CS133 Project Write up

Team Members

Andrew Bax(603783213), Pragadheeshwaran Thirumurthi(904000582), Prarthana Alevoor (604003285), Sharan Munyal (503994391), Shilpi Nayak (704000309), Shweta Phirke (103997749)

We have parallelized all operations using OpenMP as most of these operations involved data decomposition and we figured OpenMP works well when data decomposition is involved.

The details of parallelization of various operations are as follows:

Scaling

Guide to source code:

The source code for Scaling compiles on the SEASnet linux machines and it accurately scales the input image in sequential execution as well as in parallel execution. The task of assigning colors to the new calculated positions based on the formula X_{new}=Sx*X_{old} and Y_{new}=Sy*Y_{old} is parallelized. Hence each thread does the same task of assigning colors to pixels but a speedup is achieved as the overall task of scaling is divided among multiple threads. This means that there is a pragma to launch the threads before the first for loop.

Project breakdown:

The decision to parallelize the code this way was decided on the knowledge that each pixel is independent of every other pixel and that assignment of color can be done independently. This meant that a large amount of independent work was present which is ideal for a data decomposition speed up using OpenMP. This part of the code was implemented by Sharan Munyal, Pragadheeshwaran Thirumurthi and Shilpi Nayak together. This section of source code was worked on by Sharan Munyal. Both the sequential and the parallel code for the scaling tasks gave the same output with the performance (speedup) being better in parallel code. The output have been analysed in the Results section.

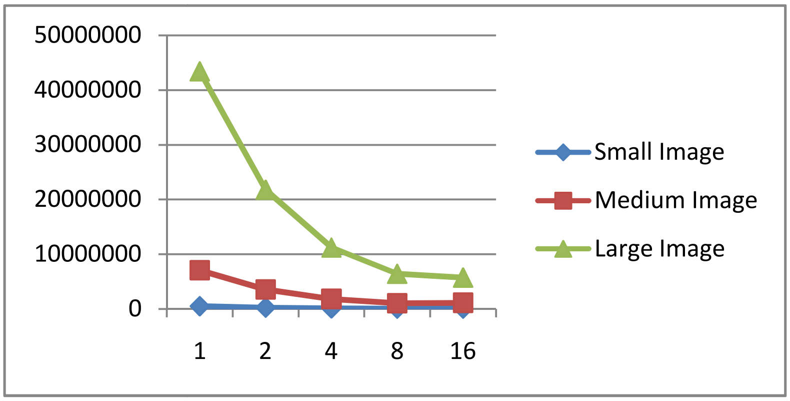
Bugs & Challenges:

There were any bugs in the implementation of this function. The implementation was straight forward and involved just one challenge. The challenge was to ensure that each thread works on its own set of new pixel co-ordinates and no two threads try to assign color to the same pixel causing inconsistency. If the input image was too small and if it was scaled down then it got distorted.

Final Results:

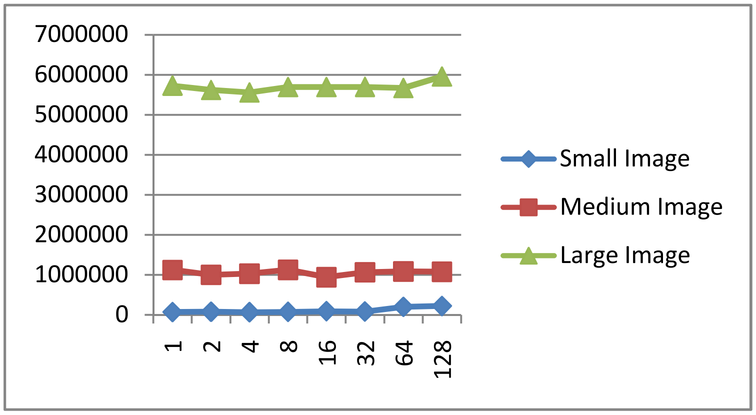
The parallelized code showed good gains in execution compared to the sequential version with the exception of small images with a large number of executing threads. Below is a input image with the produced output and graphs charting the execution time of different size images with a spectrum of execution threads. The image sizes used are 512 x512 (small), 2048x2048(medium-sized) and 5096x5096(large). A sample input and output image is as shown in the figure.

The figure on the left represents the input figure and on the right represents the output figure scaled down to 0.5 both horizontally and vertically.



Execution time (in micro-secs) v/s Number of Threads

The graph above represents execution time v/s number of threads for different image sizes as mentioned above. It can be observed that as the number of threads goes on increasing the execution time decreases irrespective of the image size. However highest speedup can be obtained in case of large images. The reason for this is that the image size increases the work done by each thread increases and the forking and joining granularity decreases proportionately.



Execution time (in micro-secs) v/s Chunk Size

The above graph displays the relation of execution time with chunk size for different images. It was observed that for all the images the execution time remained similar throughout. In some cases a minor fall in execution time was seen while in some case a small rise was observed. Hence it can be concluded that dynamic scheduling is not increasing performance here.

Rotation

Guide to source code:

The source code for rotation compiles on the SEASnet linux machines and it accurately rotates the input image in sequential execution as well as in parallel execution. The task of calculating the new pixel co-ordinates which is based on the rotation matrix(shown below) is parallelized. Hence each thread does the same task of calculating new pixel positions and assigning colors to it. Hence it is an example of domain/data decomposition. A speedup is achieved as the overall task of rotation is divided among multiple threads. There is a pragma to launch the threads before the first for loop to divide the iterations among threads.

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Project breakdown:

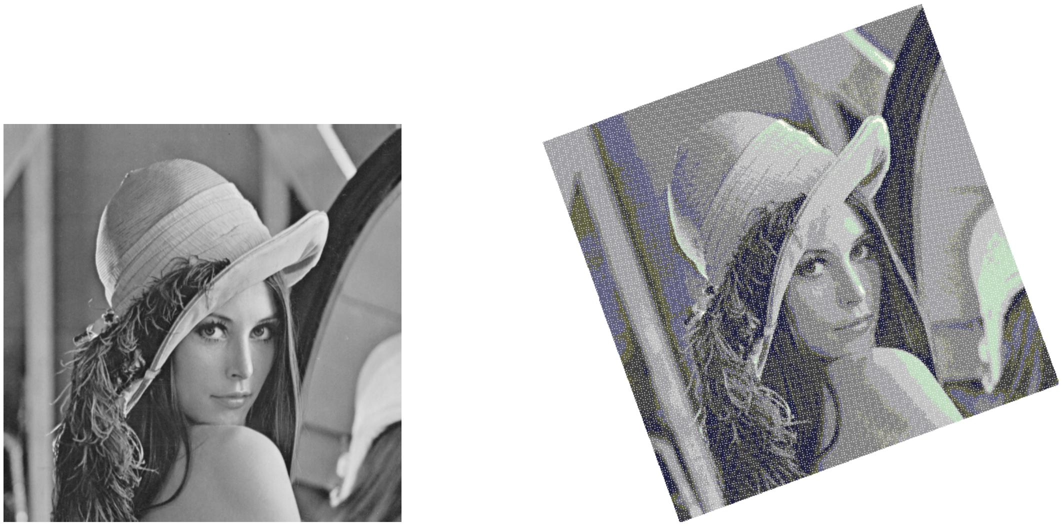
This part of the code was implemented by Sharan Munyal, Pragadheeshwaran Thirumurthi and Shilpi Nayak together. The decision to parallelize the code this way was decided on the knowledge that each pixel is independent of every other pixel and that assignment of color can be done independently. This meant that a large amount of independent work was present which is ideal for a data decomposition speed up using OpenMP. This section of source code was worked on by Sharan Munyal. Both the sequential and the parallel code for the rotation tasks gave the same output with the performance (speedup) being better in parallel code in almost all scenarios. The output has been thoroughly analysed in the Results section.

Bugs & Challenges:

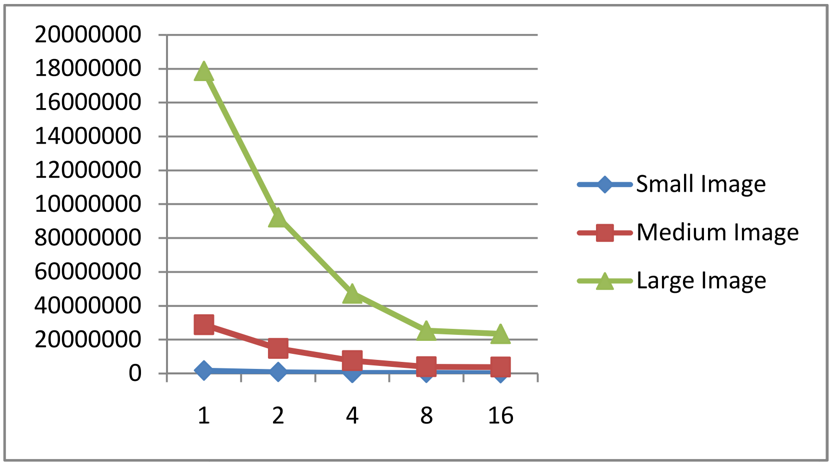
There weren't any bugs involved in the implementation of this function. The implementation was straight forward and involved one major challenge. As the new pixel co-ordinates were being calculated, for some irregular angle rotation (not a multiple of 90 degrees) some pixel co-ordinates were not shown as results and hence weren't assigned any colors in the new rotated image. For tacking this, an average of the neighboring four pixel's color was taken and assigned it to this uncolored pixel. This reduced the amount of white spots in the image. Also the size (width and height) of the new image had to be calculated mathematically for some irregular angle rotation.

