**Corner Detection**

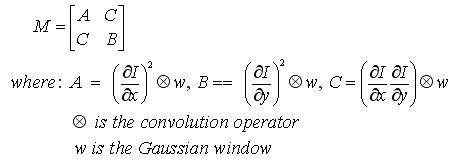
We used Harris/Plessey algorithm for implementing this task. This algorithm was chosen because:

* It is better than traditional Moravec algorithm, in that it uses image gradient to identify correct corners.
* It correctly identifies corners along diagonal edges also unlike Moravec and is the best and most widely used algorithm.

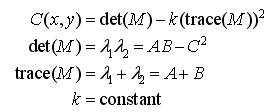
It calculates the intensity variations at each pixel. The result is approximated using image gradient. It overcomes limitations of Moravec algorithm by using Gaussian window to smooth the area around the corners. In the algorithm, partial differentiations along row and column are taken and the result is convolved with the Gaussian window to obtain an auto-correlation matrix for every pixel. The cornerness map is constructed using the values of A, B and C for every pixel which helps determine the corners.

The algorithm in short can be described in the following steps:

    1.      For each pixel (x, y) in the image calculate the autocorrelation matrix M:



    2.      Construct the cornerness map by calculating the cornerness measure C(x, y) for each pixel (x, y):



    3.      Threshold the interest map by setting all C(x, y) below a threshold T to zero.

    4.      Perform non-maximal suppression to find local maxima.

All non-zero points remaining in the cornerness map are corners.

We have implemented the above algorithm in C and then parallelized using OpenMP parallel framework. Since it is a task of data decomposition, OpenMP seemed the best suited choice. Parallelizing the code was simple as it only included inserting pragma statements at the ‘for’ loops. The experiments were carried using several images ranging from 512x512 to 2048x2048. Presented below are the results of implementing on the following:

* Small Image - 1024x1024 8bpp bmp image
* Big Image - 2048x2048 8bpp bmp image

The original image used was 1024x1024 8bpp grayscale image:

The image obtained after applying corner detection. The corners are highlighted as red dots. 

Runs carried out on Small Image (1024x1024) and results:

|  |  |  |
| --- | --- | --- |
| Runs | Execution times (secs) | CPU utilization (%) |
| Sequential | 3.89 | 99 |
| 2 Threads | 3.83 | 113 |
| 4 Threads | 3.71 | 131 |
| 8 Threads | 3.66 | 165 |
| 16 Threads | 3.67 | 235 |

The execution times in seconds are plotted in the graph below. The first bar represents the execution time for sequential execution. After parallelizing the code, the times are seen to reduce.

Also, as expected the CPU utilization is seen to increase as the number of threads increases. Thus more and more threads seem to be kept busy.

This is a good indication that the CPU is kept busy with lot of work rather than spending idle time. The CPU utilization measurements are in %.

Similar graphs were plotted for Big Image and similar results were obtained.

Runs carried out on Big Image (2048x2048) and results:

|  |  |  |
| --- | --- | --- |
| Runs | Execution times(secs) | CPU utilization (%) |
| Sequential | 15.46 | 99 |
| 2 Threads | 14.83 | 108 |
| 4 Threads | 14.76 | 125 |
| 8 Threads | 14.54 | 150 |
| 16 Threads | 14.48 | 200 |

Apart from this, parallelizing was also tested using dynamic scheduling. For this, again UNIX time was used to get the execution times. Shown below is the comparison of execution times (secs) for static and dynamic scheduling for a 1024x1024 image by keeping the chunk size fixed to 4 and testing for 2, 4, 8 and 16 threads.

|  |  |  |
| --- | --- | --- |
| Runs | Static Schedule | Dynamic Schedule |
| 2 Threads | 3.83 | 3.73 |
| 4 Threads | 3.71 | 3.61 |
| 8 Threads | 3.66 | 3.55 |
| 16 Threads | 3.67 | 3.68 |

The threads are able to dynamically schedule work better with 2, 4, and 8 threads. But for 16 threads the time is similar, in fact more for dynamic because there is lot of overhead in creating the threads while work can be easily done with lesser number of threads.