

NAIVE BAYES CLASSIFIER

+

IMAGE CONVOLUTION IN OPENMP

TEAM MEMBERS

Rahul Pillai	01FB15ECS224
Rahul Pujari	01FB15ECS223
Parashara R	01FB15ECS202

PART 1

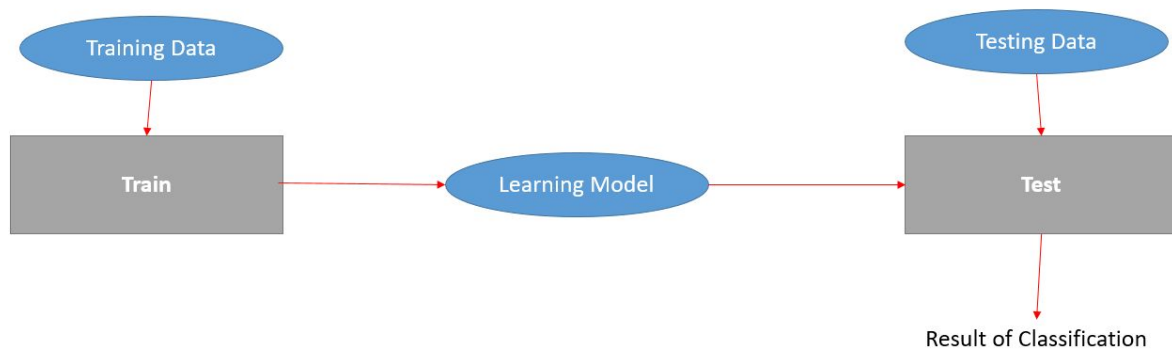
NAIVE BAYES CLASSIFIER

AIM

To build and parallelize a Naive Bayes Classifier in OpenMP.

SUPERVISED LEARNING

Supervised learning has always been a crucial part of day to day life for humans as well as computers. The concept of supervised learning stems from the way in which humans gain their knowledge and learn to identify different objects and conditions. For example, we know that there is a good chance of snow in the month of January. We know this based on the knowledge we have gathered by living in Colorado over the years. Here, the event of snowing is related to a certain time based on the knowledge we gathered from our experience. This concept can also be applied to computers using algorithms capable of learning from labelled data and using the extracted knowledge to predict the label of some input unlabelled data. These algorithms are called Classifiers. Classifiers make use of labelled data (aka Training data) to extract knowledge and then apply this knowledge to unlabelled data (aka Testing data) to predict the labels. Following diagram shows the conventional classification process.



Our project focuses on one classifier which is popularly known as “Naïve Bayesian” classifier.

NAIVE BAYES CLASSIFIER

Naïve Bayesian classifier is a simple probabilistic classifier which works by applying the Baye's theorem along with naïve assumptions about feature independence. Studies comparing classification algorithms have found Naive Bayesian comparable in performance with Decision Tree and few neural network classifiers. Naive Bayesian has also exhibited high accuracy and speed when applied to huge amounts of data [X]. It assumes value of any feature is independent of values of other features. This assumption is also known as Conditional Independence. Despite the naïve assumption and over simplification, Naïve Bayesian classifiers have proved to be quite useful in complex real world conditions. Probabilistic model of Naïve Bayesian Classifier is as follows:

$$P(X|C_i) = \prod_{k=1}^n P(x_k|C_i)$$

Naïve Bayesian considers the probability of a tuple X belonging to a class C i is equal to the multiplication of probabilities that each attribute of the tuple X belongs to the class C i where the probabilities $P(x_1|C_i)$, $P(x_2|C_i)$ and so on can be easily calculated from the training data sets. Once these probabilities are calculated for each attribute in X, the probabilities with respect to individual classes are multiplied. The class with maximum probability is selected as the result of the classification. The theoretical training and testing time complexity of Naive Bayesian is $O(N*P)$ where N is number of records to be tested and P is the number of features. But while considering the practical implementation of Naive Bayesian classifier, we can see that the training complexity is $2 * O(N*P)$ as the probabilities must be calculated for each attribute value after getting the joint distribution counts and testing complexity is $O(N*P*C)$ where C is the number of possible class values.