Final Results:

The parallelized code showed good gains in execution compared to the sequential version with the exception of small images with a large number of executing threads. Below is a input image with the produced output and graphs charting the execution time of different size images with a spectrum of execution threads. Also shown are graphs to depict the relation of chunk size with the execution time for different images. The image sizes used are 512 x512 (small), 2048x2048(medium-sized) and 5096x5096(large). A sample input and output image is as shown in the figure.

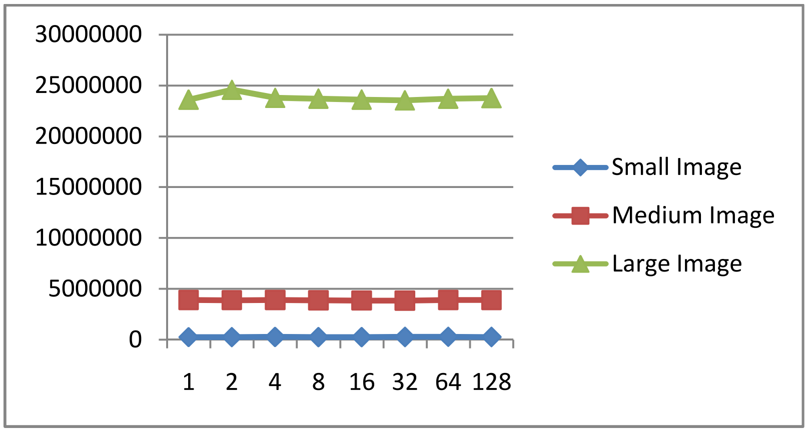


The figure on the left represents the input figure and on the right represents the output figure rotated by 20 degrees anti-clockwise.



Execution time (in micro-secs) v/s Number of Threads

The graph above represents execution time v/s number of threads for different image sizes as mentioned above. It can be observed that as the number of threads goes on increasing the execution time decreases irrespective of the image size. However highest speedup can be obtained in case of large images and when number of threads is 16. However in small image it was observed that execution time increased when threads increased from 8 to 16. The reason for this could be that the time spent by forking and joining for small images was proportionately more than the amount of work done.



Execution time (in micro-secs) v/s Chunk Size

The above graph displays the relation of execution time with chunk size for different images. It was observed that for all the images the execution time remained similar throughout. In some cases a small fall in execution time was seen while in some case a small rise was observed. Hence it can be concluded that chunk size does not impact performance.

Gaussian Blur

Guide to source code

The source code for Gaussian Blur compiles on the SEASnet linux machines and it correctly blurs the input image in sequential execution and in parallel execution. The thread code, using OpenMP, is located around each double for loop block where the horizontal and then vertical blur is computed for each pixel. This means that there is a pragma to launch the threads before the first for loops and then two pragmas, one for each block of for loops to divide up the pixels between the threads.

Project breakdown

This code was implemented by Andrew Bax. The decision to parallelize the code this way was decided on the knowledge that the computation of the blur of each pixel is independent of every other pixel. This meant that a large amount of independent work was present which is ideal for a data decomposition speed up using OpenMP. Despite all of this parallelization some synchronization was needed between the two for loop blocks. Here a barrier statement was inserted to make all threads wait for the intermediary image to be computed. On the gaussian blur function no completeness features of implementation were sacrificed but it was theorized that a functional decomposition could be implemented with the computation of the intermediary image and the final result using some synchronization to notify when a particular section is able to be computed. As the implementation for this proposed addition was very complex and the proof of parallel strength in image computation was shown it was shelved to focus on the presentation and report section of the project.

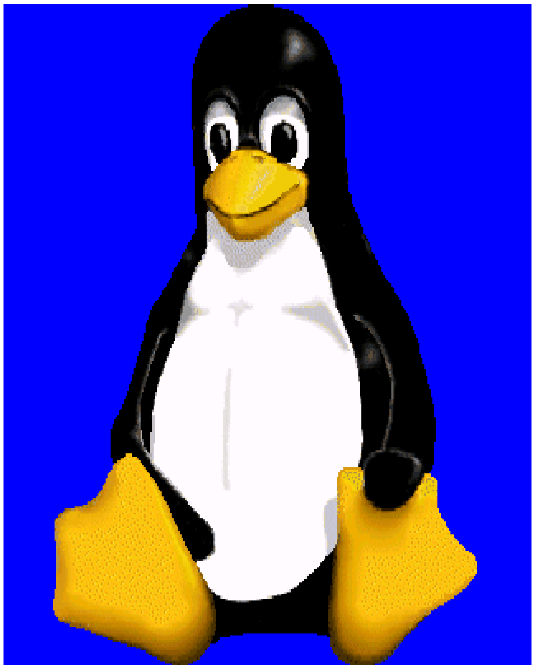
Bugs & Challenges

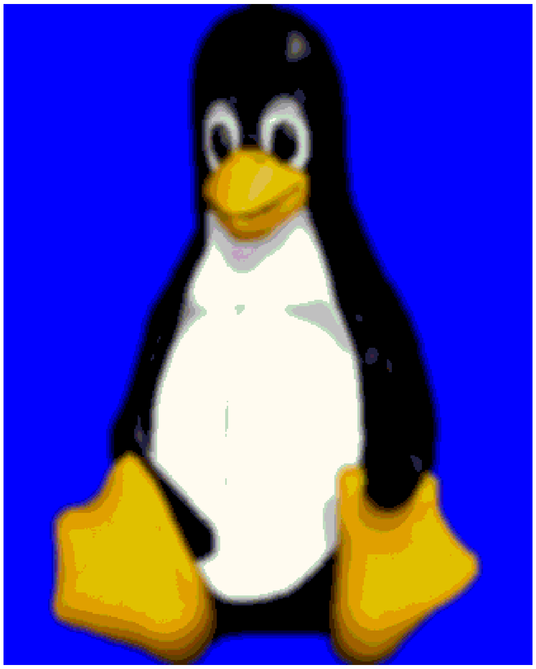
There were not very many bugs in the implementation of this function. One of the few bugs was originally perceived as a correctness issue with running the code in parallel on a local linux machine but when moved to the SEAS machines the code crashed and segfaulted immediately. After a short review of the parallel additions in the pragmas it was realized that an index variable for the gaussian kernel was not private for each thread. Once this was fixed the code ran on the SEAS machines and produces identical images to the sequential implementation. The real challenge with the OpenMP framework is ensuring that each variable is properly shared or set to private as it can greatly affect the execution of the parallel code. The only difficulty in extracting performance from the threads was when an input image would be too small for the gain in parallelization to be counteracted completely by the overhead to launch the threads. This was seen on the 400x320 image with 8 or more threads.

Final Results

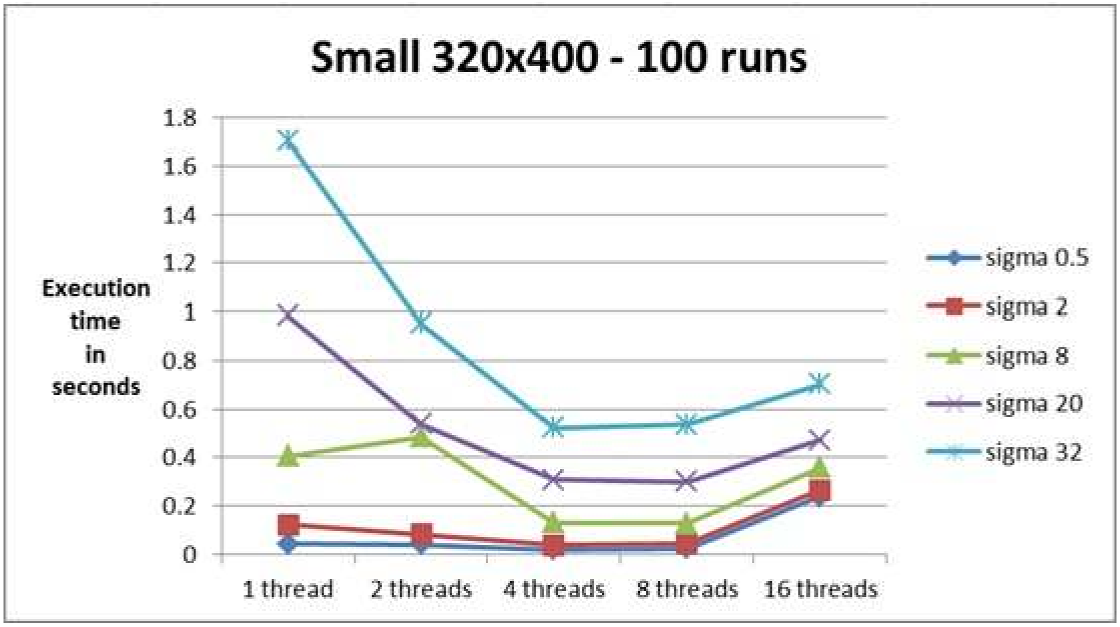
The parallelized code showed good gains in execution compared to the sequential version with the exception of small images with a large number of executing threads. Below is a input image with the produced output and graphs charting the execution time of different size images with a spectrum of execution threads.

Input image is shown on the left and the Gaussian blur of this image with a sigma of 2 is show on the right.

