Introduction to Parallel and Distributed Processing

Assignment 1

Author : Kavitha Elizebeth Varughese
UBID : 50560694

Overview

In this assignment, we design and implement an efficient OMP program for the fast application of 2D filters. We will benchmark our solution to understand the scalability and performance of our parallel model over a sequential model.

Problem Description

Given a 2D image A of dimensions $(n \times m)$, perform 2D filtering for each pixel A_{ij}. Our filter will be a convolution between a $3x3$ kernel K and a $3x3$ submatrix A centered at A_{ij}.

Our focus will be on a slightly modified version of the problem. Let $A_{i-1:i+1, j-1:j+1}$ be the $3x3$ submatrix of A. We will first compute the matrix product $Z = A_{i-1:i+1, j-1:j+1} \times K$, followed by the summation of all the elements in the resultant Z. So, we can denote filtered $A_{i,j}$ as $\sum_{a,b} Z_{a,b}$.

The values of the first and last columns and rows will remain untouched; no filter will be applied to them.

Proposed Solution

Our goal is to design and implement a model using OpenMP, to parallelize the computation and achieve enhanced performance.

Underlying Mathematics

Following the direct computation requires the allocation of memory to save the resultant matrix before finding the summation of the elements to compute A_{ij}. We will be attempting to simplify the computation and check if the operations can done in-place.

Given the following matrices

$$A = \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix}$$

$$K = \begin{bmatrix} K_{00} & K_{01} & K_{02} \\ K_{10} & K_{11} & K_{12} \\ K_{20} & K_{21} & K_{22} \end{bmatrix}$$

The matrix multiplication will result as the following

$$A \times K = \begin{bmatrix} A_{00} \cdot K_{00} + A_{01} \cdot K_{10} + A_{02} \cdot K_{20} & A_{00} \cdot K_{01} + A_{01} \cdot K_{11} + A_{02} \cdot K_{21} & A_{00} \cdot K_{02} + A_{01} \cdot K_{12} + A_{02} \cdot K_{22} \\ A_{10} \cdot K_{00} + A_{11} \cdot K_{10} + A_{12} \cdot K_{20} & A_{10} \cdot K_{01} + A_{11} \cdot K_{11} + A_{12} \cdot K_{21} & A_{10} \cdot K_{02} + A_{11} \cdot K_{12} + A_{12} \cdot K_{22} \\ A_{20} \cdot K_{00} + A_{21} \cdot K_{10} + A_{22} \cdot K_{20} & A_{20} \cdot K_{01} + A_{21} \cdot K_{11} + A_{22} \cdot K_{21} & A_{20} \cdot K_{02} + A_{21} \cdot K_{12} + A_{22} \cdot K_{22} \end{bmatrix}$$

$$\sum_{a,b} Z_{a,b} = (A_{00} + A_{10} + A_{20}) \cdot (K_{00} + K_{01} + K_{02}) + (A_{01} + A_{11} + A_{21}) \cdot (K_{10} + K_{11} + K_{12}) + (A_{02} + A_{12} + A_{22}) \cdot (K_{20} + K_{21} + K_{22})$$

For our problem, the submatrix A will keep changing while the $3x3$ kernel remains the same throughout. Thus, we can rewrite the above equation as follows.

$$\sum_{a,b} Z_{a,b} = \left(A_{i-1,j-1} + A_{i,j-1} + A_{i+1,j-1}\right) \cdot (K_0) + \left(A_{i-1,j} + A_{i,j} + A_{i+1,j}\right) \cdot (K_1) + \left(A_{i-1,j+1} + A_{i,j+1} + A_{i+1,j+1}\right) \cdot (K_2)$$

where,

$$K_0 = (K_{00} + K_{01} + K_{02})$$

$$K_1 = (K_{10} + K_{11} + K_{12})$$

$$K_2 = (K_{20} + K_{21} + K_{22})$$

This simplified computation is crucial as it enables in-place computation, preventing unnecessary and time-consuming allocation of extra memory.

Streamlined Operations for Parallelization

1. We first compute the constant values K_0, K_1 and K_2. We could run this in parallel, but as it's a constant time operation, it's not needed. *[APPENDIX 1.1]*

2. We perform column-wise partial summation in parallel on matrix A. That is, for each column, we will be computing $A_{i-1,j} + A_{i,j} + A_{i+1,j}$ and store the result in $A_{i,j}$. Note that the value on $A_{i,j}$, is needed for the computation for the next iteration within the column ($A_{i,j} + A_{i+1,j} + A_{i+2,j}$). So, we will be using a temporary variable to store the hold the original value of $A_{i,j}$, to be used in the subsequent iteration. *[APPENDIX 1.2]*

   It is also to be noted that the first and the last column will be overwritten in this step. So, we will store a copy of the first and last columns. This extra memory allocation is inevitable. But, to save time, we can perform this operation in parallel. *[APPENDIX 1.3]*

3. Finally, we perform row-wise operations to compute ($Partial\_sum \cdot K$) in parallel on matrix A. That is, for each row, we will be computing $A_{i,j-1} \cdot K_0 + A_{i,j} \cdot K_1 + A_{i,j+1} \cdot K_2$ and store the result in $A_{i,j}$. Note that, just like in the column wise operation, the value on $A_{i,j}$ is needed for the computation for the next iteration within the row ($A_{i,j} \cdot K_0 + A_{i,j+1} \cdot K_1 + A_{i,j+2} \cdot K_2$). So, we will be using a temporary variable to store the hold the original value of $A_{i,j}$, to be used in the subsequent iteration. *[APPENDIX 1.4]*

   We don't will not be performing this operation on the first and last rows.

