Program 4 Design Document and Performance analysis
Parallel - CSC 410/510

Ian Carlson
Matthew Richard
Christopher Smith

Overview

Our project involved implementing a suite of image processing functions in parallel with shared memory using OpenMP. In order to display the images, and provide a basic GUI, we used Qt. We implemented three classes of image processing functions.

Point Processes

Point processes are very efficient and easy to compute. They are called such because each pixel in the output image is a function only of itself in the input image. These functions can typically be computed in linear time O(n) where n is the number of pixels in an image.

In other words,

$$f'(x,y) = h(f(x,y)).$$

Point processes are easy to parallelize because there is basically no data dependency between pixels, and no communication is necessary between processes. We implemented the following point processes.

- Grayscale
- Brighten
- Darken
- Noise
- Binary Threshold
- Negate
- Gamma
- Enhance/Reduce Contrast
- Posterize

Convolution/Mask Processes

Image processing techniques that involve a convolution with a mask are a bit more complex. For a given output pixel, a mask of size k is convolved with the k*k surrounding pixels. Masks are typically square and have odd dimensions so the pixel in question can be centered in the mask. Convolutions can typically be calculated in O(n*k*k) time, where n is the number of pixels in the input image, and k is the length and width of the mask.

We implemented the following convolution based processes.

- Smooth
- Gradient
- Laplacian
- Sharpen

- Emboss
- Gaussian

Fast Fourier Transform

While not its own class of image processing techniques, we also implemented the 2D FFT. Each pixel in the output image of an FFT is dependant on the entire input image. This leads to potential dependencies. The calculation time is also much greater than in point or convolution processes. To increase efficiency, we separated the Fast Fourier Transform into 4 parallelized sections.

Foster's Design Method

Partitioning

Images lend themselves well to data parallelism. The smallest task in most image manipulations is the calculation of a single output pixel, dependant on its previous value or the values of neighboring pixels. Each operation is independent because it depends only on the previous value of the pixel and neighboring pixels (or the whole image in the case of the FFT), which are known at the start.

Most algorithms on images can be partitioned on the rows. The Gaussian Filter, brighten/darken, sharpen/smooth, gradient, increase/decrease contrast, introducing noise, edge detection, emboss, binary threshold, posterize, negate and many more algorithms can be parallelised this way.

The Fast Fourier Transform (FFT) has 4 steps of the algorithm parallelised. The FFT is implemented by doing four steps of parallelising:

1. A FFT for each row is calculated. The parallel partitioning here was done by row.
2. A FFT for each column is calculated based on the results of step one. This step was parallelized by column, which is non-ideal, but better than sequential calculation.
3. The magnitude over the real and complex parts is computed. This step is partitioned over rows.
4. The image is normalized and written back to the Qt image format. The top left quadrant is switched with the bottom right, and the top right quadrant is switched with the bottom left so that the 0,0 is the center of the image. This last step is partitioned over rows.

Communication

Interprocess communication is mostly unnecessary. All processes need access to most of the initial and modified image, so those can be shared. With the exception of the FFT, there are no dependencies between tasks, so no further communication is required. For the FFT, each step is dependant on the result of the previous step, but no concurrent communication is required.

Agglomeration

We have grouped the data by rows, to conform with the way data is stored and to help prevent thrashing in the cache with the exception of the FFT, which requires the FFT to be done on the columns of the image.

Mapping

Mapping will be done by block allocation of rows of the image. Each process/thread will get an equal number of rows plus or minus one. Also columns will be done in the same way for the FFT.

Deliverables

We have implemented the following list of algorithms using OpenMP and c++ with Qt. All algorithms just use a static value for modifying pixel values (ex. brighten and darken add or subtract a value of 20 from each pixel value)

- Gaussian Filter
- FFT
- Brighten
- Darken
- Sharpen
- Smooth
- Gradient
- Increase/Decrease Contrast
- Introducing Noise (salt/pepper, etc.)
- Edge detection
- Emboss
- Binary Threshold
- Posterize
- Negate

Performance Analysis

The performance analysis was calculated using three of the above algorithms Gamma, Gaussian smoothing, and the FFT.

Gamma

Liechtenstein (512x512)

Processes	Time	Speedup	Efficiency
1	0.0726403	-	1
2	0.0505233	1.43775842	0.7188792102
4	0.0432803	1.67836868	0.4195921701
8	0.0382515	1.899018339	0.2373772924

raindrop (2592x2538)

Processes	Time	Speedup	Efficiency
1	2.3246	-	1
2	1.26692	1.834843558	0.9174217788
4	0.721734	3.220854221	0.8052135551
8	0.708012	3.283277685	0.4104097106

brick_wall (5760x3840)

Processes	Time	Speedup	Efficiency
1	5.98182	-	1
2	4.38484	1.364204851	0.6821024256
4	2.81753	2.123072336	0.5307680841
8	2.68727	2.225983991	0.2782479989

Point processes are already efficient with their image operations, however as the number of threads increase the efficiency goes down a lot. This is mostly due the fact that the overhead of creating processes and assigning each one its own data takes the most time. Aksi as the image size increase the efficiency dropped a lot so point processes are not scalable enough to be more efficient. Two to four threads seems to be a good balance on medium to large image sizes, while sequential and two threads are good for smaller images.

Gaussian Smoothing

Liechtenstein (512x512)

Processes	Time	Speedup	Efficiency
1	0.471	-	1
2	0.269	1.751	0.875
4	0.233	2.021	0.505
8	0.145	3.248	0.406

raindrop (2592x2538)

Processes	Time	Speedup	Efficiency
1	10.178	-	1
2	5.223	1.949	0.974
4	2.783	3.657	0.914
8	2.341	4.348	0.543

brick_wall (5760x3840)

Processes	Time	Speedup	Efficiency
1	32.978	-	1
2	17.045	1.935	0.967
4	14.116	2.336	0.584
8	7.473	4.413	0.552

The Gaussian smoothing was very efficient when moving from one to two cores, but when moving to four and eight cores the efficiency went down drastically. Although, the solution still seems scalable since the efficient increased as the image size increased.

Fast Fourier Transform

Lena(256x256)

Processes	Time	Speedup	Efficiency
1	4.5197	-	1
2	2.7074	1.669387604	0.8346938022
4	1.7277	2.6160213	0.654005325
8	0.9327	4.845823952	0.605727994

Liechtenstein(512x512)

Processes	Time	Speedup	Efficiency
1	35.1832	-	1
2	18.2923	1.92338853	0.9616942648
4	10.6577	3.301200071	0.8253000178
8	7.0045	5.022942394	0.6278677993

Galaxy(1024x1024)

Processes	Time	Speedup	Efficiency
1	330.651	-	1
2	168.102	1.966966485	0.9834832423
4	92.8038	3.562903674	0.8907259186
8	64.3373	5.139335968	0.6424169961

The Fast Fourier Transform could have been more efficiently implemented - comparing against OpenCV's sequential implementation was quite frankly depressing. Optimizations aside, the FFT algorithm did respond very well to parallelization, showing near-linear speedups per physical core. The efficiency improved on larger image sizes, showing that the FFT algorithm is at least weakly scalable.