Fast Fourier Transform with MPI

Ismail Kerem Tatlici (517418), Shubham Shubhankar Sharma (517420)

February 18, 2023

1 Analysis of the serial algorithm

1.1 Fast Fourier Transform in 2D

The one-dimensional Fast Fourier Transform (1D-FFT) is a computational algorithm used to decompose a signal in the time or spatial domain into its corresponding frequency components in the frequency domain. The 1D-FFT is based on the mathematical concept of the discrete Fourier transform (DFT), which is a way of representing a finite sequence of data points in the complex plane.

The mathematical formula for the one-dimensional Fast Fourier Transform (FFT) of a sequence of N complex numbers, x_n, 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 the k-th output frequency component, x_n is the n-th input sample, $W_N = e^{-j \cdot 2\pi/N}$ is the N-th root of unity, and k and n are integer values that range from 0 to $N-1$.

The inverse FFT (IFFT) is given by:

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

1.2 Two-dimensional Fast Fourier Transform

The two-dimensional Fast Fourier Transform (2D-FFT) is a computational method used to decompose an image or signal in the spatial domain into its corresponding frequency components in the frequency domain. It is an extension of the one-dimensional Fast Fourier Transform (1D-FFT) and is used to analyze images and other two-dimensional signals.

The 2D-FFT algorithm works by first decomposing the image or signal into rows, and then for each row, it applies the 1D-FFT algorithm. The resulting frequency coefficients are then arranged in a two-dimensional matrix. Next, the algorithm transposes this matrix and decomposes each column using the 1D-FFT. The final result is a two-dimensional matrix of frequency coefficients that represents the original image or signal in the frequency domain.

2 A-priori study of available parallelism.

2.1 Strategies for parallelisation for the MPI program of FFT 2D.

2.1.1 First Approach

This approach involves dividing the data into smaller chunks and distributing them among the different processes:

1. First, the master divides the 2D input matrix $complex < double > [][]$ in a set of rows depending on the number of cores. For example: If we have a 512×512 matrix and 4 processes (1 master and 3 slaves), then each of the processes will get $(512 \times 512/4) = 128 \times 512$ sized matrix.

2. After that, the slave performs FFT 1D on the rows of the sub-matrix individually. This result is sent back to the master.

3. The sending and receiving of the sub-matrix is done through MPI_Send and MPI_Recv. As these are blocking functions, the processes will only start doing computation once the data has been received completely.

4. The master first combines all these results into a single matrix and then takes a transpose of the same. All the steps will run for the second iteration. This time the FFT is done on the columns (because of the transpose).

2.1.2 Second Approach

The 1D FFT can be parallelized to some extent using MPI, but it is limited by the nature of the FFT algorithm itself. The FFT algorithm relies on a divide-and-conquer approach, where the input signal is split into even and odd indices, and then each of these sub-signals is transformed separately. This process continues recursively until the size of the sub-signals becomes small enough to be transformed directly.

In the 1D FFT, this divide-and-conquer approach results in a sequential process, where each step of the transformation depends on the result of the previous step. This makes it difficult to parallelize the 1D FFT using MPI, as each process would need to wait for the results from other processes before it can continue.

2.2 A-priori theoretical assessment

We are going to use the first approach for our MPI implementation. Here, the total execution time is computed using the following formula:

$$et = sct + pt$$

where et = Total Execution Time,
sct = Synchronization and Communication time between master and slave processes,
pt = Actual processing of data by slave and master.

2.2.1 Strong Parallelism (Fat Cluster):

According to this approach, as we increase the number of cores for a fixed-sized image, our execution time should decrease linearly until a certain point and then become constant. For the fat cluster, we are going to use 2 machines in the same region with 16 cores each. So, *sct* will be small as the distance between the master and slave processors are short. These are some of our expectations:

1. The distance between master and slave processor is less if they are on the same virtual machine, and more if they are on different virtual machines.

2. As we increase the size of the image (e.g. 8192x8192), the speed up in comparison to a smaller sized image (e.g., 512x512) is much better with the increase the number of cores. This behavior could be explained by the fact that when the images are smaller, *sct* dominates the *pt*. But as we increase the size of the image, *pt* starts dominating *sct*.