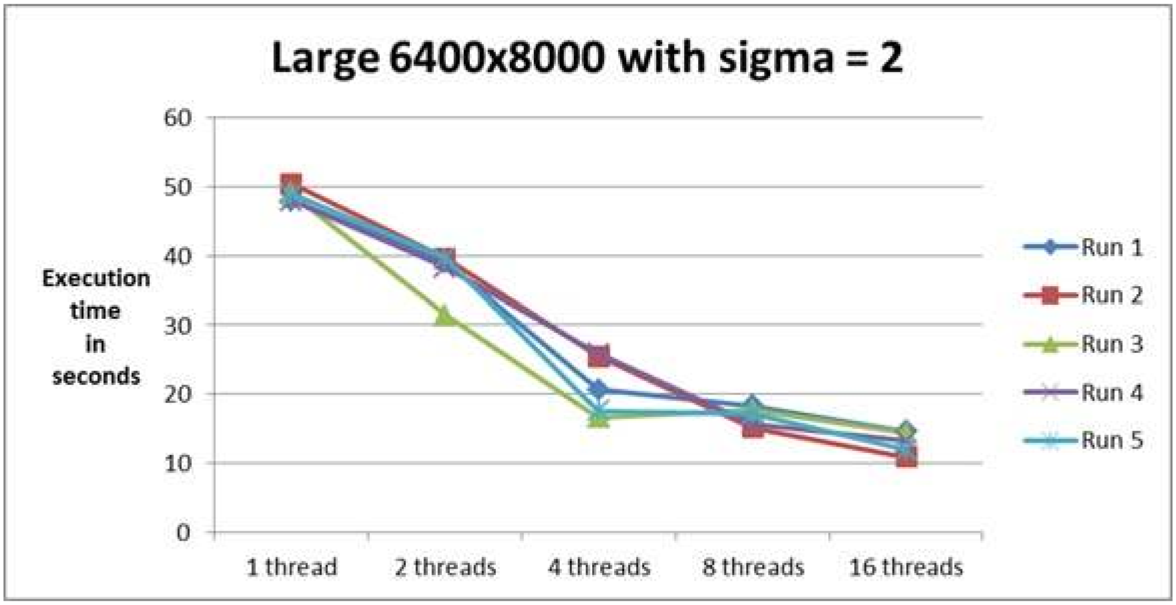




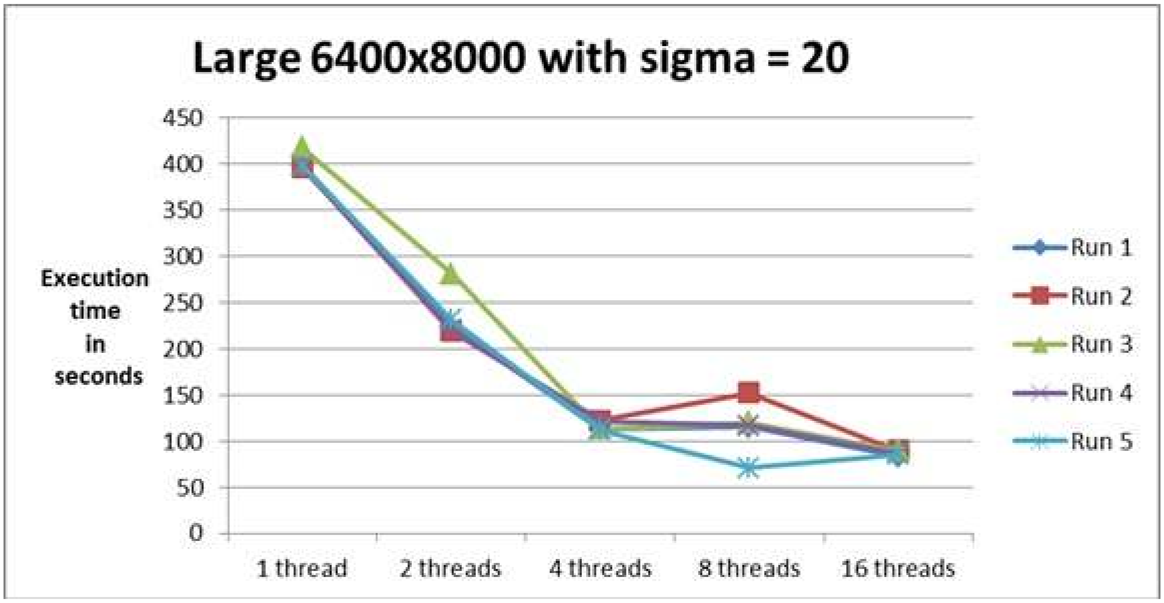
Below are the results for a small 320x400 pixel image with the average of 100 runs for each thread number and sigma value pair. There are good gains in parallelization up to 4 threads on average with speed up being better for higher sigma values where there is more work for each thread. At 8 and 16 threads and likely greater thread counts, the execution slows down and performs worse than with 4 threads. This is believed to be caused by thread overhead being too large for the small amount of work to be done.



Next are the results on an imaged scaled up 20 times, in both width and height. Due to the long read and write times for this image size running the Gaussian blur numerous times was impractical. The noticeable trait of this size image is that for 8 and 16 threads the execution of the blur was faster than small number of threads due to the large amount of independent work able to be done by threads.



Finally, here are some runs of the large image with a sigma of 20. This showed similar performance gains if not better than a sigma of 2 with a same sized image. At 8 threads the execution time of the 5 runs varied greatly which with more time and a little automation could be further tested with many more runs on that sample set.



Thresholding and Connected Component Labeling

For the sixth image processing technique we performed further methods of image segmentation [1] – image thresholding and connected components labeling. Image thresholding sets the pixels in the image to one or zero. Connected Component Labeling takes this thresholded image, groups the pixels in each separate region, and gives them unique labels.

Image Thresholding is performed based on histogram based segmentation, thus choosing the best threshold for the image gives a good thresholded image where the objects are finely differentiated from the background. We used the adaptive histogram technique which uses two passes. In the first pass, it calculates the histogram, smoothes it and uses the peak technique to find the high and low values. In the second pass it segments the image by separating the objects from the background and calculates the new high and low threshold values. Since these threshold values are compared with values between 0-255, hence we convert our color image to grayscale and then apply thresholding to get a binary output.

Image thresholding works on a grayscale input image. Hence we convert the color input to gray scale using the following formula and setting the same value as new rgb values:

rgb_pixel->red * 0.299 + rgb_pixel->green * 0.587 + rgb_pixel->blue * 0.114

Since thresholding is the common technique applied to any image processing task, we further continued by applying connected component labeling to the thresholded image. Connected Component Labeling is the technique of identifying connected pixel regions, i.e. regions of adjacent pixels which share the same set of intensity values (chosen between 0-255). The algorithm works on 8 connectivity – the fact that every pixel has 8 surrounding neighbors and hence uses stack.

```
Step 1: Given an image g with m rows and n columns

g(i,j) for i=1,m j=1,n

g(i,j) = value for object= 0 for background

Step 2. set g_label=102 this is the label value

Step 3. for (i=0; i<m; i++)

scan ith row

for (j=0; j<n; j++)

check jth element

stack_empty = true

if g(i,j) == value

Label and check neighbors

while stack_empty = false do

pop element (i,j) off the stack

Label and check neighbors

end while

Increment label to new value

end of checking jth element

end of scanning ith row
```

In the example below we see that 3 regions in the image have been detected as different components and labeled with unique numbers.

```
00000000000000000000
00000000000000000000
00011100000011111000
00011100000001110000
00111100000000100000
00001000000000000000
00000000000000000000
00000000000000000000
00001100000000100000
00011111000011110000
00001111100111000000
00000011111110000000
00000001111110000000
00000000110000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
```

Binary thresholding Image

```
00000000000000000000
00000000000000000000
00011100000022222000
00011100000002220000
00111100000000200000
00001000000000000000
00000000000000000000
00000000000000000000
00003300000000300000
00033333000033330000
00003333300333000000
00000033333330000000
00000003333330000000
00000000330000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
```

Connected Component Labeling

Finally applying connected components on various images:

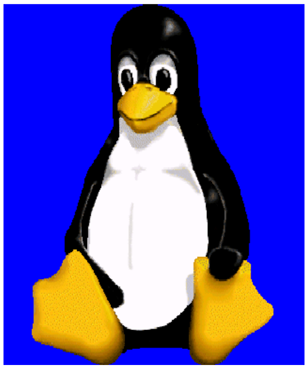


Image 1(a) Size - 320x400

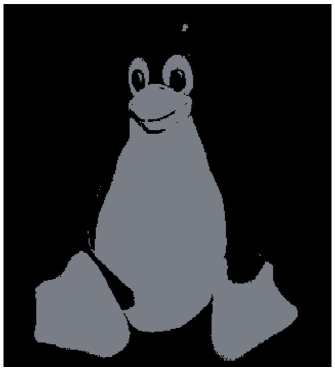


Image 1(b) 29 components

Image 2(a) Size - 640x480

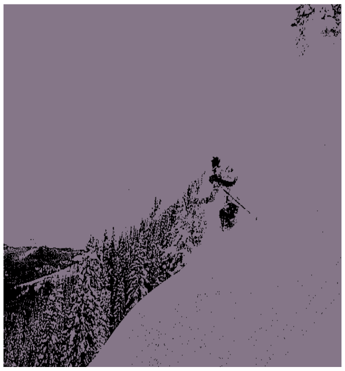


Image 2(b) 7401 components

As can be seen that in the image 1(b) the number of connected components are only 29, this is due to the fact that the whole white region is considered as one group due due its connectivity.

Where as in image 2(b), it has many components due to the presence of tree which differentiates each pixel given large number of components.