DATASET SAMPLE

age	income	student	credit	buys
<=30	high	no	fair	no
<=30	high	no	excellent	no
31 .. 40	high	no	fair	yes
>40	medium	no	fair	yes
>40	low	yes	fair	yes
>40	low	yes	excellent	no
31 .. 40	low	yes	excellent	yes
<=30	medium	no	fair	no
<=30	low	yes	fair	yes
>40	medium	yes	fair	yes
<=30	medium	yes	excellent	yes
31 .. 40	medium	no	excellent	yes
31 .. 40	high	yes	fair	yes
>40	medium	no	excellent	no

IMPLEMENTATION

Age	Buys	Count
<=30	No	3
31-40	No	0
>40	No	2
<=30	Yes	2
31-40	Yes	4
>40	Yes	3

Credit	Buys	Count
Fair	No	2
Excellent	No	3
Fair	Yes	6
Excellent	Yes	3

Income	Buys	Count
High	No	2
Medium	No	2
Low	No	1
High	Yes	2
Medium	Yes	4
Low	Yes	3

Stud	Buys	Count
No	No	4
Yes	No	1
No	Yes	3
Yes	Yes	6

Buys	Buys	Count
No	No	5
Yes	Yes	9

These frequency counts indicate the number of times a value of each of the columns has occurred along with each possible class. Simply put, these counts are spread across the Cartesian products of the domains of each of the non-class attribute and the domain of the class attribute. These counts allow Naïve Bayesian to build a probabilistic model by dividing the frequency counts of (value, class label) pairs by the frequency count of respective (class label, class label) pairs. For example, with the frequency counts given above, we can calculate the probability of the pairs (<=30, Yes) and (Fair, Yes) as follows:

$$P(\leq 30|Yes) = \frac{FC(\leq 30, Yes)}{FC(Yes, Yes)} = \frac{3}{9} = 0.33$$

$$P(Fair|Yes) = \frac{FC(Fair, Yes)}{FC(Yes, Yes)} = \frac{6}{9} = 0.67$$

Age	Buys	P(B A)
<=30	No	0.6
31-40	No	0
>40	No	0.4
<=30	Yes	0.22
31-40	Yes	0.44
>40	Yes	0.33

Income	Buys	P(B A)
High	No	0.4
Medium	No	0.4
Low	No	0.2
High	Yes	0.22
Medium	Yes	0.44
Low	Yes	0.33

Stud	Buys	P(B A)
No	No	0.8
Yes	No	0.2
No	Yes	0.33
Yes	Yes	0.67

Credit	Buys	P(B A)
Fair	No	0.4
Excellent	No	0.6
Fair	Yes	0.67
Excellent	Yes	0.33

Buys	Buys	P(B)
No	No	0.36
Yes	Yes	0.64

Once these have been calculated, the classifier is ready to classify a record.