2.2.2 Weak Parallelism (Intra Regional Light Cluster):

According to this approach, as we increase the number of cores for a fixed-sized image, our execution time should decrease linearly until a certain point and then become constant. But in comparison to the fat cluster, the time taken would be more as the distances between the master processor and the slave processors are bigger.

For the intra-regional light cluster, we are going to use 16 machines in the same region with 2 cores each. So, *sct* will be larger as the master and the slave processor are much further away. These are some of our expectations:

1. In this case, there are 15 different machines. As we increase the size of the image (e.g. 8192x8192), the total execution time series for fat clusters and weak clusters shows more similar behavior. This is because the *pt* starts dominating *sct* as we increase the size of images.

2. On the other hand, with smaller images (e.g. 2048x2048), *sct* is larger in light clusters than the fat clusters, the *pt* is always smaller in the fat cluster than the light cluster.

2.2.3 Weak Parallelism (Inter Regional Light Cluster):

According to this approach, as we increase the number of cores for a fixed-sized image, our execution time should increase linearly. For the inter-regional light cluster, we are going to use 16 machines in the same region with 2 cores each. Out of the 16 machines, 2 will be in the same region (one master and slave), 14 will be divided in three regions (different from the region of the master). These are some of our expectations:

1. The linear increase could be attributed to the fact that *sct* becomes so huge, that increasing the number of cores which decreases the *pt* by dividing the task into smaller parts doesn't help much.

2.3 Profiling

We used `valgrind` as tools for profiling the serial and the parallel code. The compilation of the binary was done with `-O2` optimization and with the `-g` flag.

```
g++ -g -O2 -o fft_serial.out fft_serial.cpp
```

Command used for `valgrind`:

```
valgrind --tool=callgrind --main-stacksize=999999999 ./fft_serial 512
```

Matrix Size	Number of Instructions	Time elapsed(sec)	CPI (2.2 Ghz)
512x512	3,815,937,072	0.833983	0.4808157381
1024x1024	15,722,423,038	3.5549	0.4974284168
2048x2048	64,874,945,375	15.1177	0.5126623199
4096x4096	259,499,781,500	65.7212	0.557174419
8192x8192	1,037,999,126,000	288.415	0.6112847151

Table 1: Serial Code

Matrix Size	Number of Instructions	Time elapsed(sec)	CPI (2.2 Ghz)
512x512	3,815,937,072	0.439615	0.2534509825
1024x1024	15,722,423,038	1.75092	0.2450019307
2048x2048	64,874,945,375	7.22725	0.2450861409
4096x4096	259,499,781,500	30.5155	0.2587058055
8192x8192	1,037,999,126,000	116.396	0.2466969322

