University of Pavia

Department of Electrical Computer & Biomedical Engineering

A Project Report on

2-D Fast Fourier Transform

(using Message Passing Interface (MPI))

Submitted to:

Prof. Marco Ferretti
Dr. Luigi Santangelo

Submitted by:

Saurav Anand
Student ID- 519599

Summary

Aim of Project 3

The algorithm 4-6
- General description 4
- Formal description 5

A Prior Study 7-14
- Serial Algorithm 9
- Computational Complexity 10
- Speedup Analysis (AMDHAL'S LAW) 12

Testing and debugging 15-16

GCP Implementation 17-22
- Serial Execution 17
- Fat Cluster 18-20
  - Strong Scalability
  - Weak Scalability
  - Speedup & Efficiency
- Light Cluster 20-22
  - Strong Scalability
  - Weak Scalability
  - Speedup & Efficiency

Conclusion 23
Contribution 23

Fast Fourier Transform

Aim of Project:

Implementation of the Fast Fourier Transform (FFT) in 2D to learn bow the parallelization can help to speed up the program or opposite on different scenarios. Along with that to analyze the Fat cluster and light cluster to show Strong Scalability and weak Scalability on GCP (Google Cloud Platform) using MPI by taking the different sizes of the images as input to get the maximum speed up. In addition we will see the speedup and Efficiency for each of then respectively.

In this project, I have first given a general idea of FFT 1D and FFT 2D along with its mathematical expression and some of its application. While continuing further I have done a prior study of the whole project and did the mathematical calculation to calculate the theoretical speedup i.e. what is the maximum we can achieve to do the operation involved in doing the FFT 2D.

For doing the FFT2d using mpi there must be many approaches to distribute the data among the Master and slave machines but in this particular project I have chosen the MPI Scatter and MPI gather method.

After doing the prior study, I worked on the code and implemented it on the GCP and collected the result. GCP implementation is done for Fat cluster (Intra and Inter region) and Light cluster (Intra and Inter Region). Along with this result the Strong scalability, weak scalability and speedup in analyzed simultaneously.

Also, we have discussed some problem cause while doing the project along with its solution. And at last we have revised the conclusion for this project.

1. The_algorithm

1.1 General Description

Fast Fourier Transform, is an algorithm for efficiently computing the discrete Fourier transform (DFT) and its inverse. The Fourier transform is a mathematical transformation that decomposes a function or a signal into its constituent frequencies. The FFT is particularly useful in numerical analysis and signal processing because it significantly reduces the number of computations needed to perform a Fourier transform, making it more practical for real-time applications.

1.1.1 Fast Fourier Transform in 1D

The Fast Fourier Transform (FFT) in 1D is an algorithm for efficiently computing the Discrete Fourier Transform (DFT) and its inverse for one-dimensional signals, or we can say that the one-dimensional Fast Fourier Transform (1D-FFT) is a computational algorithm employed to analyze a signal in the time or spatial domain by revealing its corresponding frequency components in the frequency domain. It leverages the mathematical concept of the discrete Fourier transform (DFT), offering a representation of a finite sequence of data points in the complex plane. I used Cooley-Tukey Algorithm for this particular purpose.

If, we see that Mathematical Explanation for the Fast Fourier Transform (FFT) in 1D of a sequence of N complex Numbers then, xn is given by:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot W_N^{kn}, \quad \text{for } n = 0 \text{ to } N-1$$

Where,

X_k is k^{th} output frequency component
X_n is the n-th input sample
$W_N = e^{-j.2\pi/N}$ is the n-th root of unity
k, N are integer (0 to N-1)

Similarly, we can write Inverse- FFT (IFFT) as:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot W_N^{-kn}, \quad \text{for } k = 0 \text{ to } N-1$$

1.1.2 Fast Fourier Transform in 2D

The Fast Fourier Transform (FFT) in 2D is a computational technique designed for the efficient computation of the Discrete Fourier Transform (DFT) and its inverse in the realm of two-dimensional signals, such as images or matrices. This method extends the principles of the 1D FFT to simultaneously analyze signals along both horizontal and vertical dimensions. The 2D FFT is grounded in the mathematical concept of the 2D discrete Fourier transform, which decomposes a two-dimensional signal into its respective frequency components.

The mathematical expression for the 2D FFT builds upon the principles of the 1D FFT, involving the summation of complex exponentials for each pair of horizontal and vertical indices within the signal. This computational approach effectively decomposes the 2D DFT into smaller subproblems, akin to the 1D FFT algorithm, resulting in notable computational efficiency.

The 2D FFT finds extensive applications in image processing, offering an efficient means for the analysis and manipulation of images in the frequency domain. It facilitates operations such as filtering, compression, and feature extraction by exposing the frequency content inherent in an image. This algorithm plays a crucial role in diverse fields, including computer vision, medical imaging, and remote sensing.

1.2 Formal Description/ Mathematical Description

Mathematically we can define Fast Fourier Transform as:

Given a sequence of N complex numbers x(n), where n ranges from 0 to N-1. The 1D FFT denoted as X(k), is calculated using the formula:

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot e^{-j\frac{2\pi}{N}kn}$$

Here:

- $X(k)$ represents the complex amplitude at frequency k,
- N is the total number of samples in the sequence,
- $x(n)$ is the complex number at the nth position in the sequence,
- j is the imaginary unit.

This formula shows that how the 1D FFT transforms a sequence of complex numbers from the time domain into the frequency domain. The computation involves summing up complex exponentials for each frequency component, revealing the contribution of different frequencies in the original signal.