For example, if we see customer X: <=30, medium, yes, fair

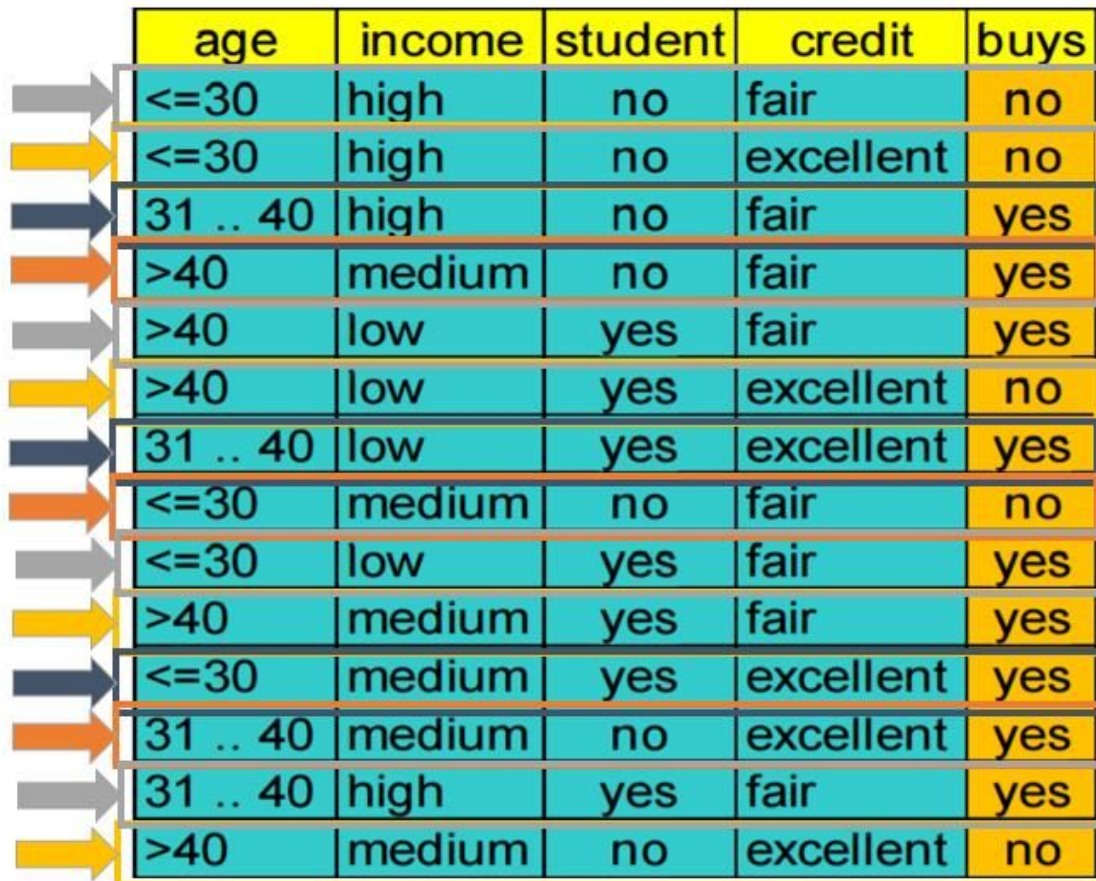
$$P(X|\text{"yes"}) = 0.22 * 0.44 * 0.67 * 0.67 * 0.64 = 0.028$$

$$P(X|\text{"no"}) = 0.6 * 0.4 * 0.2 * 0.4 * 0.36 = 0.007$$

$P(X|\text{"yes"}) > P(X|\text{"no"})$, therefore we predict that the customer will buy.


PARALLEL NAIVE BAYES

Naïve Bayesian is one of the few classifiers which are very well built for parallel computation. So much so that we might even be able to call it an embarrassingly parallelizable algorithm. Taking the concept of the classifier into consideration, there are two ways in which it can be parallelized. For now, let us focus only on the training part of the classifier, the first way to parallelize this algorithm is as follows:



	age	income	student	credit	buys
→	<=30	high	no	fair	no
→	<=30	high	no	excellent	no
→	31 .. 40	high	no	fair	yes
→	>40	medium	no	fair	yes
→	>40	low	yes	fair	yes
→	>40	low	yes	excellent	no
→	31 .. 40	low	yes	excellent	yes
→	<=30	medium	no	fair	no
→	<=30	low	yes	fair	yes
→	>40	medium	yes	fair	yes
→	<=30	medium	yes	excellent	yes
→	31 .. 40	medium	no	excellent	yes
→	31 .. 40	high	yes	fair	yes
→	>40	medium	no	excellent	no

As show in the above diagram, we can make use of cyclic work distribution with multiple threads to be able to cover several rows of the dataset at a time. However, because Naïve Bayesian maintains counts of combinations of values and classes, this type of an approach will take a significant hit on the performance side as once each row has been processed for training, the classifier will need to go in and update the frequency count. This means that multiple threads might have to wait for their turn to update the frequency counts while another thread is doing the same. A better way to do this will be to capitalize on the conditional independence assumption of the Naïve Bayesian classifier and follow a Column-major parallelization approach.

