CS301: Assignment 3 Report

Team members:

- Mit Naria (201401448)
- Aayush Kapadia (201401407)

Hardware details:

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                32
On-line CPU(s) list:   0-31
Thread(s) per core:    2
Core(s) per socket:    8
Socket(s):             2
NUMA node(s):          2
Model name:            Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
CPU max MHz:           3200.0000
CPU min MHz:           1200.0000
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              20480K
NUMA node0 CPU(s):     0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31
CPU Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe sysca
ll nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xto
pology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl v
mx smx est tm2 ssse3 fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe
popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm ida arat epb
pln pts dtherm tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi
1 avx2 smep bmi2 erms invpcid cqm xsaveopt cqm_llc cqm_occup_llc
```

Q1: Image warping

Explanation of algorithm:
In this algorithm , we simply rotate image by theta which is defined from relation of distance from center of image and angle made with respect to center of image. Suppose (centerX,centerY) is center of image and (x,y) is the point in the output image. (newX,newY) correspond to point in input image.

```
radius = sqrt((y-centerY)*(y-centerY)+(x-centerX)*(x-centerX));
theta = (radius/2.0) * (PI/180);
newX = (int)( (cos(theta)*(y-centerY)) - (sin(theta)*(x-centerX)) + centerY );
newY = (int)( (cos(theta)*(x-centerX)) + (sin(theta)*(y-centerY)) + centerX );
```

Input Image

Output Image

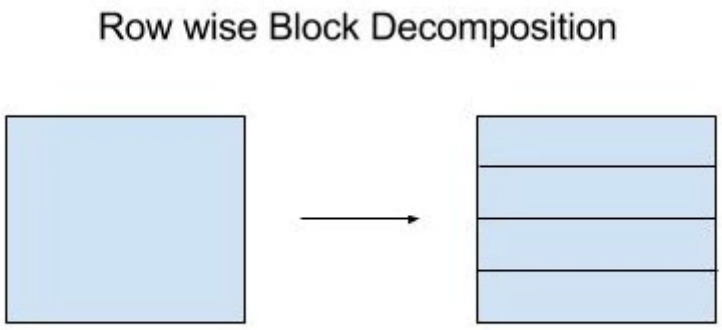
Algorithm:

```
for(loopx=0;loopx < rows;++loopx)
    {
        for (loopy=0; loopy < columns; ++loopy)
        {
            radius = sqrt((loopy-centerY)*(loopy-centerY)+(loopx-centerX)*(loopx-centerX));
            theta = (radius/2.0) * (PI/180);

            newX = (int)( (cos(theta)*(loopy-centerY)) - (sin(theta)*(loopx-centerX)) + centerY );
            newY = (int)( (cos(theta)*(loopx-centerX)) + (sin(theta)*(loopy-centerY)) + centerX );