Guide to source code:

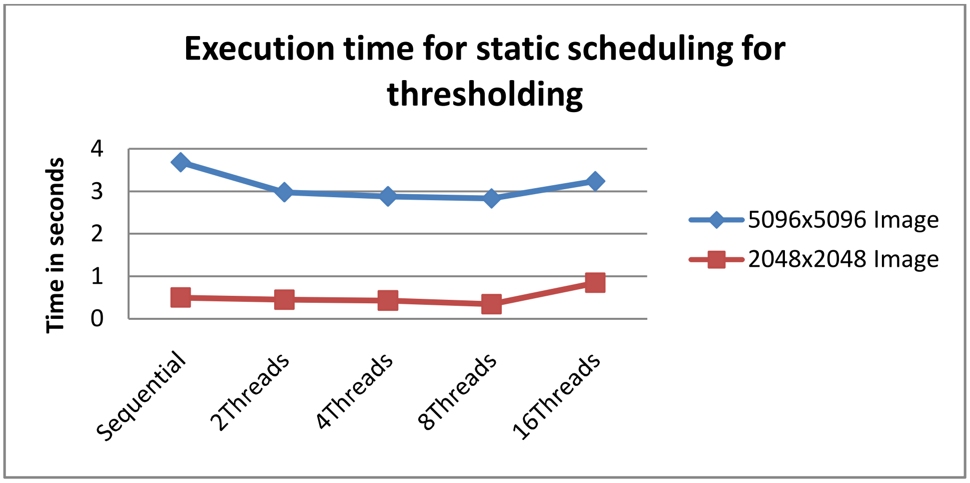
The source code in C of both sequential part as well as parallel part compiles on the SEASnet linux machines and correctly thresholds the input image and detects the connected components in both sequential and parallel execution. Performing thresholding on the image is done through fork-join threading model of Open MP. The parallel pragmas are added to for loops, which reads the image column by column and processes it. Parallelizing the code required usage of critical sections, especially while using label numbers or counters in the program. We also tried using critical sections for the stack part of the code, but later removed it as it was technically making the whole for loop sequential which is the same without the usage of parallel for and critical.

Project Breakdown:

This part of the code was implemented by Prarthana Alevoor. For thresholding, it was very evident that the results of one pixel do not affect the results of the other pixel. Hence we decided to perform data decomposition based on column of the image. Though the problem arised when we implemented the connected component labeling, where the result of each pixel depends on the surrounding pixels. Hence it was difficult to parallelize the stack portion of the connected components. It was not boosting our performance in any way. Hence we left the stack portion to run sequentially. Also there were sections in the code which required critical sections for the global or shared variable which was changed by each thread in its own loop. This section of the image processing task was written by Prarthana G Alevoor. Though the completeness of connected components labeling is not compromised, sufficient performance gains were not observed.

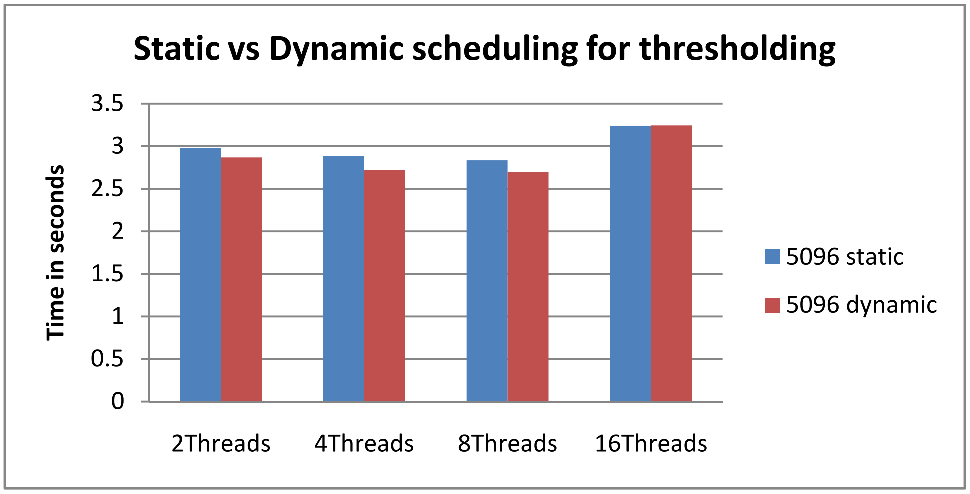
Final Results:

All the execution times have been taken without printing the labeled component output. Parallelizing initially involved parallelizing the thresholding part. The code was run on Seas Linux Servers with various numbers of threads. This was achieved easily and with a good improvement as shown in the figure.

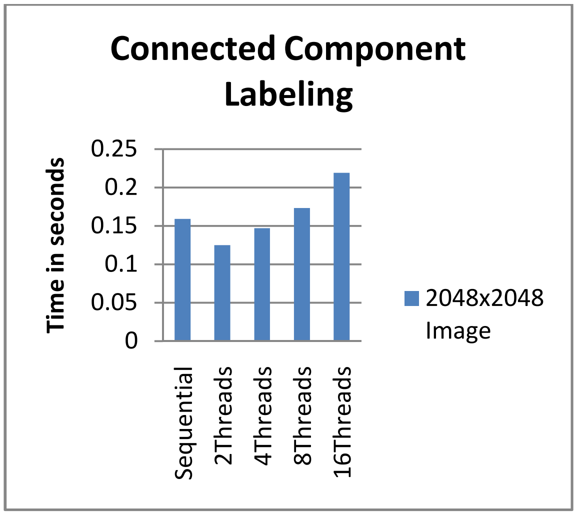


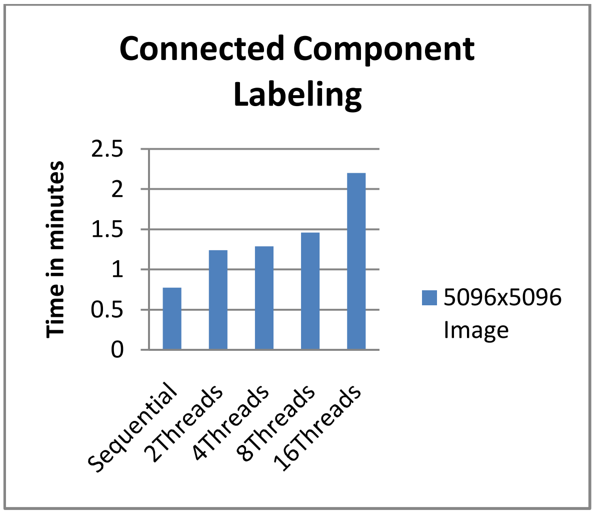
It shows that the code was parallelized well and hence the time reduced throughout from sequential part up to 8 threads. The speed up was as good as 1.3 times for image of size 5096x5096 for 8 threads. Though, we see an increase in the time for 16 threads. This might be due to the overhead for creating the threads in various for loops with the grain size being small.

The code was also executed for dynamic scheduling with chunk size of 4. The threads show a better performance with dynamic scheduling with almost equal division of work amongst threads.



Connected components labeling made use of Stacks heavily which was tougher to parallelize. Initially we got different values for different runs of parallelizing the stack for loop. As only a small portion could be parallelized, and the overhead of creating the threads were more we removed this parallel code. Even then since the stack portion takes majority of the time which is sequential, connected component labeling performs poorly in the parallelized section. The code was run on Seas Linux Servers with various numbers of threads.

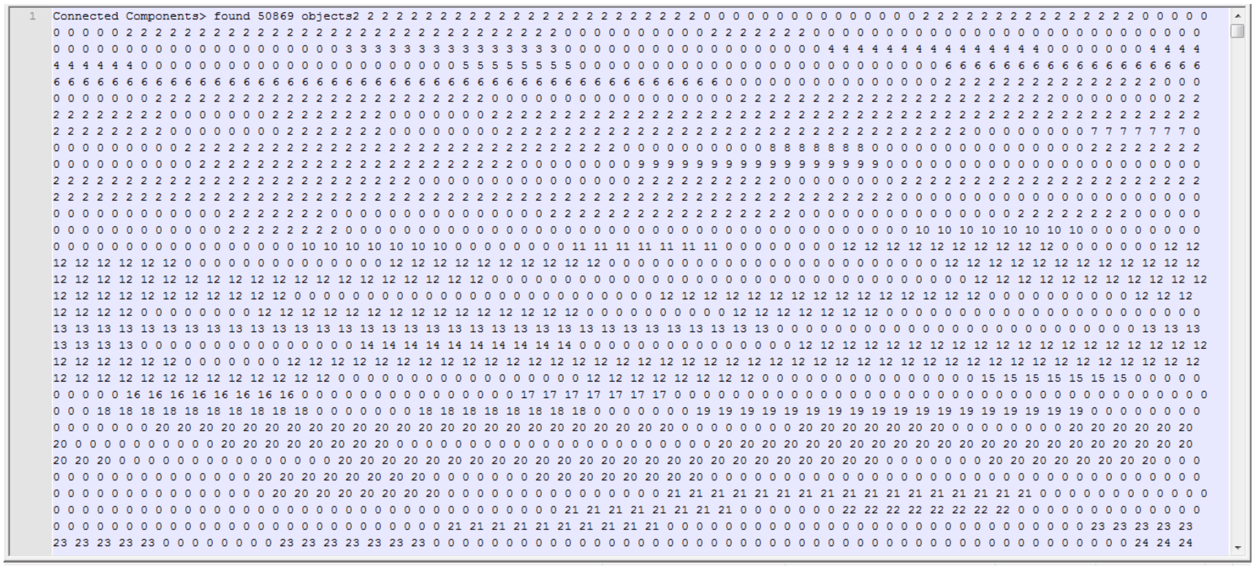




The parallel version of connected components is worse than the sequential version. This might be due to the fact that the connected component code which uses stack has to be anyway sequential(with the critical sections) and hence create a lot of overhead. Hence the grain size is small creating a lot of overhead for creating the threads in the 'for loops'. Though we see some improvement in the performance for 2048 size image for 2 and 4 threads.