Similarly, we can also define FFT in 2D mathematically as:

$$X_{kl} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x_{mn} \cdot e^{-j\left(\frac{2\pi}{M} km + \frac{2\pi}{N} ln\right)}$$

Here:

- X_{kl} represents the complex amplitude at frequency (k, l).
- M is the total number of rows in the matrix.
- N is the total number of columns in the matrix.
- x_{mn} is the complex number at the position (m, n) in the matrix.
- j is the imaginary unit.

This formula represents the computation of the 2D FFT, revealing the frequency content of a two-dimensional signal. The nested summations iterate over both horizontal and vertical dimensions. The expression within the exponential term incorporates the frequencies along both dimensions.

2. A Priori Study

For this particular project, solving the problem of FFT 2D and for solving the problem with the help of parallelization for the MPI program.

Method- 1st

we used the strategy which will divide the input data into smaller chunks and distributing them to other different available processes.

Below is the sequential form of, how the operation will work and how the data will be distributed Assuming the image size is 512x512 and the number of cores available is 8.

Firstly, the master divides the input 2D matrix in a set of rows depending upon the number of the cores present. Here the matrix will be divided between the number of cores in the size of 512x512/8 which will be equal to 512x64 sized matrix.

Then, all the slave will perform the FFT 1D matrix on the received matrix and send the result back to the master.

After that the master will collect all the result from the slave and combine the matrix into a single matrix and transpose it.

Again, all the steps will be done again because this time the FFT will be done for the Column (because we did the transpose.) i.e. distributing the matrix to slave, slave doing the operations, slave sending the data to master and master Computing the result.

Method – 2nd

In this approach ww uses a block-cyclic distribution for parallelizing the 2D FFT. This method distributes sub-blocks of the matrix to different processes in a round-robin fashion, aiming to balance the load more evenly across processes and potentially improve performance on large, non-square matrices. However, it introduces additional complexity in data management and communication.

In this approach the input matrix is divided into smaller sub-blocks. After that it distribute these sub-blocks to different MPI processes in a block-cyclic manner. For example, if you have a 512×512 matrix and 4 processes, instead of dividing the matrix into rows or columns, divide it into smaller blocks (e.g., 64×64 blocks) and distribute these blocks in a cyclic pattern.

Then, each process performs a 1D FFT on the rows of its local sub-blocks and after completing the row-wise FFT, an all-to-all communication is performed to redistribute the data for column-wise FFT.

After that, it Transpose the local sub-blocks within each process and perform a 1D FFT on the columns of the transposed sub-blocks (which were originally rows).

In this case we can see that Scatter Gatther is an ideal solution to the FFD 2D because it has few advantages over the block cyclic distribution i.e

Scatter/Gather operations are straightforward, distributing contiguous blocks of data to processes and collecting them back, without the complexity of managing non-contiguous blocks or cyclic patterns while more complex due to non-contiguous and cyclic data distribution.

Scatter Gather Works well when data is naturally uniform and can be divided into equal-sized contiguous blocks while block Cyclic Distribution is Better for non-uniform data but may introduce overhead in uniform cases.

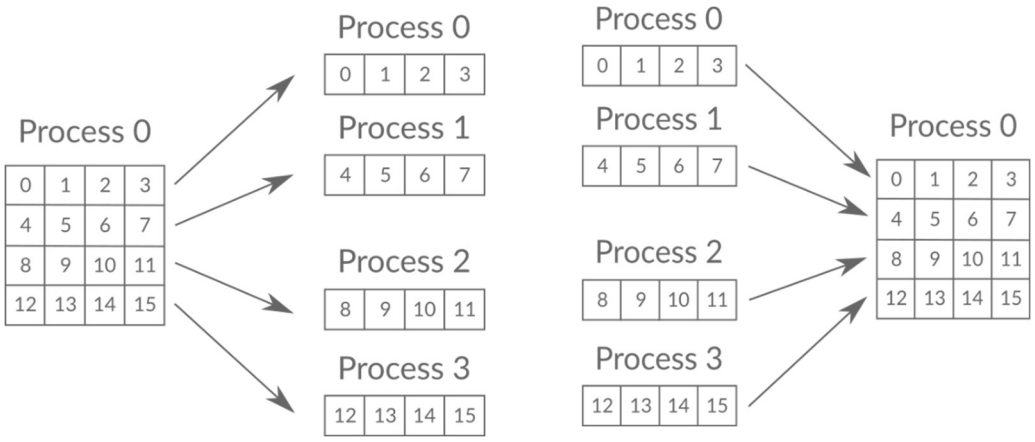
Scatter Gather has Lower communication overhead for distributing and collecting data while block Cyclic Distribution Involves more complex communication patterns, which can increase overhead.

Hence, for this particular FFT 2d Problem, we decided to move forward with the method one, where the size of the matrix will divide in the form of rows wise then transposing it, then column wise and then repeating the same while using the scatter Gather method because of the above-mentioned advantages over Block cyclic distribution to distribute the data from the master to salves and collecting it back to give the final result.

Sending and Receiving of Data (communication b/w Multiple Vm)

Here, sending and receiving of the data that is matrix in our case is done via the Scatter and the Gather.

MPI_scatter is used for sending the data from the master to the available number of slaves while MPI_gather is used to collect the data from all the slaves to the master.