Table 2: MPI Code with 8 Cores

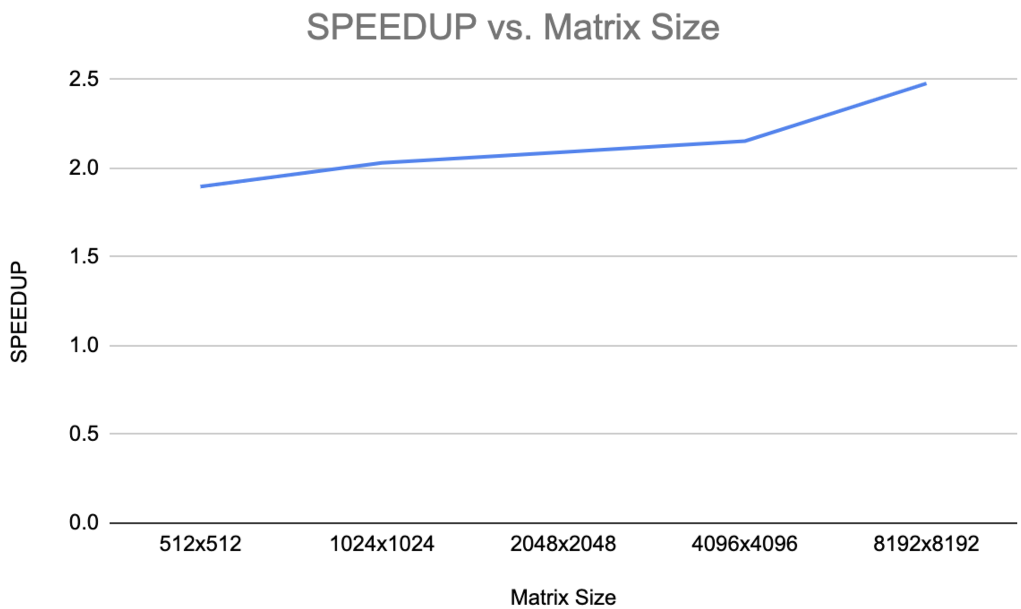

Figure 1: A-priori theoretical assessment using Amdahl's Law

3 MPI parallel implementation.

In this implementation, we have used MPI_Send() and MPI_Recv() to send the data of *complex* $<$ *double* $>$ type from master to slave and vice-versa. The master divides the data and then sends it to the slave. The below figures illustrates this process in detail.

```
// Send the values for the 1st time.
for (int i = 1; i < size; ++i)
{
  int retVal = MPI_Send(buf[i*chunk], MAX * chunk, MPI_C_DOUBLE_COMPLEX, i, 555, MPI_COMM_WORLD);
}
```

Figure 2: Sending the data from master to slave

```
complex<double> buf1[chunk][MAX];
//I'm the slave
// Receiving the value for the 1st time.
int retVal = MPI_Recv(buf1, MAX * chunk, MPI_C_DOUBLE_COMPLEX, 0, 555, MPI_COMM_WORLD, &status);

// FFT Part starts here.
// 1 D fft row wise.
for (int row = 0; row < chunk; row++)
{
  fft1d(buf1[row], MAX);
}
```

Figure 3: Receiving the data from master to slave and implementing 1D-FFT on rows

1. We adopted the first approach explained in section 2.1.1.

2. The images read by the python program called "convert_image.py" from "datasets/rgb/" folder and that python program converts the images into gray scale and save the images as a txt file into "datasets/gray/" folder.

3. The image is read from the "datasets/gray/" which are in text format. The image size is provided by the user and correspondingly an image in txt format is read from the folder.

4. The application applies FFT-2D and IFFT-2D and store the results in "results/fft_txt"

5. Than by running another python program which called read_txt_image.py, reads the result images from "results/fft_txt" file and convert them into png image files into "results/fft_png" folder.

4 GCP Implementation

4.1 Fat Cluster Setup

4	Name of the Setup	Region	Zone	Series	Machine Type	Memory	Persistant Disk
5	instance-1 (master)	us-central-lowa	us-central1-a	e2	custom-e2	16	10GB
6	instance-2 (worker)	us-central-lowa	us-central1-a	e2	custom-e2	16	10GB

Status	Name	Zone	Recommendations	In use by	Internal IP	External IP	Connect
	instance-1	us-central1-a			10.128.0.21 (nic0)	35.192.74.157 (nic0)	SSH
	instance-2	us-central1-a			10.128.0.24 (nic0)		SSH

4.2 Intra Regional Cluster Setup

9	Name of the Setup	Region	Zone	Series	Machine Type	Memory	Persistant Disk
10	instance-1 (master)	us-central-lowa	us-central1-a	e2	custom-e2	16	10GB
11	instance-2-16 (15 workers)	us-central-lowa	us-central1-a	e2	custom-e2	16	10GB