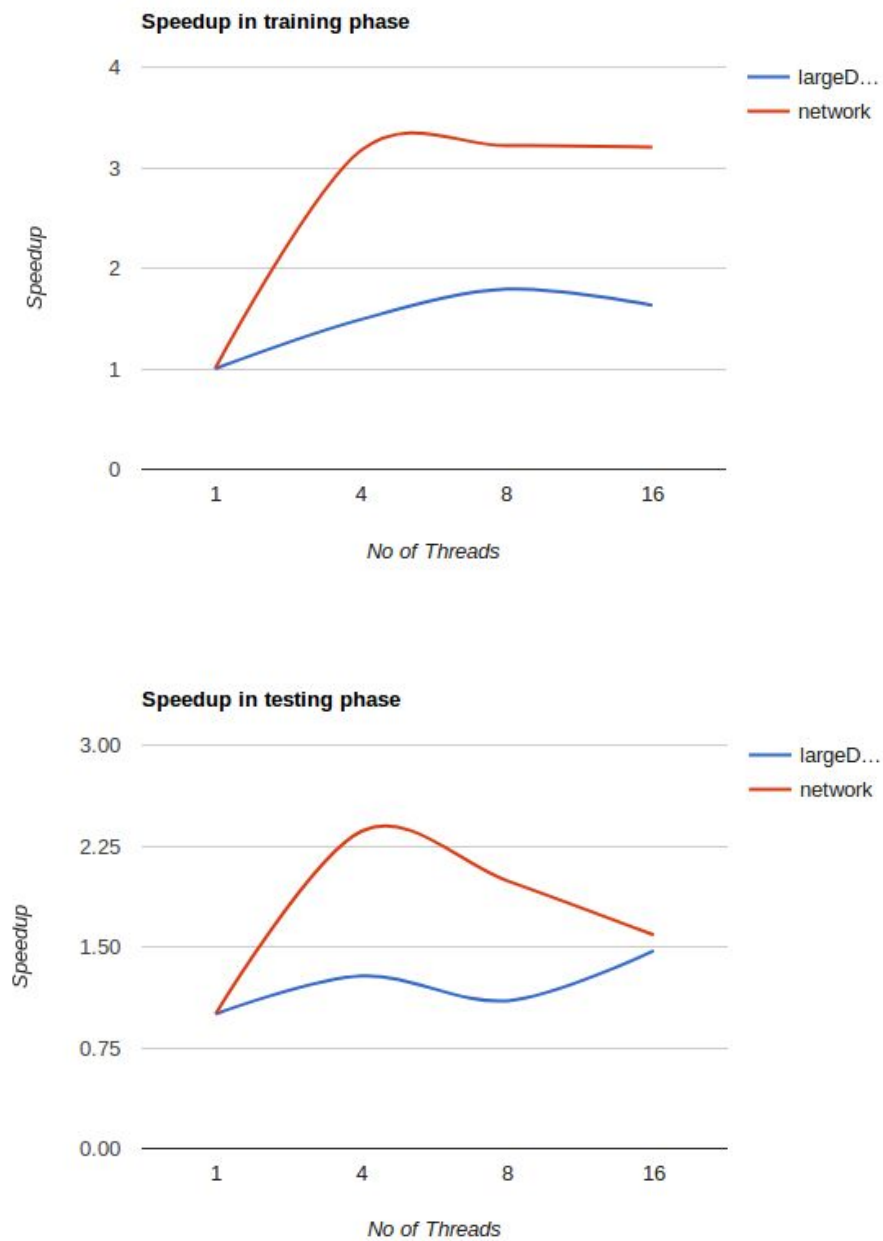


age	income	student	credit	buys
<=30	high	no	fair	no
<=30	high	no	excellent	no
31 .. 40	high	no	fair	yes
>40	medium	no	fair	yes
>40	low	yes	fair	yes
>40	low	yes	excellent	no
31 .. 40	low	yes	excellent	yes
<=30	medium	no	fair	no
<=30	low	yes	fair	yes
>40	medium	yes	fair	yes
<=30	medium	yes	excellent	yes
31 .. 40	medium	no	excellent	yes
31 .. 40	high	yes	fair	yes
>40	medium	no	excellent	no