2.1 Serial Algorithm- working Explanation

Initialization

Initialize the ‘imageIndex’ map with the following values

imageIndex = {512: 0, 1024: 1, 2048: 2, 4096: 3}

Initialize the ‘images’ 2D array with the file paths

images = [

["source_gray_image/512_gray.txt", "source_gray_image/1024_gray.txt", "source_gray_image/2048_gray.txt", "source_gray_image/4096_gray.txt"],

["fft_txt/512_fft.txt", "fft_txt/1024_fft.txt", "fft_txt/2048_fft.txt", "fft_txt/4096_fft.txt"]

]

Execution

Main Function Execution:

Get the ‘Validsize’ from command line argument
Initialize rows and columns with validsize.
Record the start time t1.

Reading the Image:

Open the file corresponding to the validsize from images array.
Read the file line by line and store the values in a 2D complex buffer array.
Record the time t2.

Perform FFT:

Row-wise FFT:
For each row in the buffer, perform 1D FFT.
Record the time t3.
Transpose the Matrix:
Transpose the buffer matrix.
Record the time t4.
Column-wise FFT:
For each column in the buffer, perform 1D FFT.
Record the time t5.

Perform Inverse FFT:

Row-wise Inverse FFT:

For each row in the buffer, perform 1D Inverse FFT.
Record the time t6.

Transpose the Matrix Again:

Transpose the buffer matrix.
Record the time t7.

Column-wise Inverse FFT:

For each column in the buffer, perform 1D Inverse FFT.
Record the time t8.

(Optional) Write the Result to a File:

Open the file corresponding to the validsize from images array.
Write the buffer matrix to the file.
Record the time t9.

Termination

Print Execution Times:

Print the time taken for reading the image, performing FFT, performing inverse FFT, and total execution time.

Note: After every operation we have printed the time taken by that particular operation to complete it. By this way it will become easier for us to analyze the time and helped us to know which process is taking what much amount of time.

2.2 Computational Complexity Analysis

For calculating the Computational Complexity, here we have first analyzed the code to find out that how many steps the total execution of the FFT 2D are going to take place. Also, we have to make sure which of those operations are working in serial manner and which of those are working in the parallel manner. After that we need to find out the number of Instructions.

For this particular code the total 7 no. of Steps are involved:

Reading the File:

Outer while loop (getline): N operations
Variable declarations and initializations inside the outer loop: 3N operations
Inner while loop condition check and execution: N^2 operations
Assigning values to the buffer and incrementing second_dmns_count: $2N^2$ operations
Incrementing first_Dmns_count: N operations

Total Number of Instructions: $N + 3N + N^2 + 2N^2 + N$

$= 5N + 3N^2$

FFT for rows (First Time):

Base case check: 1 instruction.

Dividing the input vector: N instructions

Combining the result: 3N / 2

Total excluding recursive calls $= 1 + 5N/2$

The FFT algorithm divides the problem into two halves recursively. The recurrence relation for the total number of instructions T(N) is:

$$T(N) = 2T(N/2) + (1 + 5N/2)$$

Solving this recurrence relation:

At each level of recursion, the non-recursive part (combining results, etc.) takes = $(1 + 5N/2)$ instructions.

The number of levels of recursion is $= \log_2 (N)$

Total instructions (for 1 time of fft 1d) $= N + 5N/2 \log_2 (N)$

Total instructions for N rows= $N^2 + (5N^2 \log_2(N)) / 2$

Transposing the matrix (first time):

Outer loop: N times

Inner loop: N(N-1) / 2

Swap operation in each inner loop iteration: 3 assignment per swap

Total instructions: $= 3N(N-1) / 2 = (3N^2 - 3N) / 2$

FFT for column (second Time):

Total instructions for N rows= $N^2 + (5N^2 \log_2(N)) / 2$

IFFT for columns (First Time):

First conjugate loop: 2N

FFT Call: $N + 5N/2 \log_2 N$

Second conjugate loop: 2N

Normalizing loop: 2N

Total instruction=: $6N + 5N/2 \log_2(N)$ for one column
Total instructions for N columns= $6N^2 + (5N^2 \log_2(N)) / 2$

Transposing the matrix (second time):

Total instructions= $(3N^2 - 3N) / 2$

IFFT for rows (Second Time):

Total instructions for N columns= $6N^2 + (5N^2 \log_2(N)) / 2$

2.3 Analysis of the Speedup

2.3.1 Speedup Analysis using AMDHAL'S LAW

Amdahl's Law is used to find the maximum improvement possible for a system's performance when only part of the system is improved. It is commonly used in parallel computing to predict the theoretical maximum speedup of a task using multiple processors.

Here, we need to keep in mind that we have to put the Total workload, Proportion of the Parallelizable work constant while the number of the processors keeps increasing.

Amdahl's Law is typically expressed with the following formula:

$$Speedup = \frac{1}{(1-p) + p/N}$$

$$parallel\ fraction\ = \frac{number\ of\ instructions\ of\ parallelizable\ blocks}{total\ number\ of\ instructions}$$

Where:
S is the theoretical speedup
P is the proportion of the program that can be parallelized (Parallel Fraction)
N is the number of processors

Now, here to compute the parallized part we need to know that which part of the Serial code is going to be parallel which is not. Below is the table showing the same after calculating the instruction, here is the final chart which looks something like this:

Hence the final chart will look something like this.