Filter Enter property name or value

Status	Name	Zone	Recommendations	In use by	Internal IP	External IP	Connect
	instance-1	us-central1-a			10.128.0.21 (nic0)	34.28.218.60 (nic0)	SSH
	instance-10	us-central1-a			10.128.0.37 (nic0)	34.123.233.99 (nic0)	SSH
	instance-11	us-central1-a			10.128.0.38 (nic0)	35.223.165.154 (nic0)	SSH
	instance-12	us-central1-a			10.128.0.39 (nic0)	104.155.175.248 (nic0)	SSH
	instance-13	us-central1-a			10.128.0.40 (nic0)	34.28.139.252 (nic0)	SSH
	instance-14	us-central1-a			10.128.0.41 (nic0)	34.136.125.157 (nic0)	SSH
	instance-15	us-central1-a			10.128.0.42 (nic0)	35.202.162.117 (nic0)	SSH
	instance-16	us-central1-a			10.128.0.43 (nic0)	35.193.72.250 (nic0)	SSH
	instance-2	us-central1-a			10.128.0.24 (nic0)	34.66.120.84 (nic0)	SSH
	instance-3	us-central1-a			10.128.0.12 (nic0)	34.172.7.232 (nic0)	SSH
	instance-4	us-central1-a			10.128.0.13 (nic0)	34.68.97.199 (nic0)	SSH
	instance-5	us-central1-a			10.128.0.14 (nic0)	34.68.66.210 (nic0)	SSH
	instance-6	us-central1-a			10.128.0.15 (nic0)	34.173.16.145 (nic0)	SSH
	instance-7	us-central1-a			10.128.0.22 (nic0)	34.135.111.3 (nic0)	SSH
	instance-8	us-central1-a			10.128.0.23 (nic0)	34.122.164.13 (nic0)	SSH
	instance-9	us-central1-a			10.128.0.36 (nic0)	35.193.201.247 (nic0)	SSH

4.3 Inter Regional Cluster Setup

Name of the Setup	Region	Zone	Series	Machine Type	Memory	Persistant Disk
instance-1 (master)	us-central-Iowa	us-central1-a	e2	custom-e2	16	10GB
node-3,10,12,13,14,15 (6 workers)	asia-south-Delhi	asia-south2-a	e2	custom-e2	16	10GB
node-16,17(2 workers)	australia-southeast	australia-south	e2	custom-e2	16	10GB
node-2,5,6,7,8,9(6 workers)	europe-west-Milan	europe-west8-a	e2	custom-e2	16	10GB
node-11 (1 worker)	us-central-Iowa	us-central1-a	e2	custom-e2	16	10GB

Status	Name	Zone	Recommendations	In use by	Internal IP	External IP	Connect
	aca-slave-copy-intercluster-light-1	europe-west8-a			10.198.0.3 (nic0)	34.154.187.219 (nic0)	SSH
	instance-1	us-central1-a			10.128.0.21 (nic0)	34.28.218.60 (nic0)	SSH
	instance-10	us-central1-a			10.128.0.37 (nic0)		SSH
	instance-11	us-central1-a			10.128.0.38 (nic0)		SSH
	instance-12	us-central1-a			10.128.0.39 (nic0)		SSH
	instance-13	us-central1-a			10.128.0.40 (nic0)		SSH
	instance-14	us-central1-a			10.128.0.41 (nic0)		SSH
	instance-15	us-central1-a			10.128.0.42 (nic0)		SSH
	instance-16	us-central1-a			10.128.0.43 (nic0)		SSH
	instance-2	us-central1-a			10.128.0.24 (nic0)		SSH
	instance-3	us-central1-a			10.128.0.12 (nic0)		SSH
	instance-4	us-central1-a			10.128.0.13 (nic0)		SSH
	instance-5	us-central1-a			10.128.0.14 (nic0)		SSH
	instance-6	us-central1-a			10.128.0.15 (nic0)		SSH
	instance-7	us-central1-a			10.128.0.22 (nic0)		SSH
	instance-8	us-central1-a			10.128.0.23 (nic0)		SSH
	instance-9	us-central1-a			10.128.0.36 (nic0)		SSH
	node10	asia-south2-a			10.190.0.3 (nic0)	34.131.25.170 (nic0)	SSH
	node11	us-central1-a			10.128.0.47 (nic0)	34.123.233.99 (nic0)	SSH
	node12	asia-south2-a			10.190.0.4 (nic0)	34.131.90.245 (nic0)	SSH
	node13	asia-south2-a			10.190.0.5 (nic0)	34.131.170.178 (nic0)	SSH
	node14	asia-south2-a			10.190.0.6 (nic0)	34.131.116.19 (nic0)	SSH
	node15	asia-south2-a			10.190.0.7 (nic0)	34.131.135.209 (nic0)	SSH
	node16	australia-southeast2-a			10.192.0.2 (nic0)	34.129.175.51 (nic0)	SSH
	node17	australia-southeast2-a			10.192.0.3 (nic0)	34.129.199.228 (nic0)	SSH
	node2	europe-west8-a			10.198.0.2 (nic0)	34.154.108.110 (nic0)	SSH
	node3	asia-south2-a			10.190.0.2 (nic0)	34.131.192.191 (nic0)	SSH
	node5	europe-west8-a			10.198.0.4 (nic0)	34.154.243.21 (nic0)	SSH
	node6	europe-west8-a			10.198.0.5 (nic0)	34.154.151.255 (nic0)	SSH
	node7	europe-west8-a			10.198.0.6 (nic0)	34.154.62.226 (nic0)	SSH