```

```
10
11              if( (newX < 0) || (newX >= rows) || (newY < 0) || (newY >= colum
ns) )
12                  continue;
13
14              bilinearlyInterpolate(inPixel,outPixel,columns,rows, newX, newY,
loopx,loopy);
15          }
16      }
```

Parallel version of this algorithm makes use of idea called “Rowwise Block Striped Decomposition” . In this idea, matrix is divided into blocks of certain rows. As mentioned below:



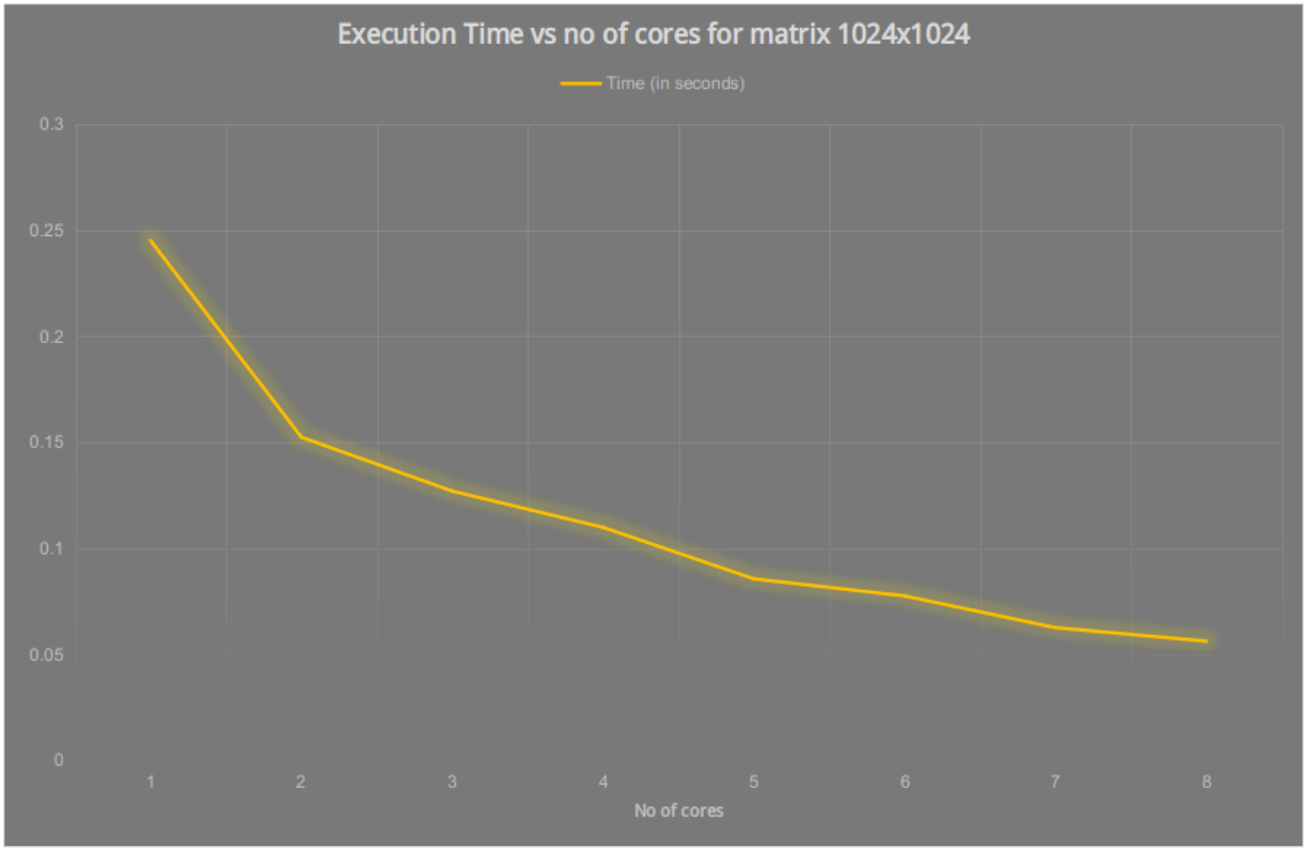
Also, this can be done by just putting line mentioned below before outer for loop.

```
1  #   pragma omp parallel for private (loopy,newX,newY,theta,radius) num_threa
ds(noOfThreads)
2
```

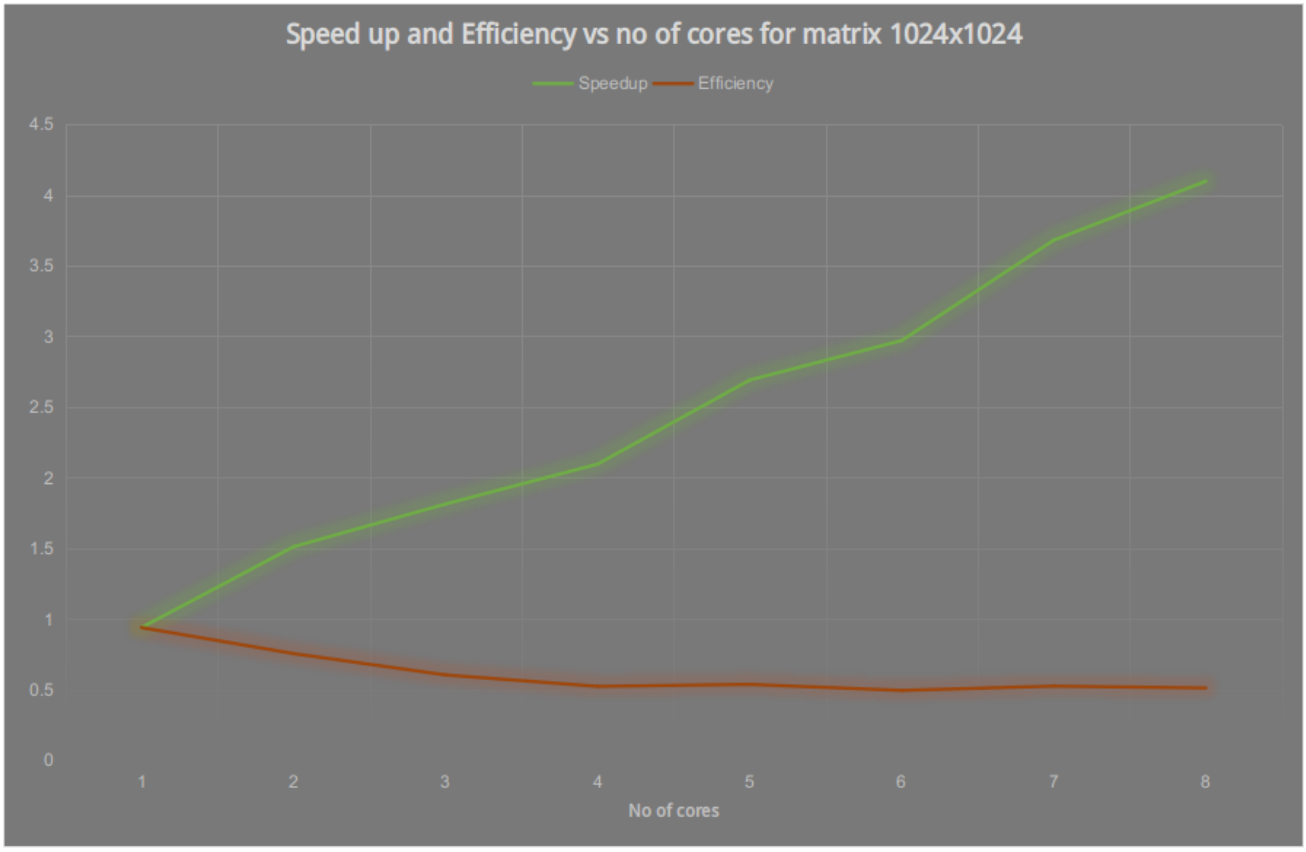
Results:
Matrix size 1024x1024
Sequential time = 0.23034079791978 s
Parallel overhead time = 0.01477 s
Comparison of serial and parallel output: (output files are available in .zip file)

```
1  username@ubuntu14:~$ diff output_twisted_lenna_serial.ppm output_twisted_len
na_parallel.ppm
2  username@ubuntu14:~$
```