We can apply both approaches to training as well as testing phases. These approaches can be applied to testing only if two critical assumptions are true:

1. We have all the training and testing data captured and stored
2. We are not making use of an incremental training model

PERFORMANCE COMPARISON IN TRAINING AND TESTING PHASE



CONCLUSION

We have explored and evaluated the Naïve Bayesian classifier as part of supervised learning. We have implemented a parallelized version of the classifier in OpenMP to understand the speedup that it provides.

This was implemented through attribute and record parallelization. The parallel implementation of this algorithm didn't result in an important increase in the speed of the naïve Bayesian algorithm.

PART 2

IMAGE CONVOLUTION

Aim

To implement a parallel version of the naive bayes classifier and compare the performance with the sequential version of naive bayes classifier and the incremental learning model version of the naive bayes classifier.

Introduction

We chose this topic because we thought it be a great way to practically test what we learnt in our Digital Image Processing class, while at the same time being a nice problem to parallelize.

We take an image in PNG (Portable Network Graphics) format, with height H pixels and width W pixels. Each pixel is composed of 3 channels for the values Red (R), Green (G) and Blue (B) with values between 0 and 255. Thus we represent the image in C using a three-dimensional arrangement: $\text{char image}[H][W][3]$. The application of a linear mask of blur consists of applying to each pixel the following mask of $M \times N$

$$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

In particular, if we consider a 3×3 mask, and what we are doing is replacing each pixel with the average the pixels covered by the mask. This process can be applied iteratively increasing the effect.

What is image convolution?

Convolution is the process of adding each element of the image to its local neighbors, weighted by the kernel. This is related to a form of mathematical convolution. For example, if we have two 3×3 matrices, the first a kernel, and the second an image piece, convolution is the process of flipping both the rows and columns of the kernel and then multiplying locally similar entries and summing. The element at coordinates $[2, 2]$ (the central element) of the resulting image would be a weighted combination of all the entries of the image matrix, with weights given by the kernel:

$$\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \right) [2, 2] = (i \cdot 1) + (h \cdot 2) + (g \cdot 3) + (f \cdot 4) + (e \cdot 5) + (d \cdot 6) + (c \cdot 7) + (b \cdot 8) + (a \cdot 9).$$

The other entries would be similarly weighted, where we position the center of the kernel on each of the boundary points of the image, and compute a weighted sum.

Kernel

In image processing, a kernel (or mask) is a small matrix. It is used for blurring, sharpening, embossing, edge detection, and more. This is accomplished by doing a convolution between a kernel and an image. In our test cases, we used (normalized) box blur kernel-

$$K = \frac{1}{M \times N} \times \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & 1 & \dots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \dots & 1 \end{bmatrix}$$

As the name indicates, it is used to blur an image.

CONVOLUTION ALGORITHM

```
for each image row in input image:
    for each pixel in image row:
        set accumulator to zero
        for each kernel row in kernel:
            for each element in kernel row:
                if element position corresponding* to pixel position then
                    multiply element value corresponding* to pixel value
                    add result to accumulator
                endif
            set output image pixel to accumulator
```

WHAT WE ARE PARALLELIZING

```
#pragma omp parallel for collapse(2)
for row = 0 to height:
    for col = 0 to width:
        convolution at (row, col) // threads perform this operation
```

We are essentially parallelizing each kernel accumulation(As this operation is completely independent for each set of neighbouring pixels) by dividing this task over all the available threads

We are using the default static scheduling which evenly assigns operations to each thread.