Operations	Execution Manner	Total No. of Instructions
Reading the File	Serial	$5N + 3N^2$
FFT for rows (First Time)	Parallel	$N^2 + (5N^2 \log_2(N)) / 2$
Transposing the matrix (1st time)	Serial	$3N(N-1) / 2 = (3N^2 - 3N) / 2$
FFT for columns (2nd Time)	Parallel	$N^2 + (5N^2 \log_2(N)) / 2$
IFFT for columns (1st Time)	Parallel	$6N^2 + (5N^2 \log_2(N)) / 2$
Transposing the matrix (2nd time)	Serial	$(3N^2 - 3N) / 2$
IFFT for rows (2nd Time)	Parallel	$6N^2 + (5N^2 \log_2(N)) / 2$

Number of Parallel Instruction (P)	$14N^2 + 10N^2 \log_2(N)$
Total Number of Instruction (N)	$20N^2 + 10N^2 \log_2(N) + 2N$

$$\text{Parallel fraction } (P/N) = \frac{14N^2 + 10N^2 \log_2(N)}{20N^2 + 10N^2 \log_2(N) + 2N}$$

Now by substituting the value of N as 512,1024,2048 & 4096 in the above equation respecetively,

We get, parallel fraction for size
512x512= 0.945
1024x1024= 0.95
2048x2048= 0.953
4096x4096= 0.957

Now Speed up will be ,
Speed Up =

512x512 image = $1 / \{(1-0.945) + (0.945 / 16)\} = 8.77$

Similarly, calculating speedup for every image with different no. of cores we get

Image Size	Core=8	Core=16	Core= 24	Core=32
512x512	5.78	8.77	10.63	11.90
1048x1048	5.95	9.17	11.23	12.65
2048x2048	6.02	9.43	11.62	13.15
4096x4096	6.17	9.80	12.04	13.88

Hence, with all the above readings we can conclude that, theoretically while using 8 to 32 number of cores we have achieved a good amount of speed ranging between 5 as minimum to 13 as maximum.

Now the next step is to implement the written code on the GCP with multiple scenarios with multiple VMs and finally analyze the speedup achieved practically and then we will compare what speedup we get comparing both.

2.3.2 Why Serial not parallel

In the above analysis you can notice that I have done the Reading of the image part in the serial way not in the parallel way because reading from disk or cloud storage happens one piece at a time. Even if we try to read multiple parts simultaneously, the hardware and how files are stored typically limit how much speed we can gain. Instead, we focused on making the calculations themselves faster by using parallel processing for the FFT (Fast Fourier Transform) operations. This approach was more effective for our project because it maximized the use of our computing resources and made our calculations quicker, which was our main goal.

However, parallel file reading can be implemented using techniques such as multi-threading or asynchronous I/O. Multi-threading involves creating multiple threads within a program to read different parts of a file simultaneously, utilizing multiple CPU cores. Asynchronous I/O allows the program to initiate multiple read requests and continue executing other tasks while waiting for data to arrive, enhancing overall efficiency. These methods can potentially speed up file reading operations in applications where disk I/O is a bottleneck.

3. Testing and Debugging

After writing of the code and running it for taking the results there were many issues arises from a minor to a major issue which needs attention to continue the work for further analysis, some of the important ones are mentioned below which its method to correct it.

3.1 Firstly I checked the program with the image size of 128x128 and the execution take place s with a very less fraction which may be cause a problem for me to analyze the result and to plot the graph, hence started to choose the image from 512x512 onwards.

3.2 Segmentation Fault

In this particular problem, when e uploaded the code on the master and created some few machines to run the mpi code to take the result this problem caused:

When I run my MPI code on a single VM for image size 512x512, 1024,2048,4096 it works properly for all image sizes. but when I run the same code by switching on more than one VM (for example- when I use 2 VM to get 4 cores from VM one and 4 core from VM number two) then it works fine for the image size 512x512 but it does not work for the 1024,2048, 4096 size. It shows some segmentation fault.

```
saurav_anand01@m1:~/FFT_2024$ mpirun --prefix /usr/local/openmpi-4.1.1/ -np 8 --hostfile hostfile.fatCluster mpi_scatter_gather_full 1024
--------------------------------------------------------------------------
Primary job  terminated normally, but 1 process returned
a non-zero exit code. Per user-direction, the job has been aborted.
--------------------------------------------------------------------------
--------------------------------------------------------------------------
WARNING: Open MPI failed to TCP connect to a peer MPI process.  This
should not happen.

Your Open MPI job may now hang or fail.

  Local host: m1
  PID:        1073
  Message:    connect() to 10.128.0.4:1026 failed
  Error:      Operation now in progress (115)
--------------------------------------------------------------------------
[m1.c.my-project-44344-aca.internal:01068] PMIX ERROR: UNREACHABLE in file server/pmix_server.c at line 2198
Time to read the image
0.0688971
 ************************************FFT Starts here ************************************
--------------------------------------------------------------------------
mpirun noticed that process rank 4 with PID 1035 on node 10.128.0.4 exited on signal 11 (Segmentation fault).
--------------------------------------------------------------------------
saurav_anand01@m1:~/FFT_2024$
```

Approach- This was happening because of the stack size limitation, hence changed the size

hard stack size (size in kb)

soft stack size (size in kb)

3.3 Manual Connection of Slaves with Master

When we have created a master machine and some other few machines we have to establish the connection of all the slaves with the master one by one with the help of SSH where we first put the IP address of the machine in the SSH folder (e.g.- SSH hostfile.fatcluster) and then establish the connection with the master. One the connection got established then no issue arises again.

