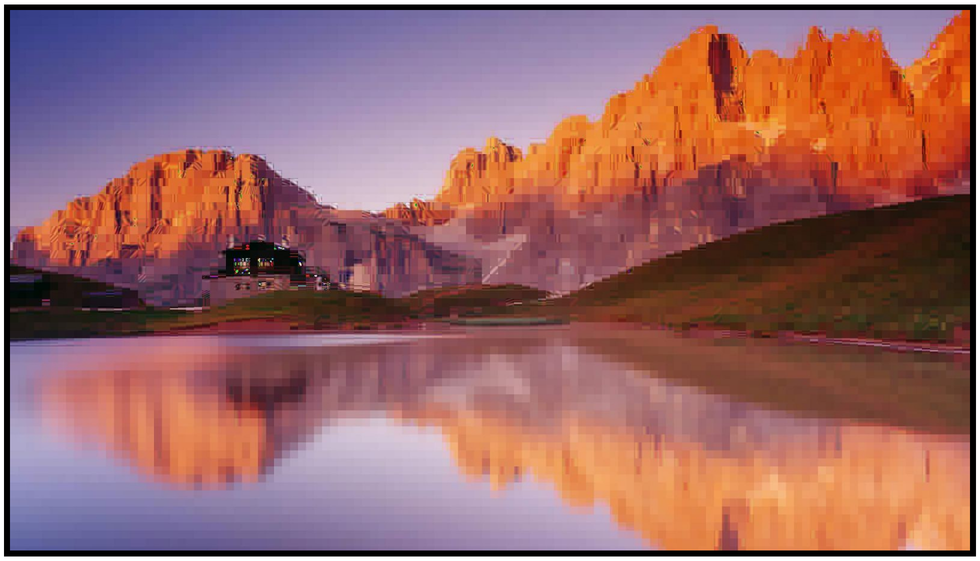
The compression parameter between 1 and 10, that is passed to the program, is scaled up to tolerance level of 0 to 2250. Thus, for a compression parameter of 1, we would get exactly the same input image. The given range has been decided by trial and error on the worst possible and yet acceptable image with the least file size. But, truly any positive integer will give some result.

The following table gives the compression results of `mountains.jpg` (size is 288.8 kB), for various compression factor values.

Compression Factor	Tolerance Value	Final File Size	Size Reduction
2	250	229.7	20%
3	500	186.7	35%
4	750	150.4	47%
5	1000	123.5	57%

6	1250	105.1	63%
7	1500	92.2	68%
8	1750	82.1	71%
9	2000	75.3	73%
10	2250	70.0	75%

The image `mountains_jpg`, with the least file size and worst quality - i.e. compression factor of 10, is shown below.



New File - mountains_new_worst.jpg, 1920x1080, Compression Factor 10, 70.0 KB

Conclusion

Thus, we have seen that the effects of multithreading in this example are limited to the hardware resources. However, keeping Amdhal's law in view, the amount of resources can provide no more speed up in this program, than the amount of serial code that exists. This includes all the initializations and reading and writing of the images. Further, this application can be used to compress images into different sizes, with decreasing levels of quality.