4. Once we have all the values in place in matrix A, we will simply overwrite the first and last columns of A with the original values

Time Complexity

For simplicity, we will be considering matrix A to be a square matrix of dimensions $n \times n$.

Sequential

In the sequential implementation using the same simplified computation,
1. Making the copy of the first and last columns will take $O(n)$.
2. Performing column wise operation will consists of two for-loops to iterate through each row in each column. This will take $O(n^2)$.
3. Similarly, the row wise operation will also take $O(n^2)$.

Thus, the sequential solution will have a time complexity of $O(n^2)$

Parallel

Given p processors with $p <<< n$,

1. Making the copy of the first and last columns takes ***O(n/p)***.
2. When performing column wise summations, ***p*** processors will perform on ***p*** columns at a time and it will iterate through all the rows in the column. Thus, the time complexity would be $O(n^2/p)$
3. Similarly, the row wise operation will also take $O(n^2/p)$.

Thus, the parallel solution should ideally have a time complexity of $O(n^2/p)$
And, if $p = n$, it would be $O(n)$.

Performance Analysis

System Specification

Hostname	cpn-v05-38.core.ccr.buffalo.edu
Number of Cores	56
CPU	Intel(R) Xeon(R) Gold 6330 CPU @ 2.00GHz
Operating System	Ubuntu 24.04.1 LTS
Compiler	g++ (Gentoo 10.4.0 p5) 10.4.0
L1d cache	2.6 MiB

Input Specification

Input values of n and p are taken while maintain n^2/p ratio.

n = (100 400 1600 6400 25600 102400)
p = (1 2 4 8 16 32)

Results

N\P	1	2	4	8	16	32
100	3.3363E-05	0.000210158	0.000277619	0.000432916	0.000686351	0.00118802
400	0.000407175	0.000313848	0.000461149	0.000452958	0.000666819	0.00129273
1600	0.00881699	0.00627769	0.00527061	0.00387837	0.00250986	0.00203413
6400	0.388235	0.197595	0.102143	0.0530067	0.0288478	0.0177588
25600	11.3098	6.7754	3.20043	1.57889	0.882615	0.682572
102400	208.36	145.69	72.6794	36.5162	29.1869	21.4627

Strong Scaling Analysis

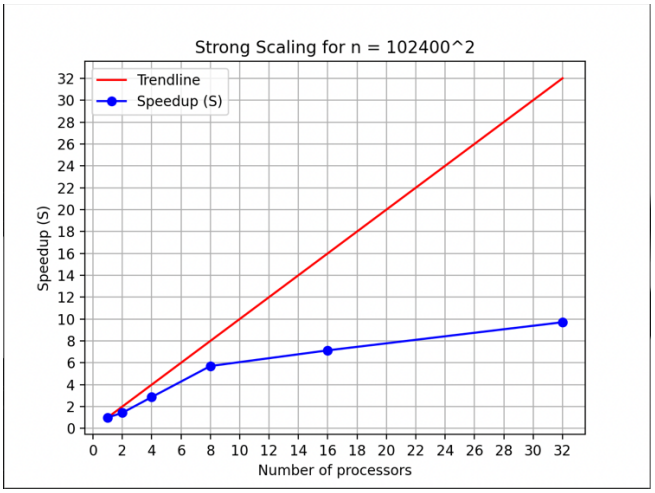
Keeping n constant to 102400 and 25600

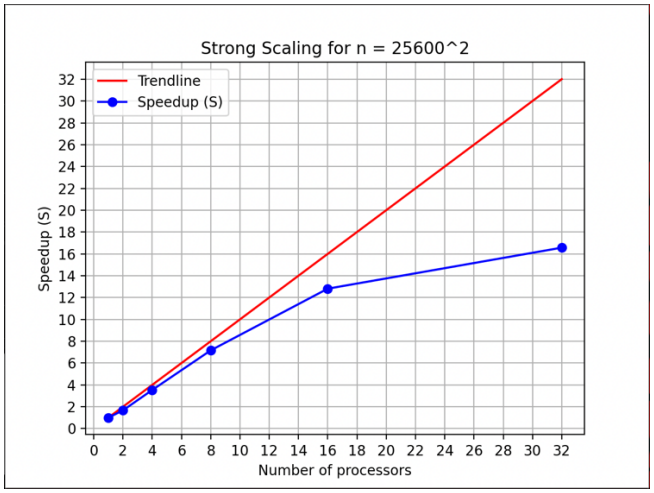
n = 102400

P	Speedup	Efficiency %
1	1.0	100.0
2	1.4301599286155500	71.50799643077770
4	2.866837095518130	71.67092738795310
8	5.705960642125960	71.32450802657450
16	7.138819127759370	44.61761954849610
32	9.708005050622710	30.33751578319600