5 Testing and Debugging

1. We were having issues with sizes bigger than 256x256. Because of the stack size we were having that issue and once we make our systems stack size unlimited with `ulimit -s unlimited` command our problem solved for the local machine.

2. For the first time, when we create a new virtual machine, we have to manually establish a connection by ssh into the machine and then back to the source. This creates an entry in the known_hosts file in `.ssh` folder. And the next time, when we try to connect. There are no issues of such kind. For example: If we have 15 slaves and 1 master. We have to go to these machines one by one from our master node and create this connection.

3. The quota issues number with IN_USE_ADDRESSES (static IP range) region=us-central1 was increased from 24 to 48.

4. Connection time out error when deploying the inter-regional clusters.

5. The quota issue number with CPUS (region=us-central1) and CPUS_ALL_REGIONS was increased from 24 to 96.

6. In a single machine if we just run `fft_mpi.cpp` it gives us proper results. But if we want to do this in a light cluster we had to increase the stack size permanently in each of the machines by copying this command at these three files: In file `/etc/security/limits.conf`. The below-given command increases the hard and soft limit for the stack for all users to unlimited.

```
hard stack unlimited
soft stack unlimited
```

   In file `/etc/profile` and `/etc/bash.bashrc`, we added

```
ulimit -s unlimited
```

7. When we tried to run our mpi code by using multiple virtual machines in GCP with the following command:

```
mpirun -np 4 --hostfile hostfile
```

   We got this error:

```
ORTE was unable to reliably start one or more daemons.
```

   In order to resolve this problem, we used `--prefix` parameter in the `mpirun` command, which is shown below:

```
mpirun --prefix /usr/local/openMPI/ -np 8 --hostfile hostfile fft_mpi
```

8. When we open the ssh connection from GCP directly, it deletes the `authorized_keys` inside the `.ssh` folder and replaces it with its own key. To counter this problem, we have added a line of code in `.bashrc`

```
(cat .ssh/id_rsa.pub >> .ssh/authorized_keys)
```

   in all our nodes. We only access GCP ssh for the master so this issue remains limited to the master only for the time being.

6 Performance and Scalability Analysis

6.1 Execution Time:

The main purpose of parallelism is to reduce the execution time of a program. As a result of the experiment below charts were drawn for fat cluster, intra-regional and inter-regional light clusters with their execution time for the 5 different sized images. It is seen that not all the parallel configurations achieved the purpose of parallelism.