No of cores	Time (in seconds)	Speedup	Efficiency
1	0.24511976	0.939707177	0.939707177
2	0.152404007	1.51138282	0.75569141
3	0.12699882	1.813723917	0.604574639
4	0.109895526	2.095997957	0.523999489
5	0.085623149	2.690169669	0.538033934
6	0.07756319	2.96971794	0.49495299
7	0.062603333	3.67936956	0.525624223
8	0.056229029	4.096474768	0.512059346



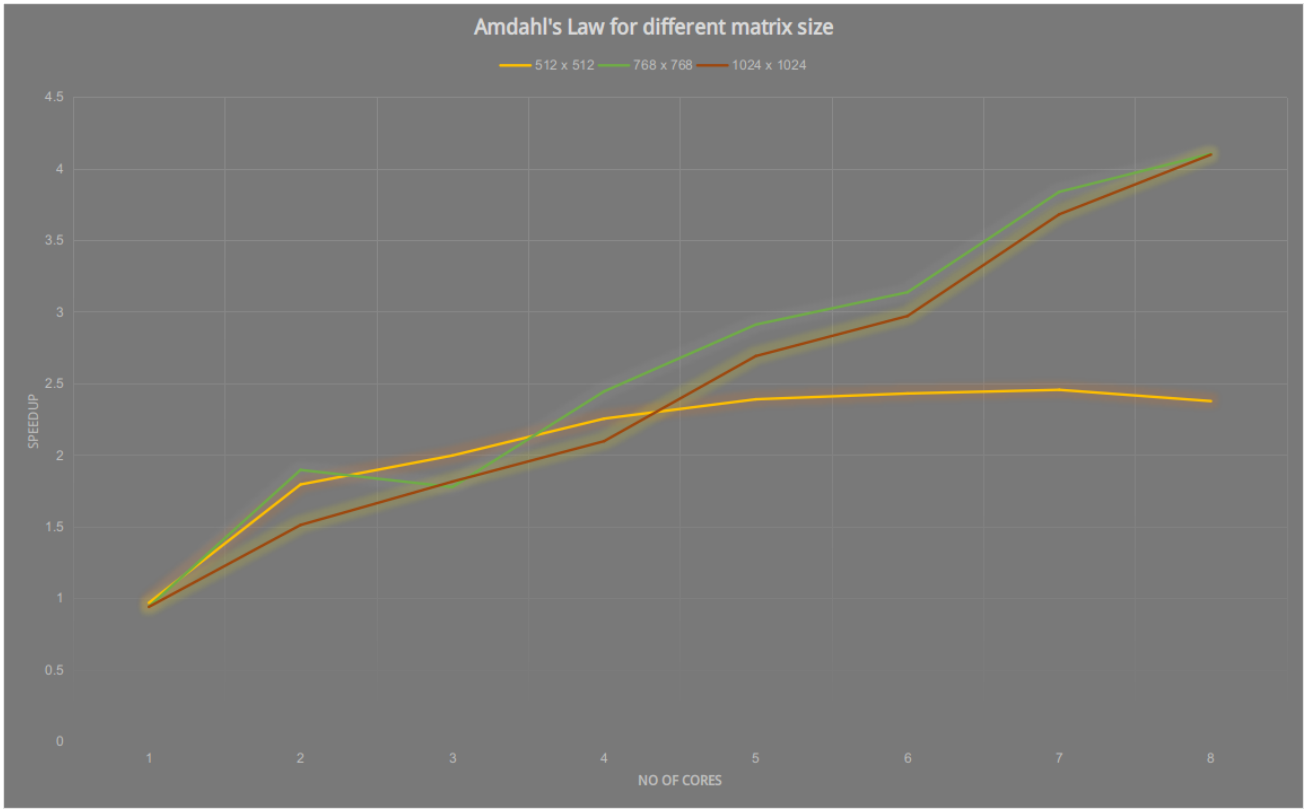
As you can see execution time is decreasing as no of cores increases.



Speedup graph is increasing as no of cores increasing.
This is because this code doesn't contain any serial component.

Analysis:

1. **Amdahl's effect:**



Amdahl's effect suggest that as problem size increases , speedup increases. (Effect is visible after no of cores = 5).

2. **Karp-Flatt metric:**

Let's calculate Karp-Flatt metric.

No of cores	Karp-Flatt metric
2	0.3233
3	0.3270
4	0.3028
5	0.2146
6	0.2040
7	0.1504
8	0.13612

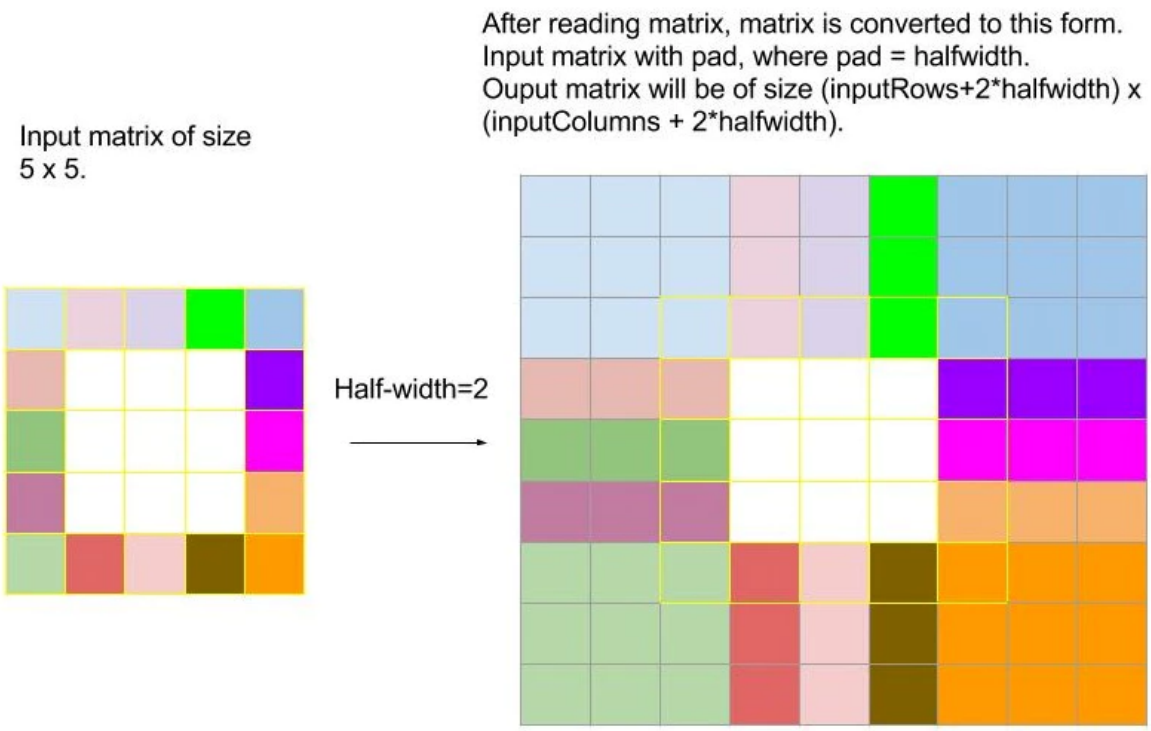
As you can see that Karp-Flatt metric is decreasing thus indicating "No bottleneck of parallel overhead" and "no bottleneck of serial computation" for this code.

