

## Shared Memory Thread Performance

In this assignment, you will try to maximize parallel performance on a four-core machine using Posix threads. The assignment is relatively simple, but should offer enough complexity for you to find interesting problems to reason about with regard to finding parallelism, sharing data between cores, load balancing, and other issues involved in getting several processors to cooperate effectively with the shared memory thread approach to parallelism.

You should work in teams of two to four. Part of the grade will involve relative performance on test images (incorrect results will obtain low performance scores, regardless of speed), but I encourage you to implement a sequential version first, share test image results and test images, etc. The class web board may be the best forum for such exchanges.

Your program will perform the following task. Given command-line arguments specifying a (color) JPEG image file name, a threshold value, and a segment size, your program will read in the named JPEG image file, run Sobel edge detection on the image (which involves convolving each of the RGB planes of the image with two 3x3 filters to calculate a color gradient at each pixel), identify edge pixels by comparing the gradient at each pixel with the threshold, identify connected non-edge segments of the image, and output original image pixels corresponding to each segment that is at least as large as the size given on the command line as a separate image. Both the edge detection and the segment identification offer opportunities for parallel execution.

A starting package has been available to you for a couple of days now. Download the starting code from the class web page. The current version was developed on Cygwin and thus has some warnings using `gcc` on the machines on which you'll develop your code. I will try to fix them and post a new version, but you can fix them if you'd prefer.

The given code simply reads a JPEG file named as a command-line argument, calls an empty `operate` function, and writes the results back as a new JPEG file called `new.jpg`. You can try to play with the `libjpeg` call sequence, decompression style, etc., if you think that it will help your performance. Do not, however, change the compression scheme from the default when writing files. Also, I'm not sure that the library is re-entrant, so be careful if you want to parallelize output.

### Specifics

Here are a few details on the things that you'll need to do:

- The command-line argument order should be: `<filename> <threshold> <segment size>`. The threshold is a floating-point number to be compared with the gradient of each pixel (described in the next section) to determine whether or not it is an edge pixel. The segment size is in pixels. Only segments containing *that many pixels or more* should be used to form output images.
- When forming segments, only connect non-edge pixels that are adjacent vertically or horizontally (not diagonally) in the image. Do not include any edge pixels, and do not fill in segments. You might find complex topologies in some images, e.g., a donut shape.
- For each segment of the requested size in pixels, write a new JPEG image file containing the original image data for the segment along with black pixels (RGB=0x000000) for all pixels outside the segment. Name the output files `output1.jpg`, `output2.jpg`, etc.. Do not worry about the order of segments found; pick something.
- Again, do NOT change from the default compression parameters when writing output files.
- Pick your favorite algorithm for finding the segments.
- Be careful with pixel data; signed/unsigned issues can crop up easily.

## Sobel Edge Detection

For a monochromatic image, Sobel edge detection estimates the intensity gradient at each pixel by convolving the surrounding pixels with a small matrix. The gradient for each dimension is calculated separately using these matrices:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \qquad G_y = G_x^T = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

The gradient in a monochromatic image is then calculated as the square root of the sum of the squares of the gradients in each dimension. Do **not** use an approximation here.

For a color image, you must calculate gradients in both dimensions separately for each of the three color planes (red, green, and blue). Sum the six squared gradients rather than two, and take the square root of that sum to find the gradient at a given pixel.

Some of you may not have seen convolution before. Essentially, think of mapping the matrix above on to the image with the middle element centered on a pixel for which the gradient is to be calculated. Multiply the matrix values by the corresponding pixel values (0 to 255), then sum all nine numbers to find the convolution at that point. Feel free to find ways to do less work.

At the boundaries, the matrix will not have corresponding image data. In such cases, use the central pixel value in place of all missing pixels (there will be five missing pixels in each corner of an image).

## Documentation

Some specification information for using `libjpeg` can be found at  
[refspecs.freestandards.org/LSB\\_3.1.0/LSB-Desktop-generic/toclibjpeg.html](http://refspecs.freestandards.org/LSB_3.1.0/LSB-Desktop-generic/toclibjpeg.html)  
and documentation for use can be found at  
[apodeline.free.fr/DOC/libjpeg/libjpeg-2.html](http://apodeline.free.fr/DOC/libjpeg/libjpeg-2.html)

You can find a tutorial on edge detection, including Sobel, at  
[www.pages.drexel.edu/~weg22/edge.html](http://www.pages.drexel.edu/~weg22/edge.html)

There are some better methods; for example, see  
[www.pages.drexel.edu/~weg22/can\\_tut.html](http://www.pages.drexel.edu/~weg22/can_tut.html)

## Handin

Put all files, including a Makefile updated as necessary to support your code, into one directory, pack it up, and mail it to me. As with Lab 2, you'll lose some points for compiler warnings, and most points for failing to compile. I may have some specific test and expected output for you at some point, but hopefully all of the teams can contribute useful tests via the web board.

(I'm a bit worried again about this problem having too little computation, so we may need to use a more sophisticated/complex image processing algorithm instead.)