Sample output

As shown above in figure 1(b) and 2(b), the binary images after thresholding along with the number of connected components are displayed. It also prints the whole image with its label values. I have taken the screen shot of the file which has the image label values for the first line. There are many components(regions) separated by 0's which determinate the region separation.



```
Connected Components> found 50869 objects2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0
0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 0 0 0 0 0 0 0 4 4 4 4
4 4 4 4 4 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 5 5 5 5 5 5 5 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0
0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 2 2
2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 7 7 7 7 7 7 7 0
0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 8 8 8 8 8 8 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2
0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 10 10 10 10 10 10 10 10 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 10 10 10 10 10 10 10 0 0 0 0 0 0 0 0 11 11 11 11 11 11 11 0 0 0 0 0 0 0 0 12 12 12 12 12 12 12 12 12 12 0 0 0 0 0 0 0 12 12
12 12 12 12 12 12 0 0 0 0 0 0 0 0 0 0 0 0 0 0 12 12 12 12 12 12 12 12 12 12 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 12 12 12 12 12 12 12 12 12 12 12 12 12 0 0 0 0 0 0 0 0 0 0 0 0 0 12 12 12
12 12 12 12 0 0 0 0 0 0 0 0 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 0 0 0 0 0 0 0 0 0 0 12 12 12 12 12 12 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 13 13 13
13 13 13 13 0 0 0 0 0 0 0 0 0 0 0 0 0 0 14 14 14 14 14 14 14 14 14 14 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 0 0 0 0 0 0 0 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12 12 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 12 12 12 12 12 12 12 12 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 15 15 15 15 15 15 15 0 0 0 0 0
0 0 0 0 0 16 16 16 16 16 16 16 16 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 17 17 17 17 17 17 17 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 18 18 18 18 18 18 18 18 18 18 0 0 0 0 0 0 0 18 18 18 18 18 18 18 18 0 0 0 0 0 0 0 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 0 0 0 0 0 0 0 0 20 20 20 20 20 20 20 0 0 0 0 0 0 0 0 20 20 20 20 20 20
20 0 0 0 0 0 0 0 0 0 0 20 20 20 20 20 20 20 20 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 0 0 0 0 0 0 0 20 20 20 20 20 20 20 20 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 20 20 20 20 20 20 20 20 0 0 0 0 0 0 0 20 20 20 20 20 20 20 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 20 20 20 20 20 20 20 20 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 21 21 21 21 21 21 21 21 0 0 0 0 0 0 22 22 22 22 22 22 22 22 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 21 21 21 21 21 21 21 21 21 21 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 23 23 23 23 23
23 23 23 23 23 0 0 0 0 0 0 0 0 23 23 23 23 23 23 23 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 24 24 24
```

Bugs and Challenges:

One interesting challenge we came across was parallelizing stack implementation which is used for adding the neighboring points in LIFO order. We tried using critical sections with proper OpenMP codes, still parallelizing the stack gave varying outputs. Also another importing finding during this work was,

```
#pragma omp parallel for{

    // code snippet

}
```

vs

```
#pragma omp parallel

{    #pragma omp for

{

    // code

}

}
```

Though the above two snippets follow the same logic, they give different results. Debugging the parallel code to find the root cause of any issues was much tougher as it was very difficult to know follow the execution of various threads. Also the ‘for loops’ used to run infinitely sometimes. It was debugged and found that there was a special case when there used to be a Push of the pixel and Pull of the pixel for 2 adjacent pixels. This used to cause the two connected component labeling functions to be called repeatedly infinitely. This was solved by using a variable as a special case check and not inserting the pixel again if it was already checked and has no connected components beside it.

[1] Image Processing in C by Dwayne Phillips

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:

$$M = \begin{bmatrix} A & C \\ C & B \end{bmatrix}$$

$$\textit{where}:\ A = \left(\frac{\partial I}{\partial x}\right)^2 \otimes w,\ B == \left(\frac{\partial I}{\partial y}\right)^2 \otimes w,\ C = \left(\frac{\partial I}{\partial x}\frac{\partial I}{\partial y}\right) \otimes w$$

$$\otimes \ \textit{is the convolution operator}$$

$$w \ \textit{is the Gaussian window}$$

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

$$\begin{aligned} C(x,y) &= \det(M) - k(\operatorname{trace}(M))^2 \\ \det(M) &= \lambda_1\lambda_2 = AB - C^2 \\ \operatorname{trace}(M) &= \lambda_1 + \lambda_2 = A + B \\ k &= \text{constant} \end{aligned}$$

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.

Guide to source code:

The source code for corner detection compiles properly without errors and warnings. As per the algorithm described above, each of the points form sections in the code and all the sections are working as needed. We have implemented the above algorithm in C and then parallelized using OpenMP parallel framework. The parallelized pragmas are found at:

- The first section of the code where the arrays for holding differentials along x, y and diagonal axes are declared and data written into them. It can be found at line 91.
- The second section of the code where values A, B and C are constructed. It can be found at line 139 in the parallelized code. This is also the area where the cornerness map is built. The private variables are declared as required as each thread will need to calculate the required values separately.
- The third section where actual corner is detected by shifting a 3x3 window through the image and highlighting the corners. It can b found at line 175.

Project Breakdown:

Since it is a task of data decomposition, OpenMP seemed the best suited choice. The images are read column-wise using the ‘for’ loops. Thus parts of the image column-wise were distributed through the threads. No synchronization was needed as there was no common resource that all threads needed to access or modify. The Corner Detection code was implemented by Shweta Phirke. Parallelizing the code was simple as it only included inserting pragma statements at the ‘for’ loops. On the corner detection function, no section was compromised in regards to completeness.

Bugs and Challenges:

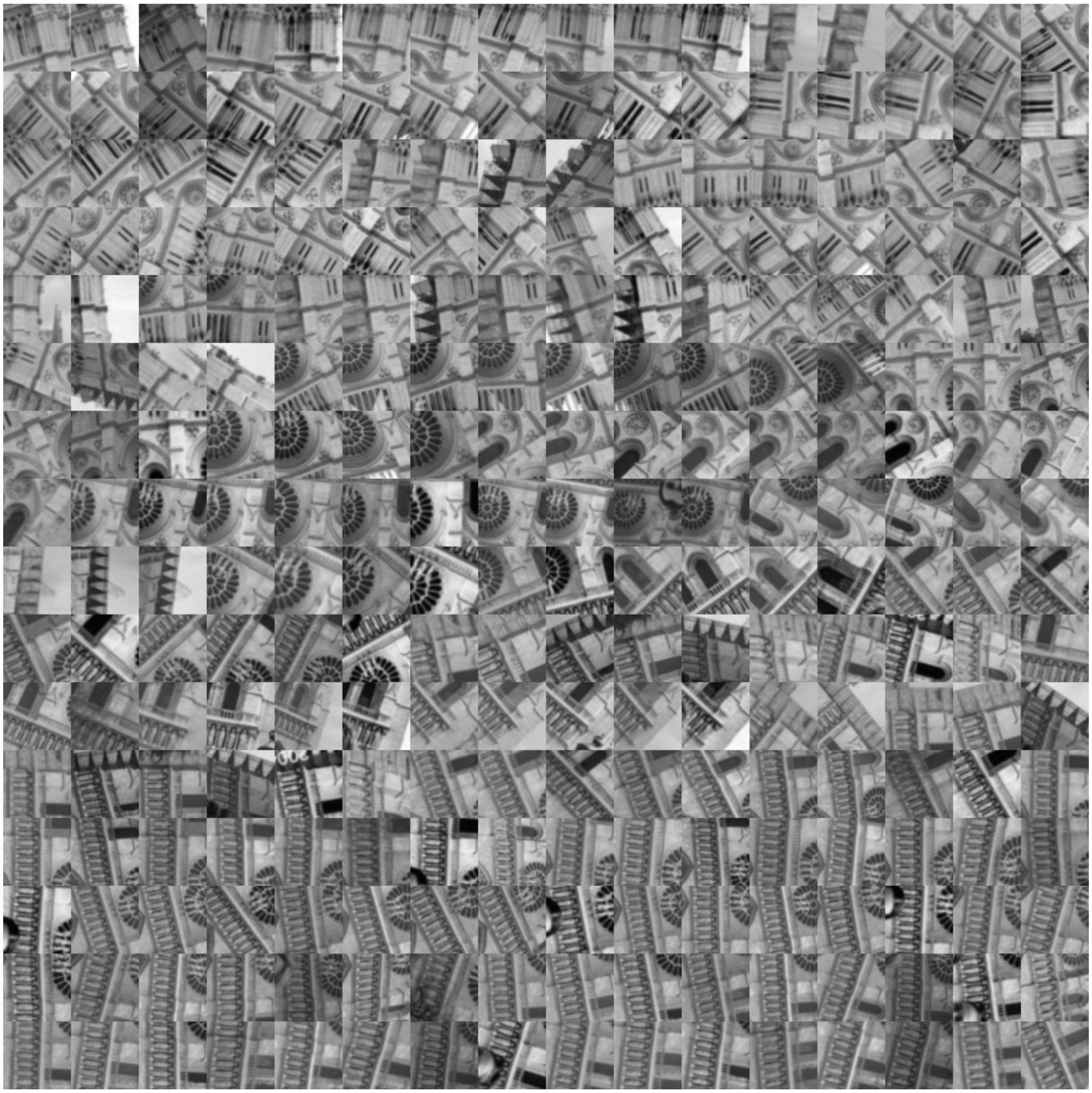
This code was implemented by Shweta Phirke. This algorithm is computationally a bit heavy as in it involves reading and writing from many arrays. But apart from this the code ran perfectly as per the algorithm. There was no difficulty in parallelizing the code. It was a challenge finding an 8bpp image in grayscale tone. After finding one, scaling it to different sizes also took time. But we were able to finally use a couple of images which gave us good results. Also when executing on SEAS machines, if a small sized image was used, it did not give good results. So, sufficiently large images were required. The last section of the program involves 4 for loops. Parallelizing this section was a challenge as where the pragma should be inserted was a task. An option would have been to parallelize the 3rd inner for loop. We decided we should go ahead with parallelizing the outermost for loop and it gave good results.

