# Assignment 1 - 2D Filter in OMP
Sanyam Gupta (sanyamgu)
10/17/2021

## Overview
The task here is to implement a 2D filter using OMP so that the calculation can be spread across multiple cores. Analysis of the algorithm is also provided by repeating the calculation for 3 different matrix sizes. The speedup for [2, 4, 6, 8, 10] cores has been analysed. The reported data has been collected from JobID - **8130740.** The slurm.sh and 8130740.stdout files are included with the submission.

## Algorithm for applying 2D filter
The overall step is broken down into 2 parallel computations.

1. Apply the filter in parallel and store the result in an auxiliary array.

```
std::vector<float> aDash(n*m);
#pragma omp parallel for shared(K,A,n,m)
for (int index=0; index< n*m; index++){
    // if corner indices, value is original
    if ((index/m == 0) | (index/m == n-1) | (index%m == 0) || (index%m == n-1)){
        aDash[index] = A[index];
    }
    else {
        aDash[index] = transformation(index, n, m, K, A);
    }
}
```

**Figure 1 -** Applying filter and store result in aDash

2. Copy the values from the auxiliary array to the original array.

```
#pragma omp parallel for shared(A, aDash)
for (int index=0; index<n*m; index++){
    A[index] = aDash[index];
}
```

**Figure 2 -** Updating original array(A) with Adash

## Ascertaining performance
The performance on 3 different arrays sizes, across 6 cores has been analysed. The base case i.e. without OMP support has also been reported. For each measurement, the experiment was repeated 5 times.

After discarding the maximum and minimum value, an average of **runtime** across 3 readings is shown below. The **speedup** data is also reported.

| | Times (s) | | | CORES | | | |
|---|---|---|---|---|---|---|---|
| | | 1 (without omp) | 2 | 4 | 6 | 8 | 10 |
| | 10000 * 20000 | 5.395 | 3.125 | 1.911 | 1.479 | 1.24 | 1.069 |
| SIZE (n*m) | 10000 * 40000 | 10.784 | 6.057 | 3.547 | 2.792 | 2.268 | 2.012 |
| | 10000 * 60000 | 16.184 | 9.033 | 5.092 | 4.014 | 3.368 | 2.885 |

**Figure 3** - execution times

| | Speedups | | | CORES | | | |
|---|---|---|---|---|---|---|---|
| | | 1 (without omp) | 2 | 4 | 6 | 8 | 10 |
| | 10000 * 20000 | 1 | 1.726 | 2.823 | 3.648 | 4.351 | 5.047 |
| SIZE (n*m) | 10000 * 40000 | 1 | 1.78 | 3.04 | 3.862 | 4.755 | 5.36 |
| | 10000 * 60000 | 1 | 1.792 | 3.178 | 4.032 | 4.805 | 5.61 |

**Figure 4 -** Speedup values

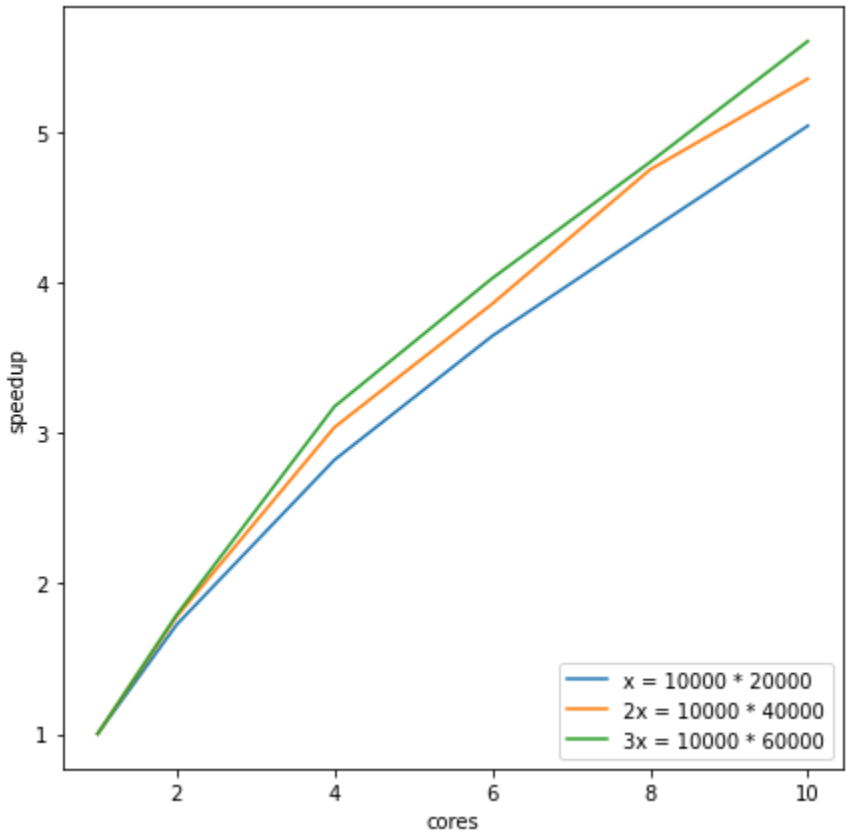


**Figure 5 -** Plot for Speedup Values

## Analysis of the Data

- **Strong Scaling** - For the same problem size (n) as the number of processing(p) elements decreases the runtime decreases. Hence the algorithm is strongly scalable.
- **Weak Scaling** - The runtime is also preserved across n/p values. Also a strongly scalable algorithm is also weakly scalable.