We used the collapse(2) option in order to set the granularity of operations. As a result, each thread will be assigned work in terms of convolution operations (the inner two for loops are not collapsed). Thus, the MxN number of convolution operations will be divided between the available threads.

Important private variables - loop indices, image convolution indices, rgb accumulators

Important shared variables - pass number, source 2d array, output 2d array, kernel

PERFORMANCE COMPARISON

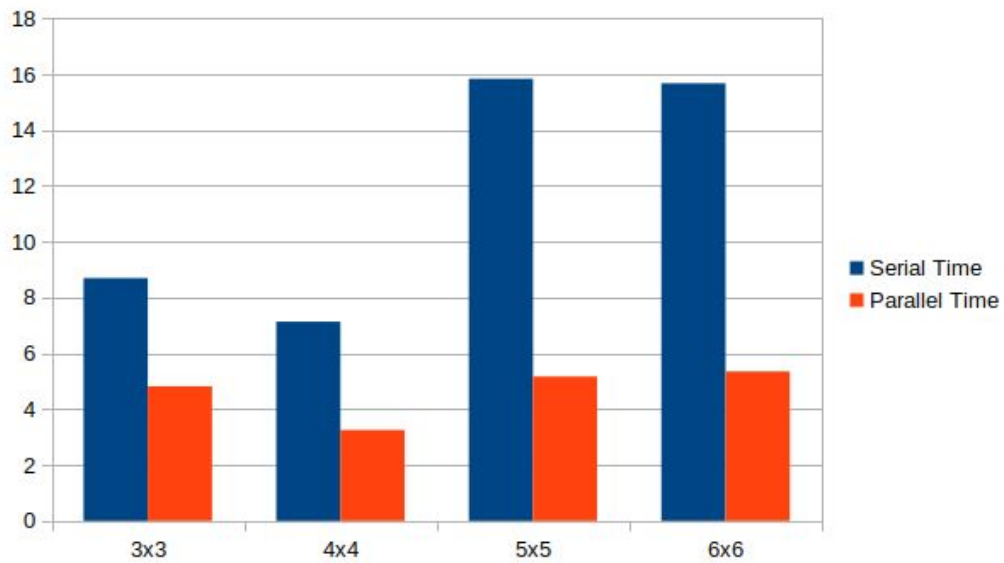
For each measure ,each cell has two times.

The time above is for sequential and the time below is for parallel

Changing filter size

Filter Size	Real	User	Sys
3 x 3	0m 8.696s	0m 8.584s	0m0.108s
	0m 4.818s	0m 14.060s	0m 0.144s
4 x 4	0m 7.132s	0m 7.024s	0m 0.108s
	0m 3.253s	0m 10.944s	0m 0.120s
5 x 5	0m 15.838s	0m 15.732s	0m 0.104s
	0m 5.167s	0m 27.652s	0m 0.140s
6 x 6	0m 15.674s	0m 15.576s	0m 0.096s

	0m 5.290s	0m 27.452s	0m 0.124s
--	-----------	------------	-----------



Changing number of threads

No of threads	Real	User	Sys
1	0m 7.646s	0m 7.544s	0m 0.096s
2	0m 4.924s	0m 7.856s	0m 0.092s
3	0m 4.080s	0m 8.264s	0m 0.108s

4	0m 3.616s	0m 8.564s	0m 0.100s
5	0m 3.927s	0m 10.364s	0m 0.112s
6	0m 3.857s	0m 11.804s	0m 0.196s
7	0m 3.401s	0m 11.912s	0m 0.140s
8	0m 3.753s	0m 13.124s	0m 0.172s

