Assignment
2D Filter in OMP

Akash Sharma
UBIT-as475
as475@buffalo.edu
Department of Computer Science and Engineering
University at Buffalo
Buffalo, NY 14214

17 October 2021

1 Introduction

This assignment is to implement and analyze efficient OMP program for fast application of 2D filters. 2D filters are frequently used in image and signal processing, and hence their efficient implementations are highly desired.

2 Code used to implenment the 2D filter

```
include <iostream>
include <omp.h>
include <vector>

    void filter_2d(int n, int m, const std::vector<float> K, std::vector<float>&A)

    std::vector<float> A1=A;
int x=0;
int arr[(m-2)*(n-2)] =0;

    pragma omp parallel default(shared) {

    int mul[27]=-(m+1),-m,-(m-1),-(m+1),-m,-(m-1),-(m+1),-m,-(m-1),-1,0,1,-
1,0,1,-1,0,1,(m-1),m,(m+1),(m-1),m,(m+1),(m-1),m,(m+1);
int kmul[27]=0,3,6,1,4,7,2,5,8,0,3,6,1,4,7,2,5,8,0,3,6,1,4,7,2,5,8;
```

```
pragma omp for schedule(auto)
for(int rowA=m; rowA<=(n*m-(m+1)); rowA++)
{

if(rowA{
int k=0;

for(int rowK=0; rowK<27; rowK++)
{
arr[x]=arr[x]+A[rowA+mul[k]]*K[kmul[rowK]];

k++;
A1[rowA]=arr[x];
}

x++;
} } }
A=A1;
}
```

3 Code run for different threads and row(n)and column(m) value

3.1 Time in code run -1

P\n*m	10k*10k	10k*20k	20k*20k	20k*30k	30k*40k
1	8.97934	17.7819	35.3718	53.0395	105.838
2	5.12753	11.9849	23.0813	33.6683	66.5917
4	4.62232	9.62425	17.08	26.3669	52.1229
6	4.26495	8.71415	16.3341	25.844	49.0067
8	4.17458	8.01187	15.0097	21.9733	49.7077
10	3.87094	7.44439	14.4372	21.994	46.1046
12	3.59286	7.11827	14.6282	21.8393	44.2568

Figure 1: No of Threads with n * m in run 1

3.2 Time in code run -2

P\n*m	10k*10k	10k*20k	20k*20k	20k*30k	30k*40k
1	9.10758	17.7403	35.4969	53.007	107.977
2	5.76331	9.93715	20.2284	41.5443	67.2121
4	4.3643	8.85589	18.633	28.2217	52.3408
6	4.42849	8.14107	16.0779	24.5787	49.501
8	4.45014	7.50183	15.4574	23.613	43.9134
10	4.14964	7.39695	15.9996	22.2064	43.2498
12	3.83937	6.99479	13.5451	20.3409	39.7207

Figure 2: No of Threads with n * m in run 2

3.3 Time in code run -3

P\n*m	10k*10k	10k*20k	20k*20k	20k*30k	30k*40k
1	9.28294	18.0634	35.6231	53.3299	105.97
2	5.0779	9.91351	24.2616	36.1143	67.7898
4	5.24132	9.37009	17.5785	27.804	52.7636
6	4.67739	9.33315	16.4122	24.0426	50.833
8	3.8845	7.33225	15.5873	21.4884	41.9979
10	4.0031	7.90914	14.8112	22.8264	44.0951
12	3.51311	7.11169	13.7722	20.83	39.14

Figure 3: No of Threads with n * m in run 3

3.4 Time in code run -4

P\n*m	10k*10k	10k*20k	20k*20k	20k*30k	30k*40k
1	9.26675	18.0978	35.5927	54.2831	106.851
2	5.95225	11.4998	23.6656	33.7223	62.8829
4	4.7526	8.83569	18.0872	27.7971	53.1255
6	4.3111	7.75532	16.4762	26.6035	48.4503
8	3.88837	7.51803	14.4769	20.2394	46.6398
10	4.10896	8.12078	14.6345	21.9961	47.4268
12	3.68619	7.69742	13.2945	20.6119	43.0233

Figure 4: No of Threads with n * m in run 4

3.5 Time in code run -5

P\n*m	10k*10k	10k*20k	20k*20k	20k*30k	30k*40k
1	9.26675	18.0978	35.5927	54.2831	106.851
2	5.95225	11.4998	23.6656	33.7223	62.8829
4	4.7526	8.83569	18.0872	27.7971	53.1255
6	4.3111	7.75532	16.4762	26.6035	48.4503
8	3.88837	7.51803	14.4769	20.2394	46.6398
10	4.10896	8.12078	14.6345	21.9961	47.4268
12	3.68619	7.69742	13.2945	20.6119	43.0233