3.4 Connection (Public Key- authorized kay)

One more reason which leads to the connection failure. This problem gets shorted when the public key was copied in authorized key so that the connection get established easily.

3.5 Writing the Image

When the code was completed and when I run it (help of profilers) for checking its working properly and giving the result to the provided path/folder, I analyzed that the writing of the image is taking too much time for execution. Hence once the code was working, I removed the write part to make our calculation easier and execution faster.

4. GCP Implementation (Experimental Results)

** Legends
Intra- at same physical location
Inter- at different physical location

Once everything was done, we did the setup for taking all the required results to learn about the Strong scalability, weak scalability, speedup and efficiency of the Fat and Light cluster within the Intra and the Inter Region.

Here, we also need to run the serial also so that when we take the results of our mpi code (new data) we can compare it with our serial code data (old data) to analyze and do our findings on the based data.

4.1 Serial Code Execution time

Size	512x512	1024x1024	2048x2048	4096x4096
Time	0.129659	0.646147	2.79739	12.4649

4.2 Fat Cluster, Light Cluster, Scalability, speedup and Efficiency Analysis

Fat cluster have high performance nodes that is, each node has large number of CPUs/cores. On the other hand, Light cluster have low powerful nodes that is every present node have a smaller number of CPUs/cores present.

Meanwhile, here there is two more things which needs to be in attention i.e. strong scalability and weak scalability which will help us to analyses our results which we have taken while running our machines on different scenarios.

In strong scalability, we put our Image/data size constant and keep increasing the number of CPUs/cores and then analyze the performance while in weak scalability our aim was to keep increasing the image size along with the numbers of cores, but we have to keep in mind that each nodes gets equal no of computation data. So here in weak scalability the data should come approx. equal to each other and give us a straight line.

Now, for analyzing the speedup here we need to compare the previous Execution time (old data- data captured with serial code for all four images) with the new execution time.

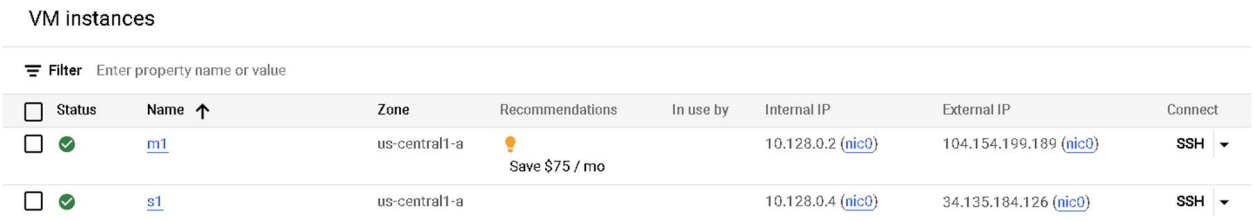
Speed up – Execution time old / Execution time new

Again, now once the speedup is calculated, we are able to get the efficiency performance i.e. what is the behavior of the efficiency as the number of cores increased.

Efficiency = Speedup Achieved / No. of Cores

4.2.1 Fat Cluster (Intra Region)

Here, we have first created two machines having 16 CPUs/Cores in each and were present at same location.



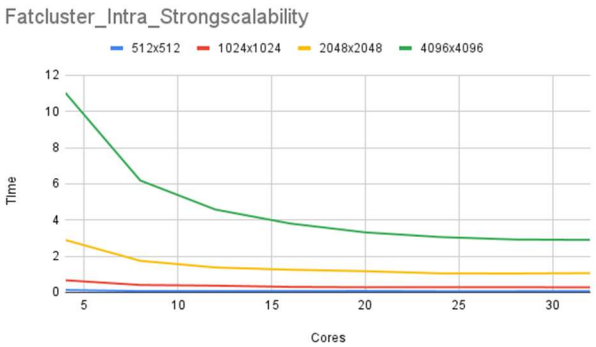
Below is the chart showing the results for different image sizes for different no of cores:

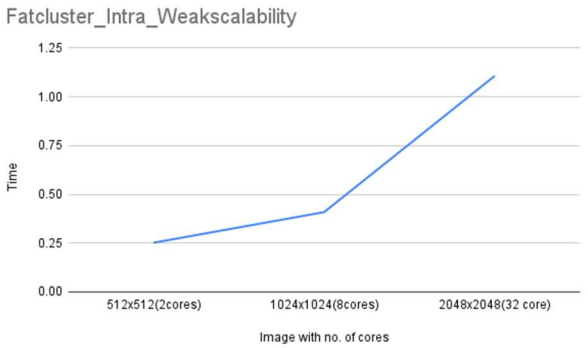
Cores	512	1024	2048	4096
2	0.252083	1.19737	5.20121	21.02
4	0.131188	0.670727	2.89722	11.0368
8	0.0738335	0.408832	1.74481	6.18404
12	0.0756369	0.372013	1.37628	4.57935
16	0.0768338	0.303193	1.25295	3.80414
20	0.0790422	0.283329	1.17085	3.31517
24	0.0634282	0.287573	1.05137	3.05664
28	0.0643286	0.285998	1.0447	2.92073
32	0.0678683	0.279615	1.06628	2.89897