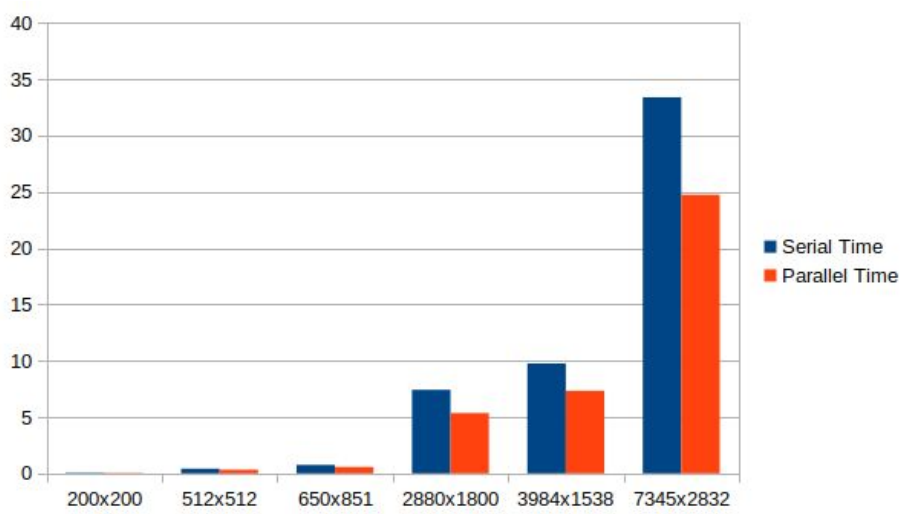
Changing Image Size

On a dual core hyperthreaded i5 processor (4 logical cores),Default no of threads = 4 .

Image sizes :

Image Size	Serial Time	Parallel Time
200x200	0.051	0.037
512x512	0.407	0.32
650x851	0.744	0.558
2880x1800	7.413	5.351

3984x1538 9.756 7.327



Blurring using smoothing filter

Filter used:

$$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Input Image:



Output Image:



Edge Detection filter

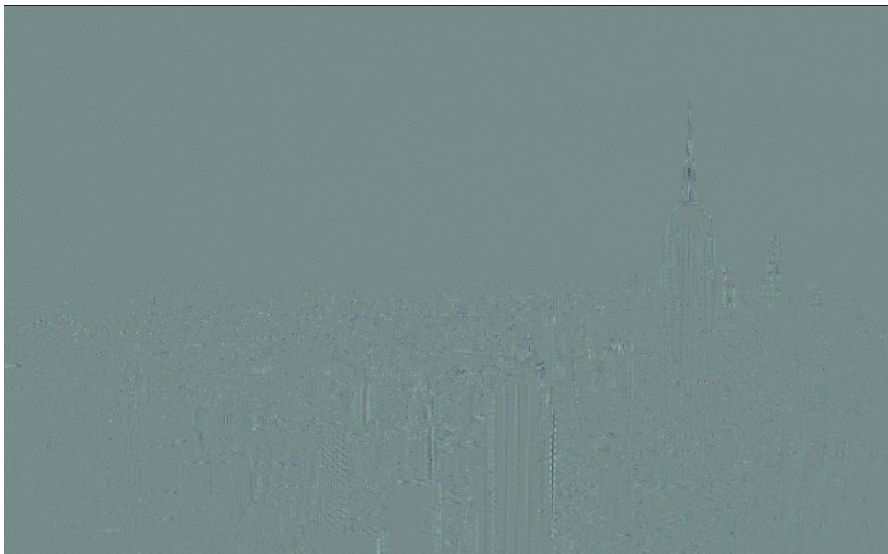
Filter used:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Input Image:



Output Image:



Selective Blurring

Input Image:



Ouput Image:



Conclusions

- When code is executed sequentially, the real and user times are (nearly) the same due to there being only one thread
- When code is run parallelly, the user time shoots up but the real time is less than that of the corresponding sequential execution (usually)
- When the filter size increases, the convolution operation takes longer to complete and hence execution time increases
- When the image size increases, there are simply more pixels, and hence iterations, so execution time increases
- When the number of threads available increase, the amount of time taken reduces at first but then saturates due to overhead costs