Q2: Image Filtering

Explanation of algorithm:

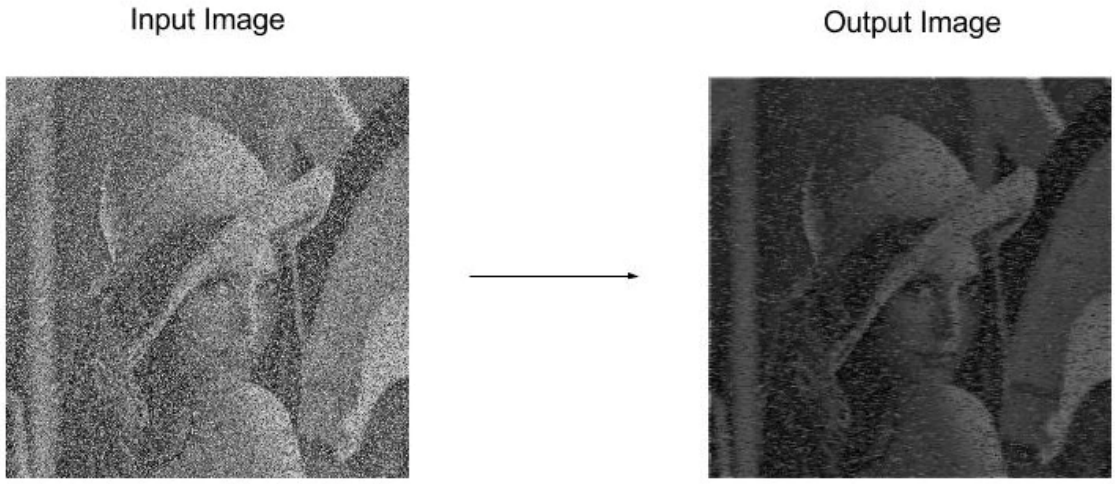
Filtering is used for removing the unnecessary noise from image. There are various types of filtering algorithms out there. Example linear filtering algorithms like mean filtering , median filtering etc. Here, we are doing median filtering. In this filtering is every pixel value is calculated from median of surrounding pixel's values. If surrounding pixel goes out of image boundary then nearest neighbor is taken. Here, surrounding neighborhood can be taken in form of square as well as rectangle. However for simplicity in this case we are considering only square neighborhoods. So let us now define "how we identify neighborhoods". Before considering neighborhood we should understand the concept of half-width. Half-width means number of elements in one direction that are to be considered as neighbors. So total width will be (2*half-width + 1). So total area of neighborhood square will be equal to ((2*half-width + 1)^2).

Neighbors and its explained in below diagram. Each color represent a unique value. Neighbors that are out of boundary points in input matrix , have value of nearest neighbor. Yellow border line shows original matrix.



One thing to notice that output matrix have same dimensions of input matrix. Above bigger matrix is created for ease of calculation. (It is intermediate matrix.)