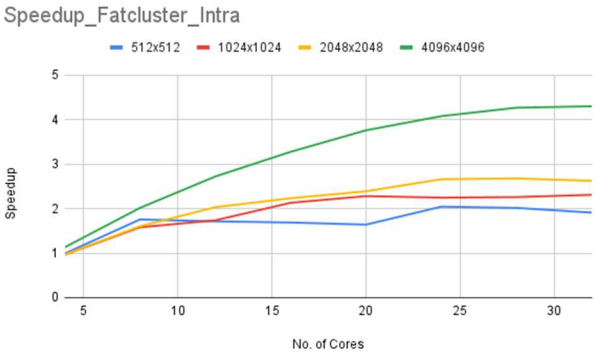
Strong Scalability & Weak Scalability:

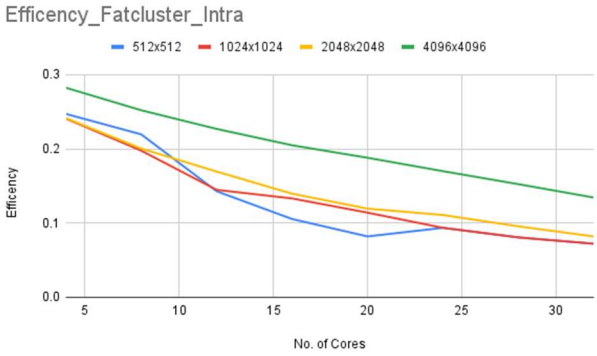
In the result above, for the strong scalability we have kept the image size constant and keep increasing the number of cores and as a result we can see that the expectation time is kept decreasing as the number if cores increased. On the other hand for the weak scalability, we kept increasing the number of size also increased no o cores gradually but keep the equal data distribution on each core and we a graph as below.

If we talk about the weak scalability, where the size of the data should be increased with increasing number if the cores, we choose 512x512 to run with 2 cores, 1024x1024 with 8 cores and 2048x2048 with 32 cores. In this way the work load distribution will be equal to all. Now according to the theory all the data for weak scalability comes near to each other to give a straight-line graph. But here we can see that we have not achieved that data nor the graph.



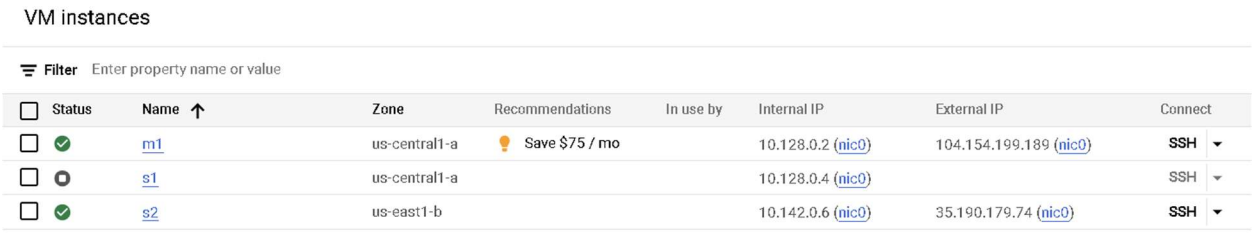






4.2.2 Fat Cluster (Inter Region)

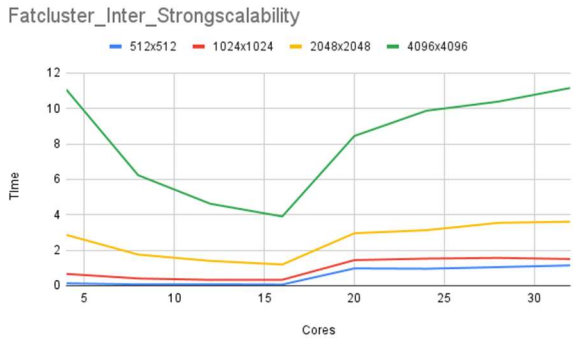
Here, again we crested two machines, where one of them was present one location and other was present at a different location.

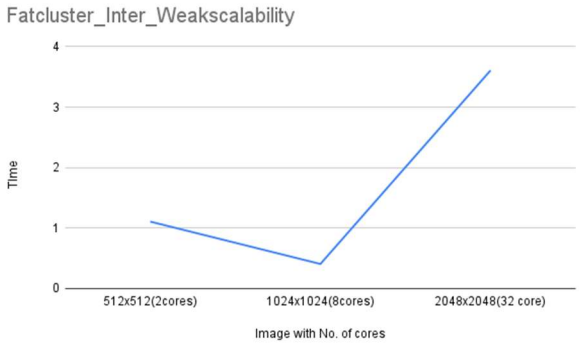


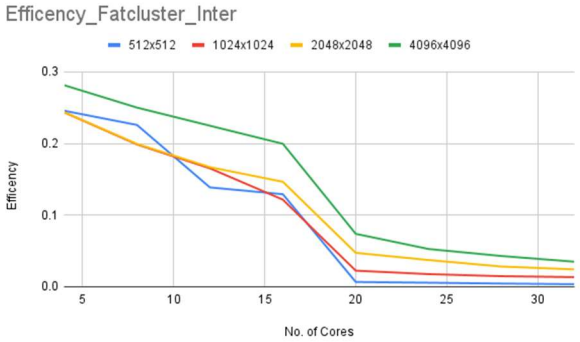
Cores	512	1024	2048	4096
2	0.252476	1.19459	5.1897	23.9158
4	0.131959	0.663352	2.86913	11.0774
8	0.0717803	0.406644	1.75437	6.23207
12	0.0779518	0.325854	1.39683	4.62216
16	0.0627525	0.33216	1.19352	3.90451
20	0.973773	1.44009	2.95706	8.44357
24	0.954078	1.52684	3.1297	9.86763
28	1.0464	1.56667	3.54487	10.3845
32	1.14387	1.50307	3.60809	11.1554

Strong Scalability & Weak Scalability:

In the above readings we can see that as the number of cores were increasing the time kept decreasing. But here as we know our one machine is at other location and having 16 cores in it. Hence once we increase the number of cores more than 16, the time increased at sudden and then kept decreasing again, this is because the location of machine and it took time to communicate between them.







