Circle detection in images using Hough Transformations

By Ungureanu Ionut-Vladimir (Group 937)

Goal

Taking as an input an image and after parameter tweaking, sucesfully detect appearances of round objects.

Requirement

Each project will have 2 implementations: one with "regular" threads or tasks/futures, and one distributed (possibly, but not required, using MPI). A third implementation, using OpenCL or CUDA, can be made for a bonus.

Computer Specification

- CPU: Intel Core i7-8550U (8M Cache, up to 4.00 GHz)
- RAM: 16 GB

Algorithms

We will apply a Canny Edge Detection algorithm in order to prepare the image for detection.

Steps of the canny edge detection algorithm:

1. Turn the image grayscale (as it's harder to work with colors)
2. Blur the image
3. Compute a gradient consisting of neighboring pixels. By applying this formula, we make the process easier.
4. Filter Non maximal values i.e values that are not necessarily relevant to the detectiong algorithm
5. Filter edges (otherwise they turn into accumulation points as describe in the Hough algorithm)

Next we apply the Hough algorithm. Here are the steps: 1. Iterate through all edges and consider each edge with each possible radius 2. Create accumulator and for each case of step 1, vote for the best candidate in the following way:

i. For every theta point in a 360 degree range(360 points)

ii. Compute the polar coordinate of the center

iii. Increment the accumulator

4. For each accumulator that passed a certain treshold, we check if drawing a circle around that particular point is valid in the current xOy system (image bounds)
5. Draw the circles on the image using a library

Thread implementation

We use a multitude of subtasks for anything that is worth parallelizing over (i.e. everything that costs more to compute than to actually create threads)

As such, we create an Executor service. Tested with 4 and 8 threads (results are similar because of CPU specification so i will only include one)

Blurring task - For the blurring part of the canny edge algorithm, for every x point in the xOy system, we create a thread. These threads will act independently on a ConcurrentHashMap to compute the blurred variaant

Gradient sub task - Will work in a similar manner to the blurring task with the exception that this will compute 2 maps one being the gradient result and the other being its direction

Filtering and Edge Detection - 2 distinct tasks which work in similar manners, but within specific bounds in order not to exceed the limit of the image

Distributed - MPI (MPJ Express)

For the distributed version I have chosen to use a variant of MPI for Java and that is MPJ for the sole purpose that Java has been consistently the programming language for this semester and MPJ seems to be the only popular wrapper for OpenMPI

The approach here is rather complex compared to the simple Thread implementation

The master process will use a HoughTransformation class where at each step that is worth parallelizing, it will send to its slaves the required information for them to compute the computation

Example:

```
Create array of objects
For every x coordinate:
    Create object of type Task and add it to object array
For every rank:
    Send the Task
For every rank:
    Await results from the slave process
```

The slave however will implement an approach based on constantly receiving commands from master.

```
Await command from master
Check type of command
Run command according to each case
Send the result back to master and return to step 1 (Except if command received is terminati
```

As such, the master only sends the command and collects the results, the other processes being the ones that do all the work.

Unsurprisingly, this approach takes slightly longer for the computer to do because we run it locally using processes. This might be more efficient given a truly parallel system (multiple computers)

Performance Tests

I have extensively used a standard image in order to test for the use of parallelization in this project.

Here werre the results

Tasks Parallelized	Time
0	48 seconds
1	35 seconds
2	31 seconds
3	29 seconds
4	25 seconds

Reasons i chose to do the previous thing:

1. Check the usefulness of parallelization in a real life situation
2. The initial implementation was sequential (for the sole reason that the literature provides such indications (i.e papers referring to the subject specifically usually don't think of parallelism as much as the algorithm itself))

Now the following are tests for 2 photos on all 2 systems.

Reasoning: We only need to find a correlation between 2 sizes and 2 measurements of time to extrapolate the required time for a different size

NOTE: The run time tests may be misleading for the simple purpose that parameter choice is the most important (after image size of course) parameter for the runtime of the test. I have personally tested A LOT of values in order to get results for the test images for the circle detection and reasonable time.

Algorithm	640x417	160x104
Threads	24 seconds	2 seconds
Distributed MPI	89 seconds	2 seconds

#Results

The results were surprising at first as it seems that image size causes runtime to grow exponentially. In this section i will explain why this happens (and why it wouldn't happen in a line detection Hough Transformation)

One should consider the fact that for each point we have to compute a 360 simulation of the possible circle. Accumulating this takes thime and slowness is to be expected.

Comparison with line detection:

Algorithm	Required operations
Line detection of 20 consecutive pixels	Detect 20 consecutive pixels of same color that follow a given equation ax+b=y
Circle detection of 20 accumulated pixels	Compute 360 circles that follow (x-xc)ˆ2 + (y-yc)ˆ2 = r for every considered radius and check where they all intersect. The result is a POSSIBLE circle (we don't know for certain if it's within image bounds

Conclusion

Altough my choice of presentation has been more or less complicated by which Hough Transformation i have chosen, i was able to draw the following conclusions:

1. Exponential time will be required in image processing if the operations require multiple filters
2. Parallelization yield great results when it's actually worth to create the parallel resource (50% better time on a medium image and most likely more on greater images as the effect will only grow)