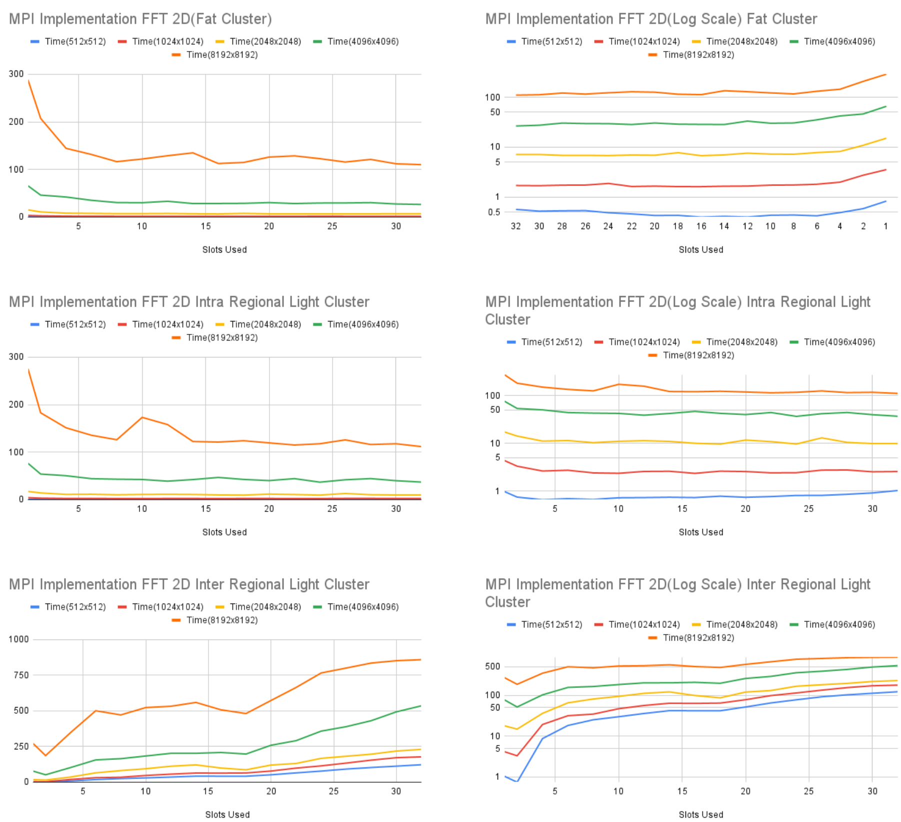

Figure 4: Execution time comparision Fat Cluster vs Light Intra Regional Cluster vs Light Inter Regional Cluster

6.2 Memory Occupancy:

```
8192x8192x16 = 2^31 = 2147.48365 megabytes
4096x4096x16 = 2^29 = 536.870912 megabytes
2048x2048x16 = 2^27 = 134.217728 megabytes
1024x1024x16 = 2^25 = 33.554432 megabytes
512x512x16   = 2^23 = 8.388608 megabytes
```

6.3 Speed Up

Using the figure 5, one can infer these things.

6.3.1 Fat Cluster

1. As we increase the size of the image, the speedup increases while keeping the slot size constant.

2. As we increase the number of slots while keeping the image size constant, the speedup increases till a point(around 14 slots) and then becomes constant.

6.3.2 Light Cluster(Intra Regional)

1. As we increase the size of the image, the speedup increases while keeping the slot size constant.

2. As we increase the number of slots while keeping the image size constant, the speedup increases till a point(around 14 slots) and then becomes constant.

6.3.3 Light Cluster(Inter Regional)

1. As we increase the size of the image, the speedup increases while keeping the slot size constant.

2. As we increase the number of slots while keeping the image size constant, the speedup decreases.

Note: On comparing fat cluster and light cluster, the speedup in fat cluster is more than speedup in light cluster for the same sized image(e.g 4096x4096 image) as shown in below given figure.

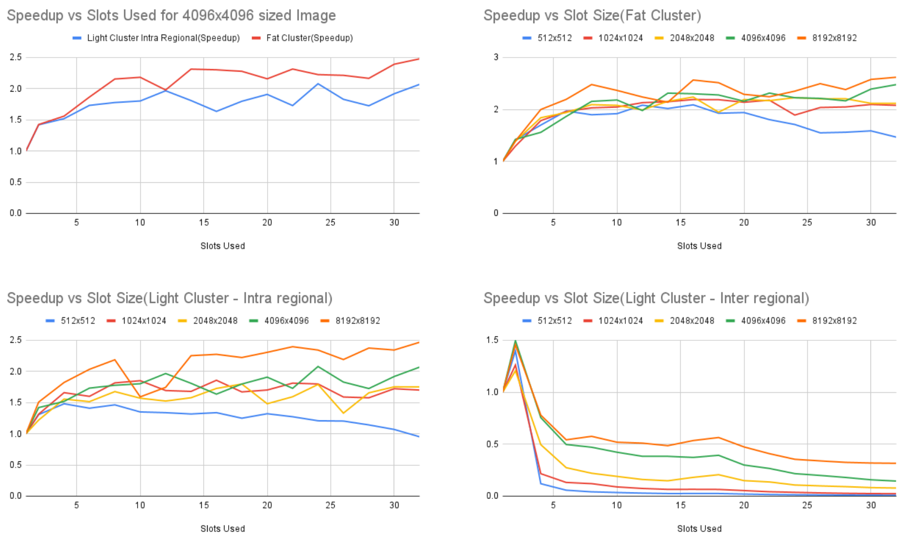

Figure 5: Speedup comparision for different type of clusters.