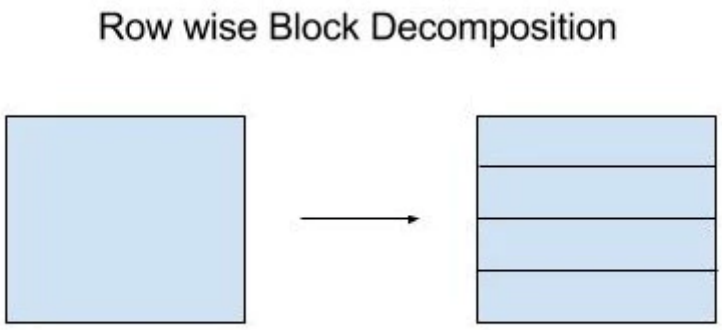
We can calculate median by several different algorithms like doing first sorting then picking the middle element. We can use either insertion sort or bubble sort or any other sort. It will not effect much because total elements will just be 49 in case of half_width = 3.



Algorithm:

```
for(loopx=0;loopx < rows;++loopx)
    {
        for (loopy=0; loopy < columns; ++loopy)
        {
            findMedian(inPixel, outPixel, columns, rows, loopx, loopy, halfW
idth);
        }
    }
```

As mentioned above, this algorithm also uses “Rowwise Block Striped Decomposition”.



Also, this can be done by just putting line mentioned below before outer for loop.

```
#   pragma omp parallel for private(loopy) num_threads(noOfThreads)
```

Results:

Half width=3

Matrix size = 512 x 512.

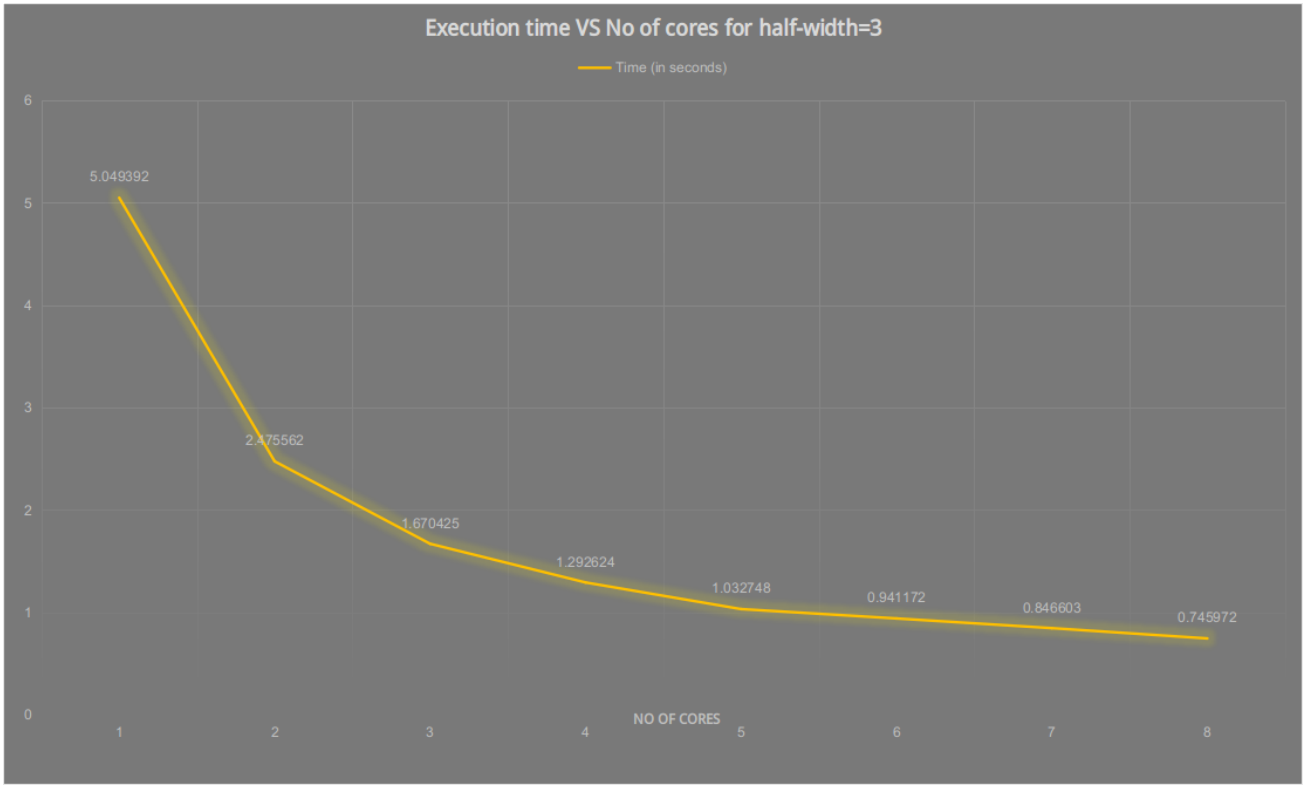
Sequential time = 4.932049 s

Parallel overhead time = 0.117343 s

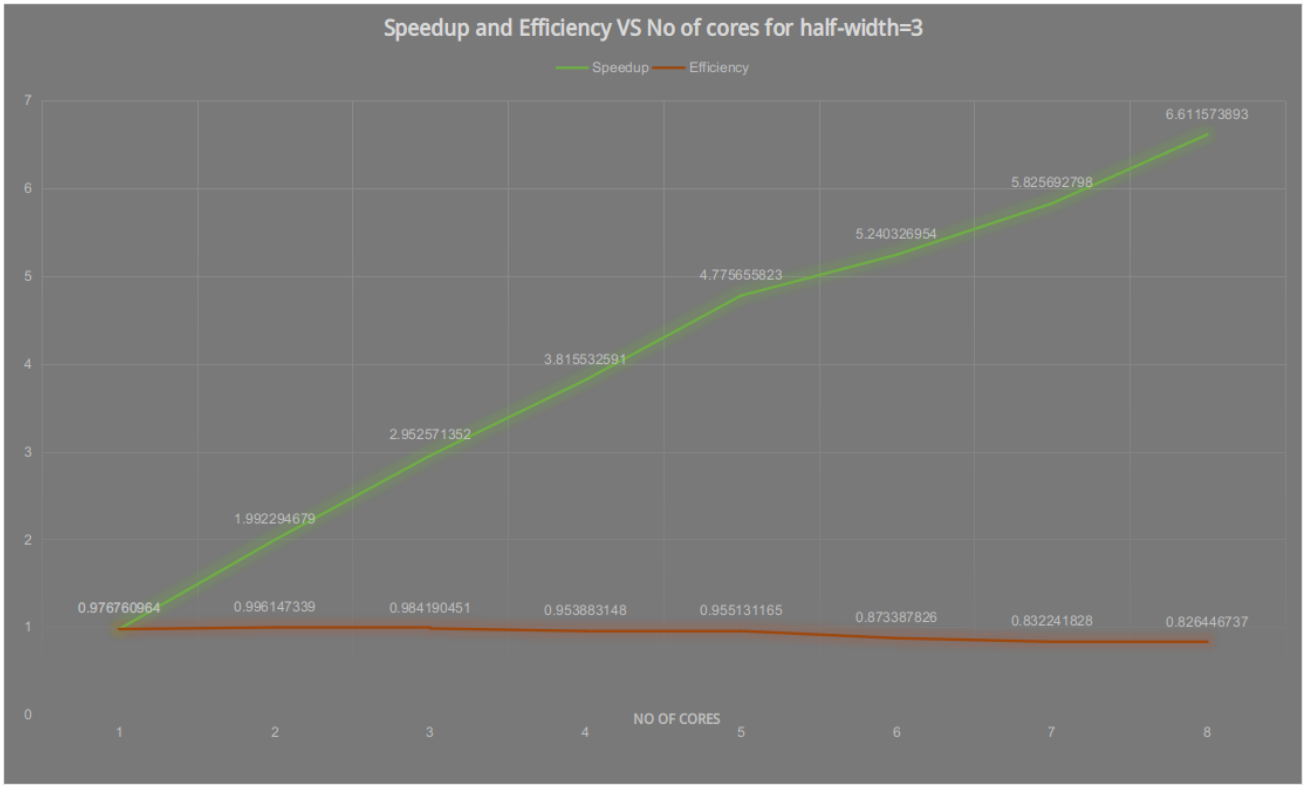
Comparison of serial and parallel output: (output files are available in .zip file)

```
username@ubuntu14:~$ diff output_image_filtering_parallel.ppm output_image_f
iltering_serial.ppm
username@ubuntu14:~$
```

No of cores	Time (in seconds)	Speedup	Efficiency
1	5.049392	0.976760964	0.976760964
2	2.475562	1.992294679	0.996147339
3	1.670425	2.952571352	0.984190451
4	1.292624	3.815532591	0.953883148
5	1.032748	4.775655823	0.955131165
6	0.941172	5.240326954	0.873387826
7	0.846603	5.825692798	0.832241828
8	0.745972	6.611573893	0.826446737



As you can see execution time is decreasing as no of cores increases.