Final Results:

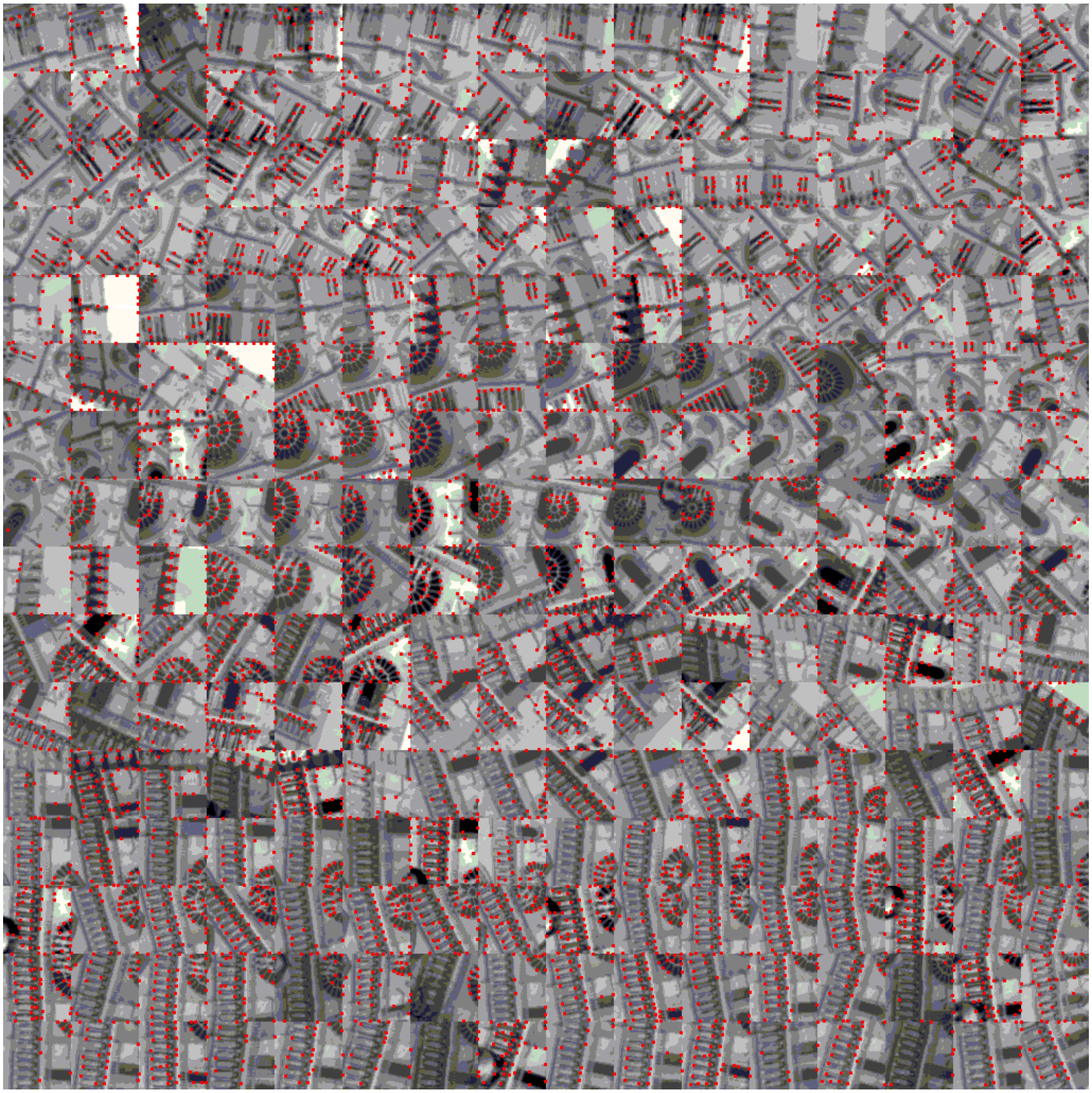
The experiments were carried using several images ranging from 512x512 to 2048x2048. Presented below are the results of implementing on the following:

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

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



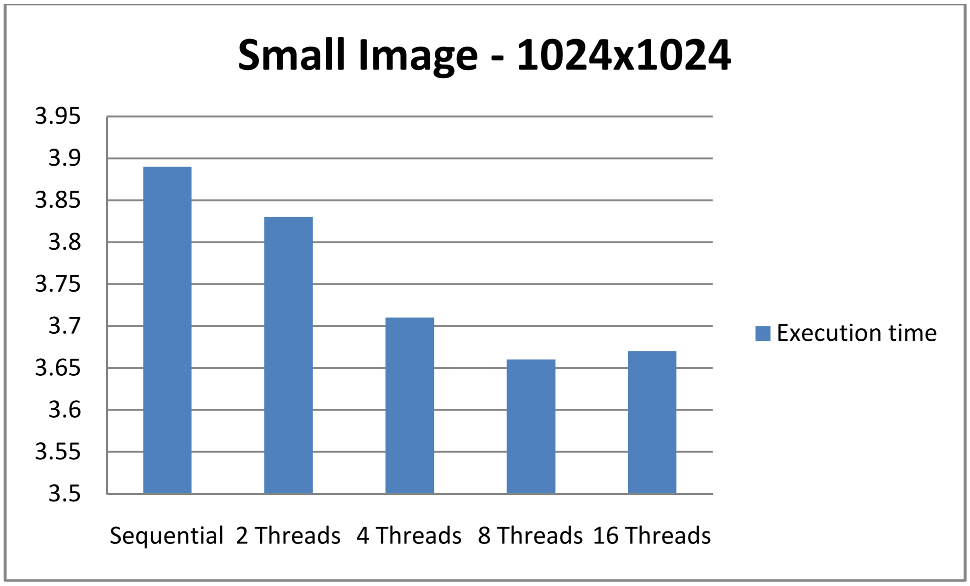
The image obtained after applying corner detection. The corners are highlighted as red dots.



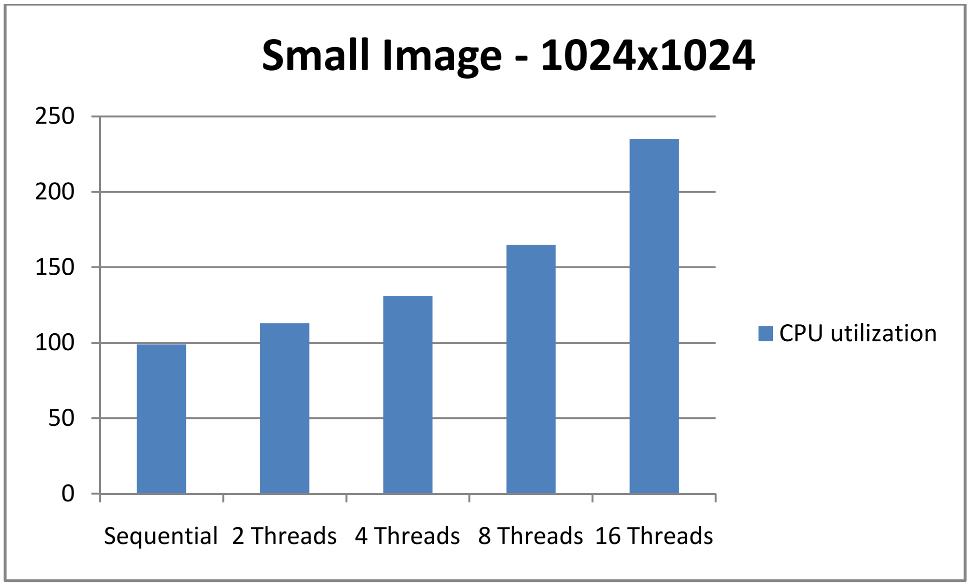
Runs carried out on Small Image (1024x1024) and results:

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.