4.2.3 Ligh Cluster (Intra_Region)

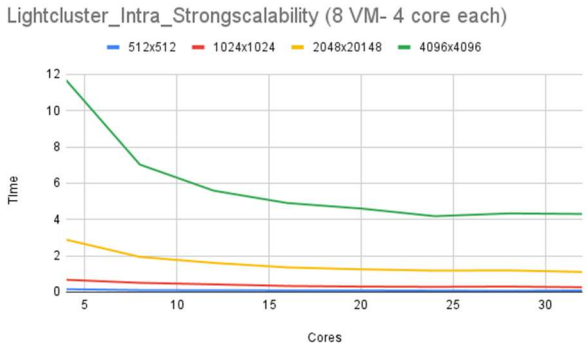
For, this particular setup, we have created 8 machines and each was having 4 CPUs/Cores in it and were present at the same location.

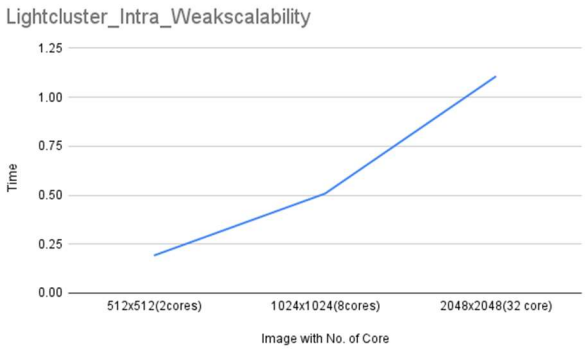
Status	Name	Zone	Recommendations	In use by	Internal IP	External IP	Connect
✓	m1	us-central1-a			10.128.0.2 (nic0)	34.68.49.97 (nic0)	SSH
✓	s1	us-central1-a			10.128.0.4 (nic0)	35.226.141.32 (nic0)	SSH
✓	s2	us-central1-a			10.128.0.7 (nic0)	35.225.104.79 (nic0)	SSH
✓	s3	us-central1-a			10.128.0.8 (nic0)	34.42.155.148 (nic0)	SSH
✓	s4	us-central1-a			10.128.0.9 (nic0)	34.30.129.190 (nic0)	SSH
✓	s5	us-central1-a			10.128.0.11 (nic0)	34.133.123.117 (nic0)	SSH
✓	s6	us-central1-a			10.128.0.10 (nic0)	35.224.219.241 (nic0)	SSH
✓	s7	us-central1-a			10.128.0.12 (nic0)	35.225.230.117 (nic0)	SSH

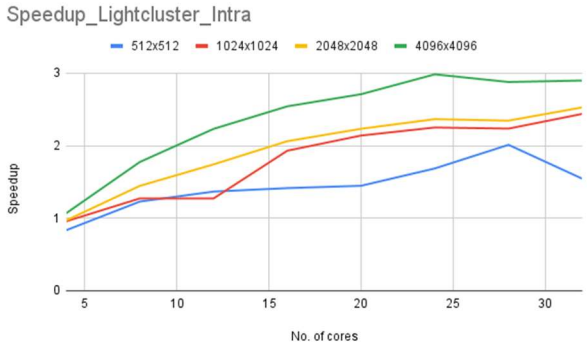
Cores	512x512	1024x1024	2048x2048	4096x4096
2	0.191702	0.842879	3.65998	15.9288
4	0.155561	0.675567	2.88686	11.6796
8	0.105493	0.507851	1.93567	7.02512
12	0.0948809	0.507851	1.60627	5.58793
16	0.091671	0.334827	1.35722	4.90486
20	0.0896347	0.302091	1.25331	4.602
24	0.0769328	0.287185	1.18292	4.17963
28	0.0644792	0.302333	1.19331	4.33311
32	0.0839565	0.26511	1.10714	4.30274

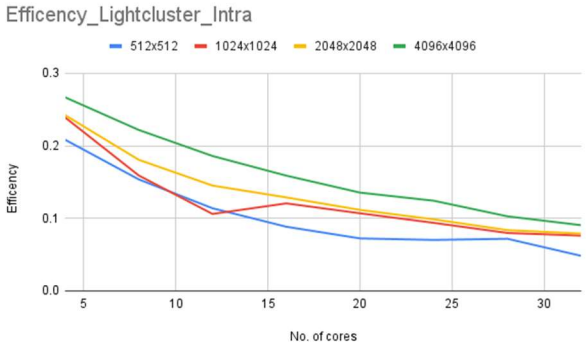
Strong Scalability & Weak Scalability:

In the above result also, we kept all the machine at same location and the image size constant and kept increasing the number of cores and we can analyze that, the operation time keep decreasing as the number of cores increased.









4.2.4 Light Cluster (Inter_Region)

For, this particular setup also, we have created 8 machines and each was having 4 is CPUs/Cores, But the machine distribution was done at 3 location that is the first two machine were at one location, the other three were at second location and the remaining three were at the third location.

