Image Modification using Cannon's Algorithm

Hashim Faisal K.V.Badhrinarayanan

1 Introduction

Image modifications done by performing non-parallelized multiplication of an image with a filter matrix can be computationally demanding and resource-intensive. This project parallelizes the operations done on images by using Cannon's algorithm for matrix multiplication. Cannon's algorithm is a widely used parallel algorithm for multiplying square matrices on high performance systems. It efficiently utilizes multiple processors to perform the multiplication simultaneously, significantly reducing computation time compared to traditional sequential methods. Its key features include scalability, parallelization and communication between cores. **Cannon's algorithm** has been implemented in **C** using **MPI**. The code is available in this Github repository.

2 Approach

The input image is read using python and libraries such as *OpenCV* and *numpy* are used to convert the image into an array of pixel values. If the image is a colour image, the pixel values are split into red, blue and green arrays representing each of the colour channels. A grayscale image would have only one array.

These pixel values are saved and loaded into a C file. A 2D filter matrix of the same size as the image is then generated. The pixel value matrix and the filer matrix are multiplied using Cannon's algorithm. This process is repeated for every channel in the case of a colour image. The output matrix is then written into a text file.

This output file is read in python and reshaped into the size of the original image. *OpenCV* is then used to reconstruct the image and display it.

3 Architecture

Cannon's algorithm:

- **Partitioning**: Divide each input matrix into square blocks, distributing them among the processors in a grid-like fashion.
- **Initial Alignment**: Initially, align the blocks in such a way that each processor has a local block that corresponds to a part of the resulting matrix.
- **Local Matrix Multiplication**: Perform local matrix multiplication on each processor using the blocks it holds.
- **Interprocessor Communication (Rotation)**: Rotate the blocks cyclically along rows and columns, ensuring that each processor gets the blocks it needs from its neighboring processors.
- **Repeat Rotation and Multiplication**: Repeat the rotation and multiplication steps for a number of iterations until each processor has contributed to computing all parts of the resulting matrix.
- **Collect Results**: Finally, gather the computed blocks from all processors to assemble the resulting matrix.

4 Advantages of HPC

Cannon's algorithm can effectively utilize High-Performance Computers because of its inherent parallelism. Some of its advantages are:

- **Scalability**: Cannon's algorithm scales efficiently with the number of processors, optimizing workload distribution for improved performance on high-performance computers.
- **Communication Efficiency**: It minimizes communication overhead by structuring computations and local data exchange, crucial for distributed memory systems.
- **Load Balancing**: The algorithm evenly distributes workload among processors, preventing idle time and ensuring efficient resource utilization.

- **Cache Optimization**: It maximizes cache efficiency by utilizing local data blocks, reducing cache misses and enhancing overall performance.
- **Modular Parallelism**: Cannon's structured approach allows for easy parallel implementation and optimization, exploiting parallelism for efficient computation across multiple processors.

5 Results

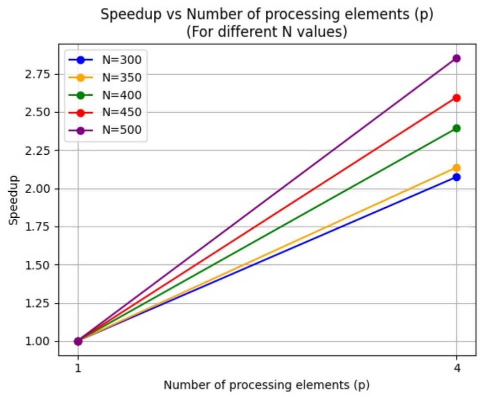
The code was executed using 1 and 4 processing elements, and the performance of parallel matrix multiplication using Cannon's algorithm was evaluated using 2 metrics: **Speed Up** and **Parallel Efficiency**.

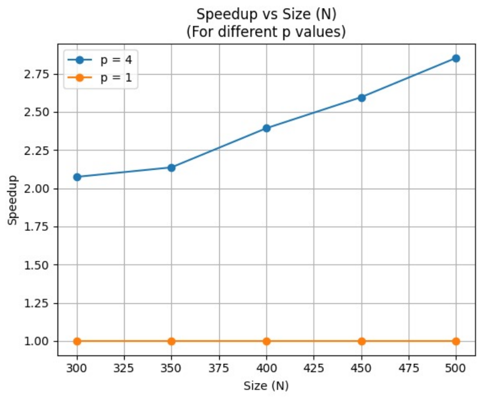
$$Speedup = \frac{\text{Running time of your program in 1 processing element}}{\text{Running time of your program in } p \text{ processing elements}} \tag{1}$$

$$ParallelEfficiency = \left(\frac{Speedup}{p}\right) \times 100\% \tag{2}$$

The performance was calculated for input images of various sizes: 300x300, 350x350, 400x400, 450x450, and 500x500.

The observed results are shown below:





N vs Speed Up:

Problem Size (N)	Processing Elements (p)	Speedup (S)
300	4	2.074210428453794
350	4	2.1363006737682086
400	4	2.3926439893947085
450	4	2.5959085773800066
500	4	2.8530850889437525

N vs Parallel efficiency:

Problem Size (N)	Processing Elements (p)	Parallel Efficiency (PE)
300	4	51.855260711344854
350	4	53.40751684420522
400	4	59.81609973486771
450	4	64.89771443450016
500	4	71.3271272235938

6 Conclusion

The project successfully parallelizes Cannon's algorithm in C using MPI. The scalability and performance for different problem sizes and number of processing elements is observed. The observations highlight the importance of using high performance computers for optimizing parallel algorithms. Cannon's algorithm being highly scalable, can be used for problems of even larger sizes, while maintaining good parallel efficiency and execution time.