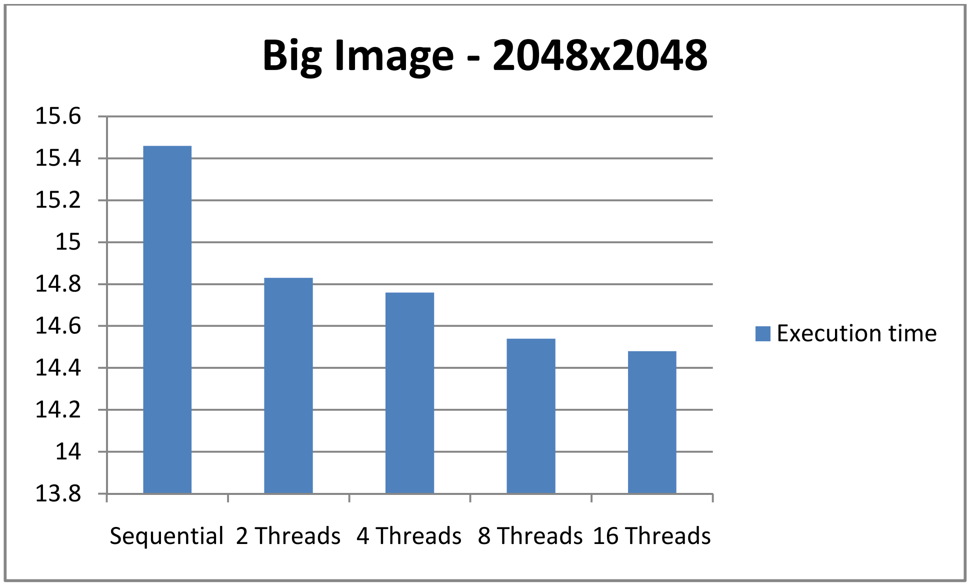


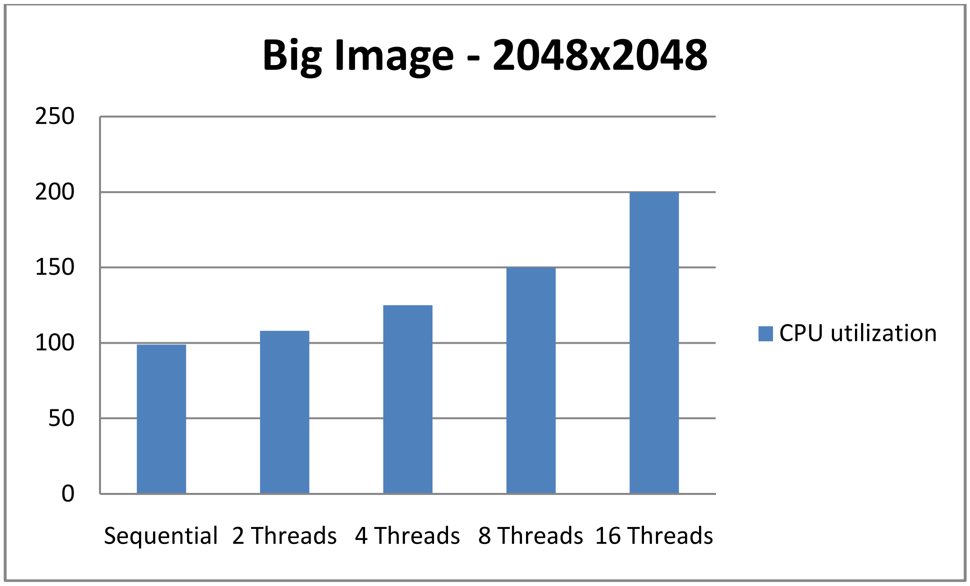
This is a good indication that the CPU is kept busy with lot of work rather than spending idle time. The CPU utilization measurements are in %.

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

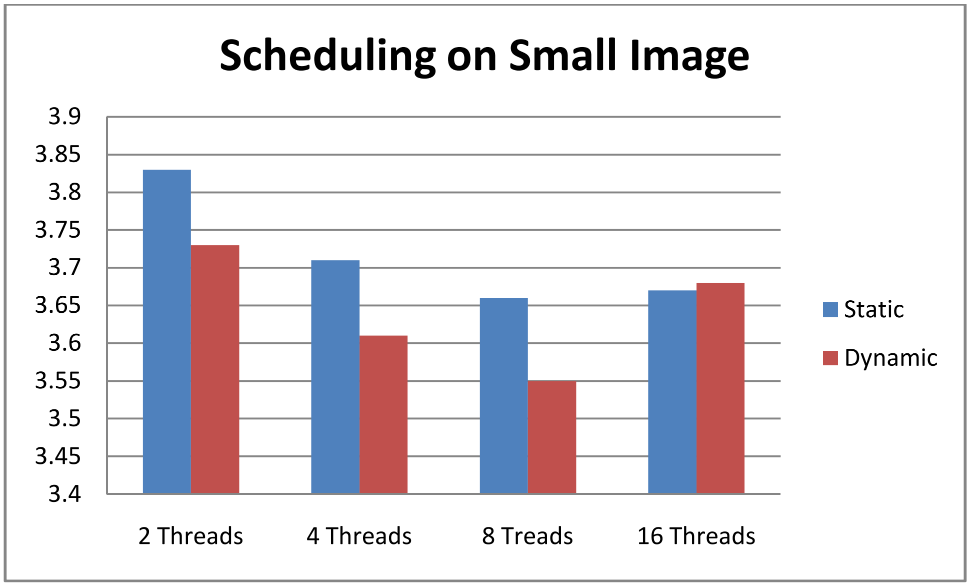




Apart from this, parallelizing was also tested using dynamic scheduling. For this, again UNIX time was used to get the execution times. Shown below is the comparison of execution times (secs) for static and dynamic scheduling for a 1024x1024 image by keeping the chunk size fixed to 4 and testing for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

Reference for algorithm:
http://kiwi.cs.dal.ca/~dparks/CornerDetection/harris.htm

Motion Estimation

Overview:

Our aim is to determine motion vectors that describe the transformation from one 2-dimensional bitmap image to another which are adjacent frames of a video sequence. We have implemented a full search block matching algorithm. The idea behind this is to divide the current frame into a matrix of blocks which are then compared with corresponding block and its adjacent neighbors in the previous frame to create a vector which shows the movement of a block from one location to another in the previous frame. This movement calculated for all the blocks comprising a frame, constitutes the motion estimated in the current frame. The search area for a good block match is constrained up to p pixels on all fours sides of the corresponding block in previous frame. This 'p' is called as the search parameter. Larger motions require a larger p and the larger the search parameter the more computationally expensive the process of motion estimation becomes. Usually the block is taken as a square of side 20 pixels, and the search parameter p is 10 pixels. The matching of one block with another is based on the output of a cost function which here is Mean Absolute Difference (MAD). The block that results in the least cost is the one that matches the closest to current block.

Guide to source code:

This part of the code was implemented by Sharan Munyal, Pragadheeshwaran Thirumurthi and Shilpi Nayak together. The source code compiles on the SEASnet linux machines and executes to give correct output. All the sections of the code are complete. We are drawing line as motion vector to show the movement in frames. We have parallelized the code in two ways using openMP. Motion estimation is implemented using four nested for loops. The outer two loops select the block and the inner two select a window. Using OpenMP, we can parallelize only one loop as nested parallelism is not supported. We have parallelized two loops separately and recorded the time taken.

The complexity of sequential code is Order of width*height*(2*window_size+1)^2*block_size^2 which is quite high.

First Method of Parallelizing:

Here the threads take up work column-wise i.e., each thread selects one column and chooses blocks along the height. So the forking is done only once at the start of the loop. Each block to be matched by a thread has starting pixel in that column. We achieved significant speed up on parallelizing the outermost loop in 800*600 size image.

Second Method of Parallelizing:

Here the inner loop is parallelized. The threads divide the data for each column i.e., for every column we fork the threads and they divide blocks that begins in that column among them. The fork join process is carried out for each column. So we are forking the threads as many times as there are columns which are the width of the image. We achieve significant parallelism with this method as well in a 600*800 image.

Comparison:

The overhead of forking and joining in the second method is more and hence even if we achieve significant parallelism in both, the first method works slightly better on 800*600 size image.

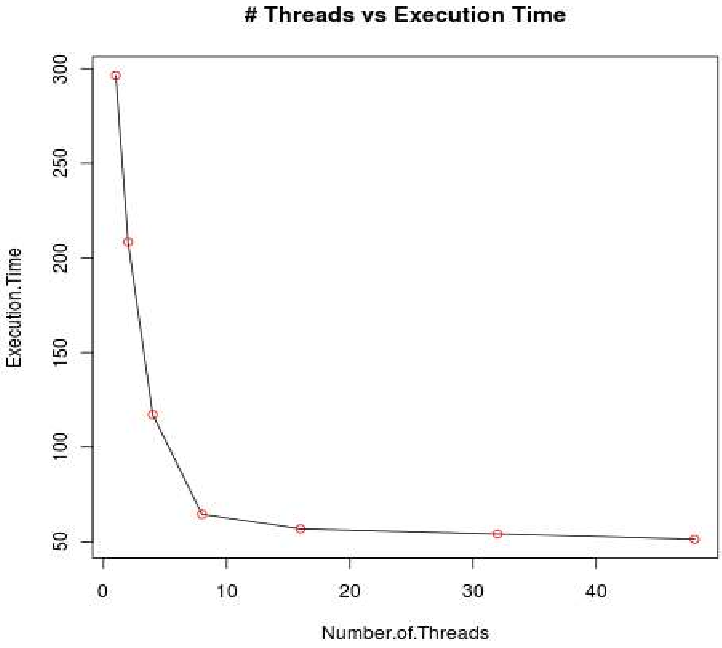
Here we used barrier synchronization supported by default in openMP. The barrier exist at the end of any parallelized for loop. We did not draw full vectors but have drawn line to show the change between two frames.

Bugs and Challenges:

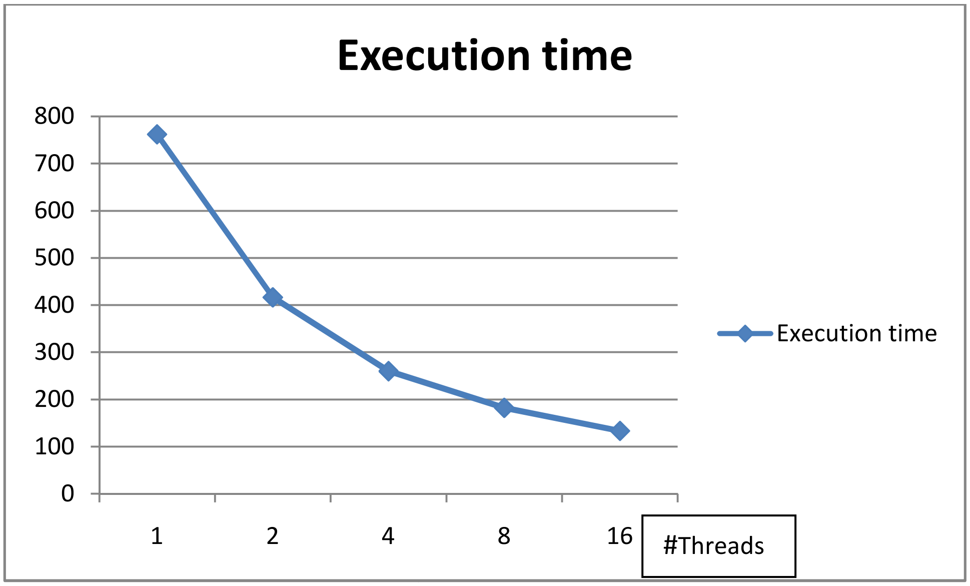
In implementing motion estimation, we had problem in determining the parameters to use for block_size and window_size. We did not run into difficulty for extracting good performance from threads. From this project, we understood the benefits of parallelizing clearly. Also, we could compare the impact of parallelizing different loops.