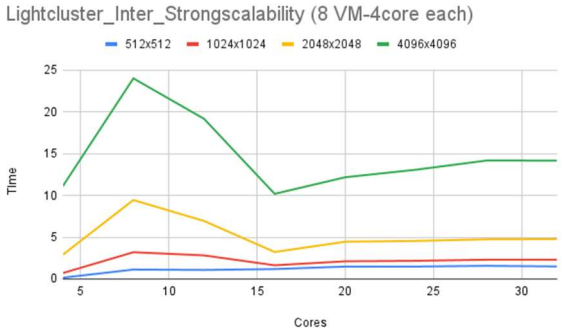
Enter property name or value

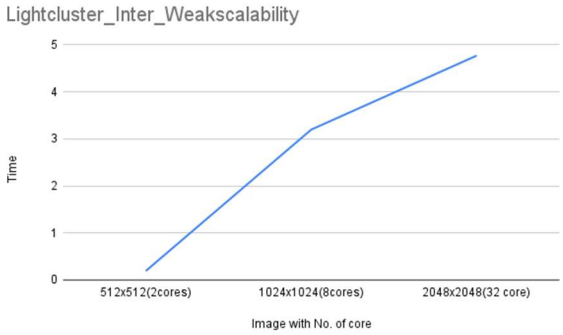
Name	Zone	Recommendations	In use by	Internal IP	External IP	Connect
m1	us-central1-a			10.128.0.2 (nic0)	34.68.49.97 (nic0)	SSH
s1	us-central1-a			10.128.0.4 (nic0)		SSH
s10	us-east1-d			10.142.0.5 (nic0)		SSH
s11	us-east4-a			10.150.0.2 (nic0)		SSH
s12	us-east4-b			10.150.0.3 (nic0)		SSH
s13	us-east4-c			10.150.0.4 (nic0)		SSH
s8	us-east1-b			10.142.0.3 (nic0)		SSH
s9	us-east1-c			10.142.0.4 (nic0)		SSH

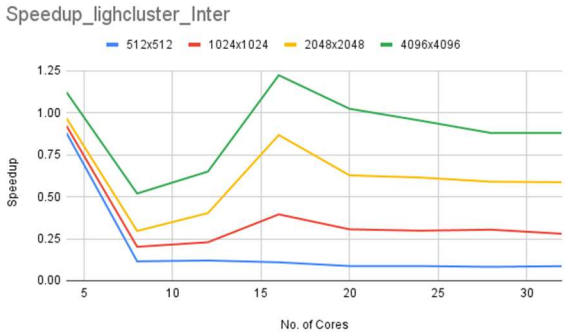
cores	512x512	1024x1024	2048x2048	4096x4096
2	0.191702	0.842879	3.65998	15.9288
4	0.147164	0.699756	2.88797	11.0932
8	1.12351	3.19694	9.44518	24.0117
12	1.07483	2.81941	6.94217	19.1713
16	1.18199	1.63465	3.2244	10.1873
20	1.47894	2.11121	4.4585	12.1756
24	1.48226	2.16886	4.55277	13.0752
28	1.57006	2.30159	4.74409	14.1746
32	1.49464	2.30999	4.76969	14.1676

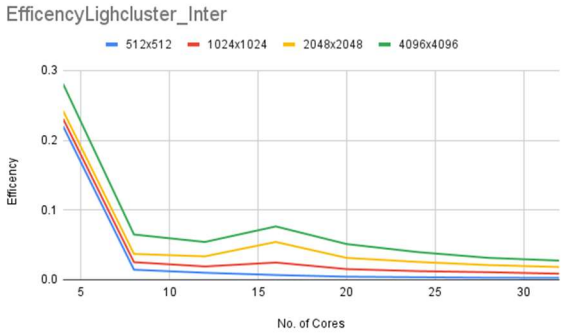
Strong Scalability & Weak Scalability:

In this particular as we have multiple location and hence the communication time between these locations can take time, hence we can see that clearly that the time consumed from initial with 4 core is lesser than the time taken to do the operation with the 32 cores.









5. Conclusion:

AS we can see from all the results above is that, the speedup we get theoretically is far greater than the speed we get while doing the practical.
This can depend on many factors such as the communication time between the machines or location where the machines are established.

If, we talk regarding the speedup, in most of the cases the speed increased as the number of cores also get increased. But at few times the speed gets decreased as the connection between the machine's changes and at few times the speed gets mostly same around it.

One Can also observe that the Fat cluster works more faster than the Light Cluster because the MPI Communication time is much lesser here.

We can also see that, in all the cases of weak scalability we have not achieved the data near to each other nor the straight-line graph which could happen due to various factor like workload distribution (but this is not valid in our case we did equal work distribution), communication overhead, synchronization, or hardware limitations. By profiling our application, optimizing communication, and potentially improving the algorithm, we can improve the weak scalability of your FFT2D implementation.

There have been some few changes which can be done to achieve more speedup. In this particular project the transposing of the matrix is done serial wise. Hence if this portion got parallelized it will be possible to achieve some more speedup.

6. Contribution

For this project, which was initially designed to be undertaken by two individuals, I chose to work independently and completed all aspects single-handedly. From the initial screening and reading about the project details to the implementation on Google Cloud Platform (GCP) and writing the comprehensive report, I managed every step on my own. Throughout the process, I utilized various online resources to deepen my understanding of specific concepts and to effectively implement virtual machines on GCP. This solo effort not only enhanced my technical skills but also significantly improved my ability to manage and execute a complex project independently.