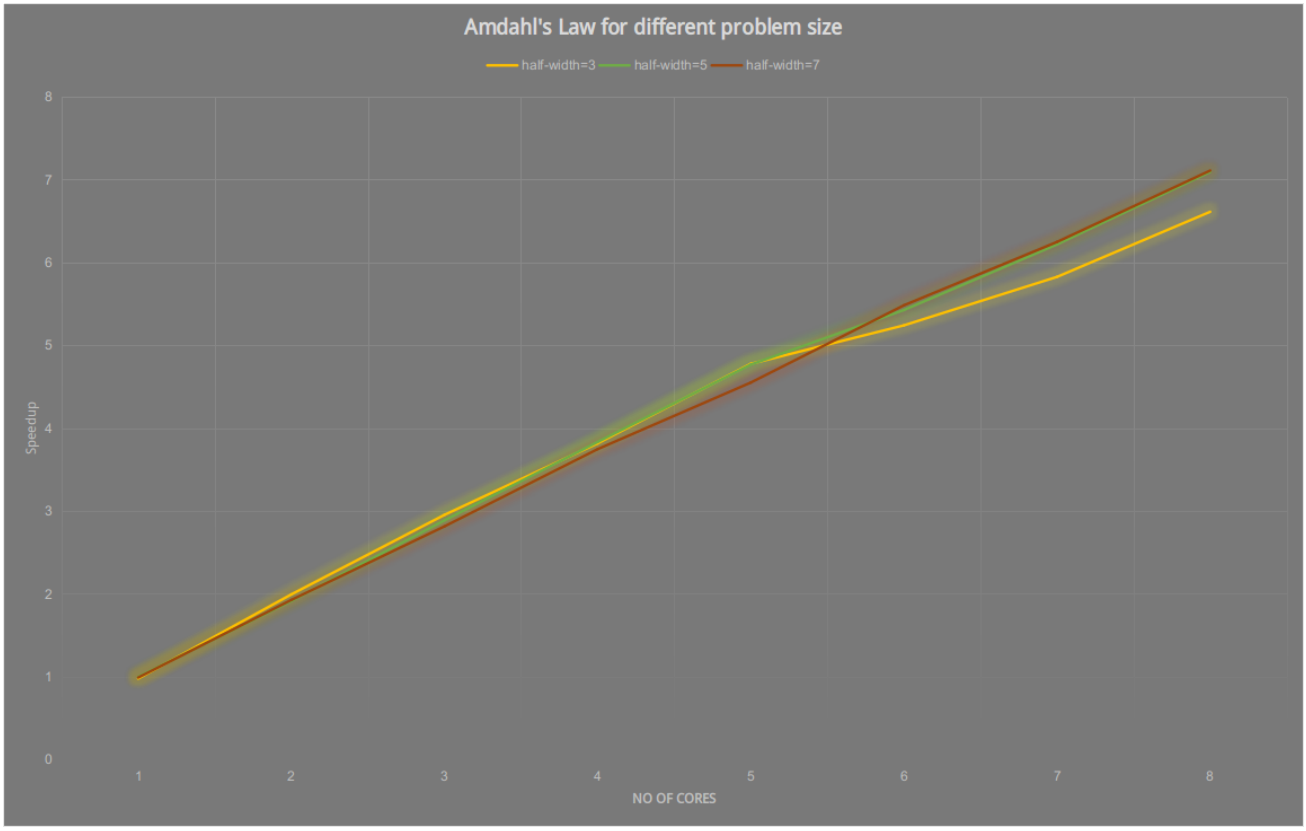


Speedup graph is increasing as no of cores increasing.

This is because this code doesn't contain any serial component.

Analysis:

1. **Amdahl's effect:**



Amdahl's effect suggest that as problem size increases , speedup increases. (Effect is visible after no of cores = 6). Also, this graph suggest that speedup is essentially same for different problem size that means there is no "serial code bottleneck" for this parallel algorithm.

2. **Karp-Flatt metric:**

Let's calculate Karp-Flatt metric.

No of cores	Karp-Flatt metric
2	0.00386
3	0.00803
4	0.01611
5	0.01174
6	0.02899
7	0.03359
8	0.02999

As you can see that Karp-Flatt metric is slightly increasing or remaining constant. Thus , indicating "no serial bottleneck for parallel code."