6.4 Strong Scalability

As per our hypothesis shown in the a-priori theoretical assessment, the execution time decreases as we increase the number of slots for a given fixed size image.(i.e 8192x8192). As predicted, when the number of slots reaches a certain point(14 and above in our case), the speedup becomes almost constant. This is due to an increase in MPI communication overhead. The increase in synchronization and communication time between the master and the slave process counters the faster execution done by each process on the sub-matrix.

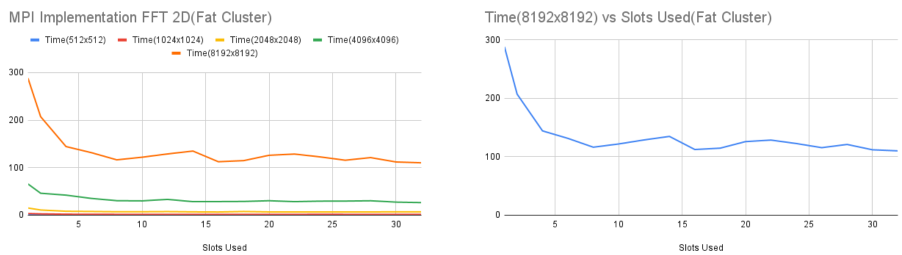

Figure 6: Strong Scalability for different size of images

6.5 Weak Scalability

6.5.1 Intra-Regional Cluster

As per our hypothesis shown in the a-priori theoretical assessment, the execution time decreases as we increase the number of slots for a given fixed size image.(i.e 8192x8192). As predicted, when the number of slots reaches a certain point(14 and above in our case), the speedup becomes almost constant. This behavior is similar to what we observe in strong scalability, but the communication overhead is more than the fat clusters because the distance between the master and the slave process is larger.

Figure 7: Weak Scalability(Intra) for different size of images

6.5.2 Inter-Regional Cluster

As per our hypothesis shown in the a-priori theoretical assessment, as we increase the number of cores for a fixed-sized image, our execution time should increase linearly. The effect is more prominent in inter-regional clusters than the intra-regional clusters because in inter-regional clusters the communication overhead is humungous in comparison to intra-regional clusters.

Figure 8: Weak Scalability(Inter) for different size of images

7 Conclusion

As parallel programming is mostly concerned about performance, a good project must also provide significant speedup.

1. For the fat cluster, with 8192x8192 image, we are able to achieve a speedup of 2.62 with 32 slots.

2. As shown in Performance and Scalability Analysis, we are able to scale our program using mpi to achieve strong scalability with fat cluster.

3. For the light cluster(intra regional), with 8192x8192 image, we are able to achieve a speedup of 2.47 with 32 slots.

4. As shown in Performance and Scalability Analysis, we are able to scale our program using mpi to achieve weak scalability with light cluster(intra-regional).

5. For the light cluster(inter regional), with 8192x8192 image, we are able to achieve a speedup of 0.31 with 32 slots.

6. As shown in Performance and Scalability Analysis, we are able to not able to scale our program using mpi to achieve weak scalability with light cluster(inter-regional).

8 Contributions

Topic	Shubham Shubhankar Sharma	Ismail Kerem Tatlici
A priori study	50%	50%
FFT Serial Code	40%	60%
Python Code	30%	70%
FFT MPI Code	70%	30%
Tests	50%	50%
Performance and Scalability	70%	30%
Profiling	40%	60%
Google Cloud	60%	40%
Final Report And Presentation	50%	50%

Table 3: Comparison of individual contributions

9 References

https://en.wikipedia.org/wiki/Fast_Fourier_transform