Final Results:

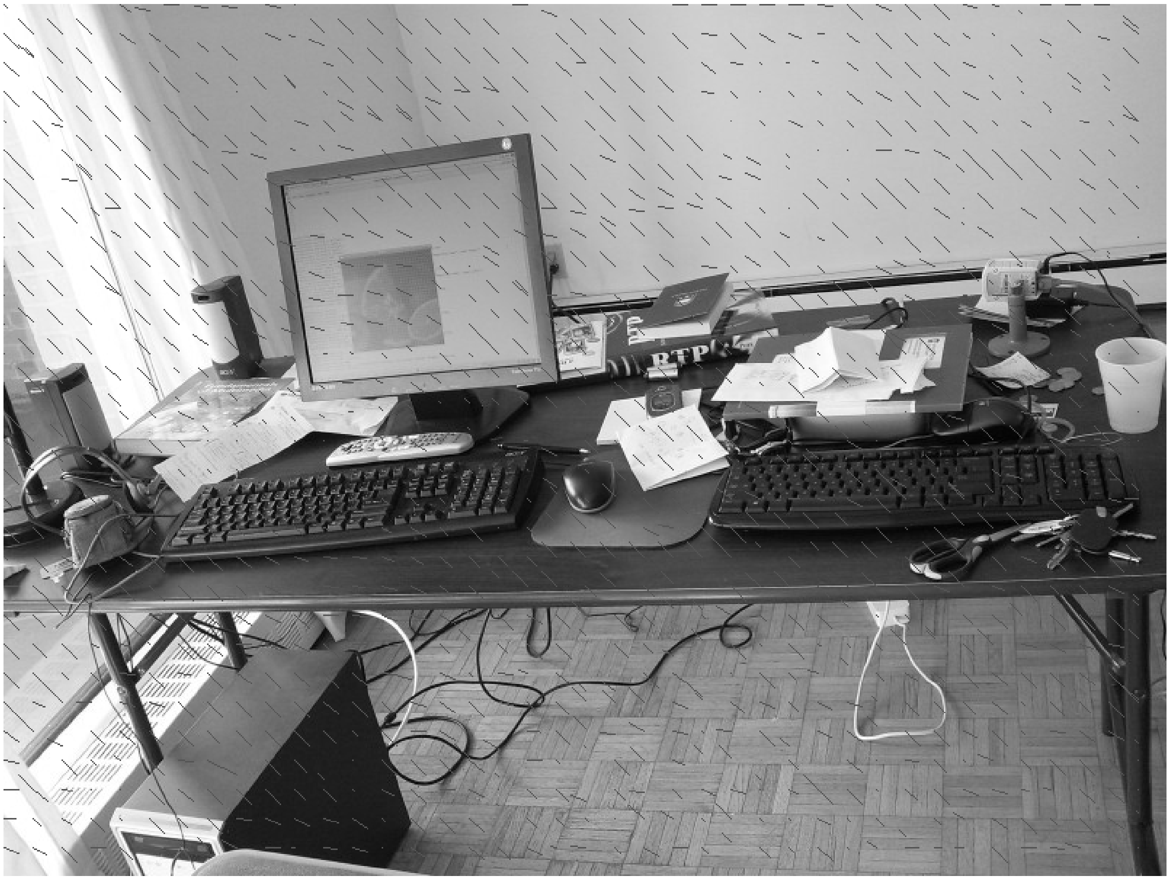
In comparison to the sequential code, the parallel code gave higher performance. We got a speedup of 5.3 when we used 16 threads in 800*600 size image. The performance output is as follows when we parallelized the outer loop:



As the input size of image increase, the speed up increases. We have tested on an input image of size 1200*1400 and achieved a speed up of 5.7. The output is as follows:

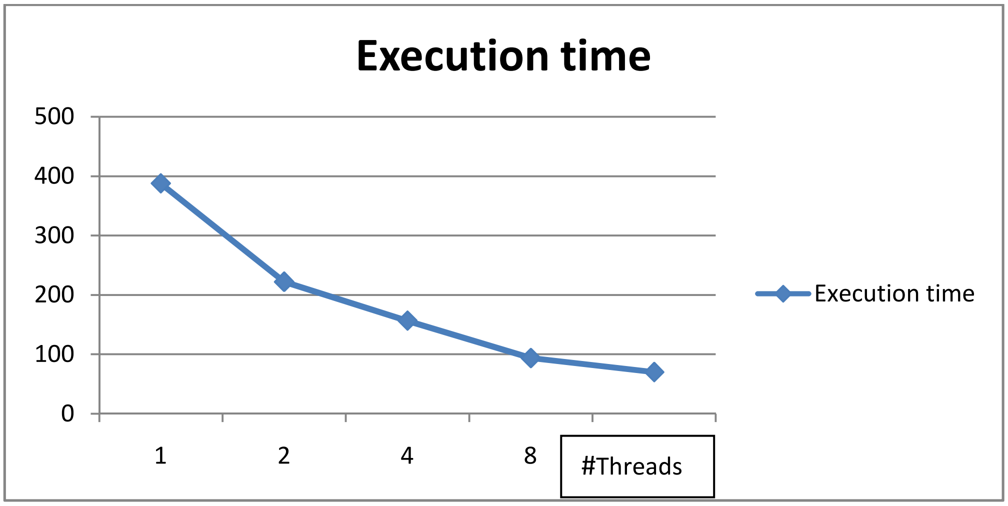


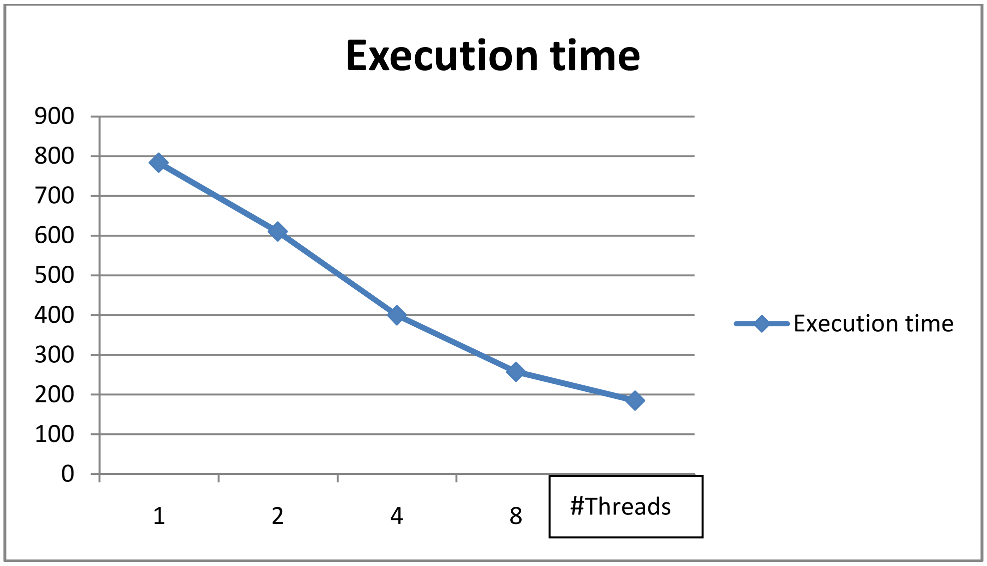
The sample output of our code is as follows:



Here we have used lines to show the motion of the pattern/blocks.

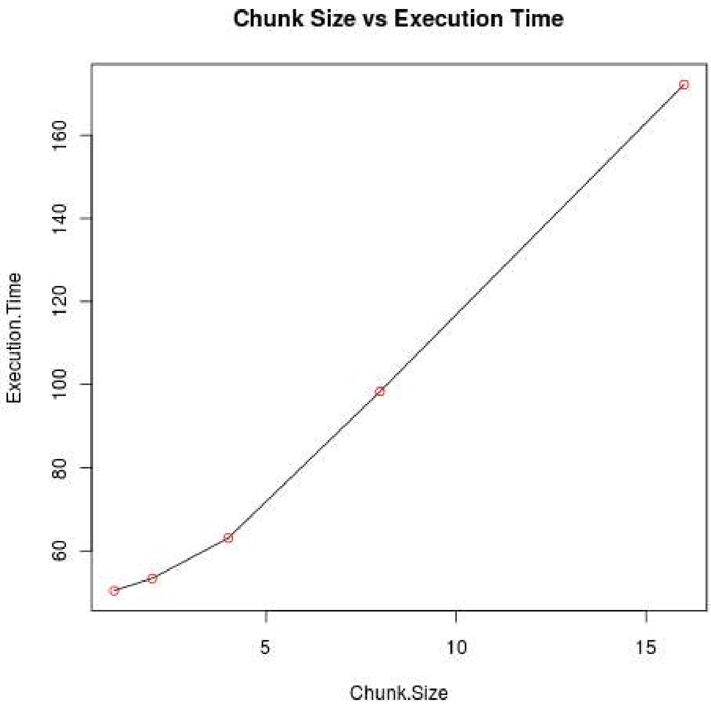
We have also implemented an alternate parallel code when the inner loop was parallelized. Improvement here is slightly lower than the first method due to overhead of fork-join. The performance output is as follows for 600*800 and 1200*1600 size image respectively:





The performance of the code increases with the number of threads increase as tested in SEASnet server.

Also with dynamic scheduling, increasing the chunk size we achieved a decrease in performance.



Conclusion:

So we achieved substantial improvement by paralleling all the operations using OpenMP.