n = 25600

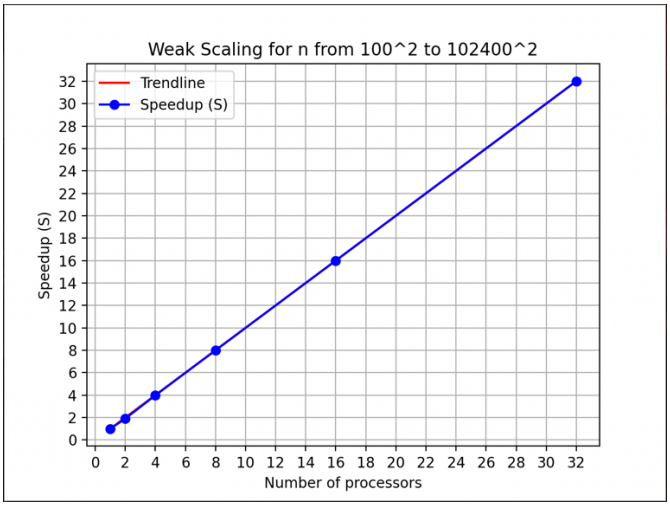
P	Speedup	Efficiency %
1	1.0	100.0
2	1.6692446202438200	83.46223101219120
4	3.533837640567050	88.34594101417620
8	7.16313359385391	89.53916992317390
16	12.813967584960600	80.08729740600370
32	16.569387551789400	51.7793360993419





Weak Scaling Analysis

P	Speedup	Efficiency %
1	1.0	100.0
2	1.9039114544182100	95.19557272091030
4	3.981129428826280	99.52823572065700
8	7.9955968943702000	99.94496117962760
16	15.999433018831800	99.9964563676985
32	31.99995181168300	99.99984941150920



Observation

This problem is not strong scalable. I think it is to do with the number of memory accesses required for the computation for each iteration. We can see that the efficiency is better when the amount of data is lesser. It is possible that the program is trying to fill up the cache to access data as soon as possible. But, when we increase the number of processors, the number of data being acted on increases, filling up the caches.

The problem is weak scalable. As the data is also increasing as the processors are increasing, cache limit is not affecting the computation to that extent.

References

1. https://en.wikipedia.org/wiki/Strassen_algorithm
2. https://www.cs.utexas.edu/~flame/pubs/2D3DFinal.pdf
3. https://docs.ccr.bu[alo.edu/en/latest/

Appendix

1.1. Implementation of calculating the constant values K_0, K_1 and K_2.

```
    /*
    Creates an array holding the sum of each row.
    */
    std::vector<float> K_rows_sum(3);
    for (long long int i = 0; i < 3; i++) {
        float sum = 0.0;
        for (long long int j = 0; j < 3; j++) {
            sum += K[i * 3 + j];
        }
        K_rows_sum[i] = sum;
    }
```

1.2. Implementation of column wise partial summation in parallel

```
/*
Input: dimensions of matrix A and matrix A
Output: array A with column partial sums

Adds A(i-1,j), A(i,j), A(i+1,j) and stores it in A(i,j)
Runs each column in parallel
*/
void col_partial_sum(long long int n, long long int m, std::vector<float>& A) {
    #pragma omp parallel for default(none) shared(A,n,m) schedule(static)
    for (long long int j = 0; j < m; j++) {
        float t = A[j];
        for (long long int i = 1; i < n-1; i++) {
            float current = A[i*m+j];
            A[i*m+j] = t + current + A[(i+1)*m+j];
            t = current;
        }
    }
}
```

1.3. Implementation of making a copy of the first and last columns of matrix ***A***.

```
    std::vector<float> A_first_col(n);
    std::vector<float> A_last_col(n);

    /*
    Creates a copy of the first and last column of matrix A in parallel
    */
    #pragma omp parallel for default(none) shared(A_first_col, A_last_col, A,n,m) schedule(static)
    for (long long int i = 0; i < n; i++) {
        A_first_col[i] = A[i * m];
        A_last_col[i] = A[i * m + m - 1];
    }
```

1.4. Implementation of row wise partial operation to get the final filter value

```
    /*
    Computer A(i, j) = A(i, j-1) * K1 + A(i, j) * K2 + A(i, j+1) * K3
    where KX is the sum of Xth row of K

    Runs each row in parallel
    */
    #pragma omp parallel for default(none) shared(A,K_rows_sum,n,m) schedule(static)
    for (long long int i = 1; i < n - 1; i++) {
        float t = A[i*m];
        for (long long int j = 1; j < m - 1; j++) {
            float current = A[i*m + j];
            A[i*m + j] = t*K_rows_sum[0] + current*K_rows_sum[1] + A[i*m + (j+1)]*K_rows_sum[2];
            t = current;
        }
    }
}
```