Figure 5: No of Threads with n * m in run 5

4 Tables for Time, Speed Up, Efficiency

4.1 Average Time of all 5 code run

P\n*m	10k*10k	10k*20k	20k*20k	20k*30k	30k*40k
1	9.117802	17.94712	35.49948	53.52044	106.5128
2	5.521298	10.95531	22.77458	36.04398	66.54486
4	4.80788	9.09056	18.15564	27.54174	52.96986
6	4.379838	8.44578	16.26496	25.183	49.20798
8	4.064602	7.517766	15.1677	21.88518	46.00036
10	3.975762	7.769342	14.90896	22.5493	44.95548
12	3.672884	7.261442	13.92488	20.96778	41.03238

Figure 6: Average Time

Above table indicates that with increase in no of threads decrease in time for particular row and column. By noticing the time for particular row and column for example in 10k*10k(n*m) for thread 2 time is nearly half this is come under the strong scaling means with threads time should reduce, but with increase in no of above 2 time is not significantly decreasing in comparison to increase in no of threads or processors this is because of overheads in communication cost increases and also there is always a sequential part which always restrict time to decrease.

As explained in Amdahl's law, there always we sequential part of code. It is nearly impossible to parallelize all code.

For 10*10K and 20k*20k(n*m), if noticed the time for threads 2 and 4 respectively is nearly which indicates that if increase in no of threads and (n*m) ratio should be same, which indicates the weak scaling.

4.2 Speedup for increasing Processors

P\n*m	10k*10k	10k*20k	20k*20k	20k*30k	30k*40k
1	1	1	1	1	1
2	1.651387	1.638212	1.558733	1.484865	1.600616
4	1.896429	1.974259	1.955287	1.943248	2.010819
6	2.081767	2.124981	2.182574	2.125261	2.164543
8	2.243221	2.387294	2.340466	2.445511	2.315478
10	2.293347	2.309992	2.381084	2.373486	2.369295
12	2.482464	2.471564	2.549356	2.552509	2.595823

Figure 7: Speedup

Above Table indicates speedup with increase in no of threads. Speedup is increasing with increasing number of threads but after certain threads its nearly constant because of overheads.

4.3 Efficiency for increasing Processors

P\n*m	10k*10k	10k*20k	20k*20k	20k*30k	30k*40k
1	1	1	1	1	1
2	0.825694	0.819106	0.779366	0.742432	0.800308
4	0.474107	0.493565	0.488822	0.485812	0.502705
6	0.346961	0.354163	0.363762	0.35421	0.360757
8	0.280403	0.397882	0.390078	0.407585	0.385913
10	0.229335	0.230999	0.238108	0.237349	0.23693
12	0.206872	0.205964	0.212446	0.212709	0.216319

Figure 8: Efficiency

Above table indicates the efficiency of parallel code at different threads. Efficiency value always lies between the 0 and 1. Greater the efficiency better the parallel code. Efficiency also restricted because of overheads and the squential part.

5 Graphs Time, Speed Up, Efficiency

5.1 Time for with increasing row and column

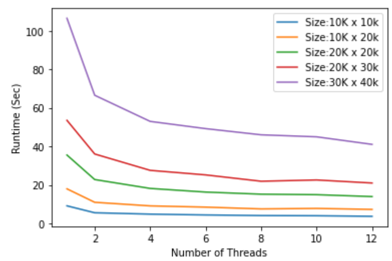

Figure 9: Time

5.2 Speedup with increasing row and column

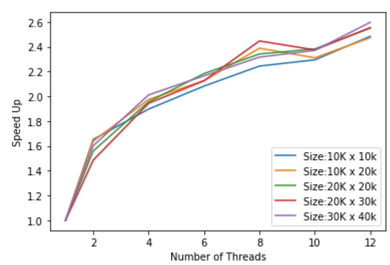

Figure 10: Speedup

5.3 Efficiency with increasing row and column

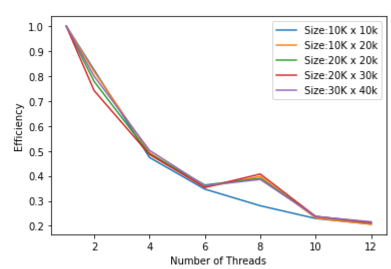

Figure 11: Efficiency

6 Conclusion

By observing above table and graph, code is showing strong scaling property with overheads due to communication cost, as increases in threads there is no significant decrease in time in compare to increase number of threads.