University of Pavia

2D-FFT Parallalization Using Open MPI

Advanced Computer Architecture Report

Bernardini Federico Pietro, Zamboni Fabio

21 September 2023

Abstract

The aim of the project is to create a serial code capable of performing 2D FFT (Fast Fourier Transform) on a PGM (Portable Gray Map) P2 type image and then Improve the performance achieved by code parallelization through Open MPI, eventually performances are tested using Google Cloud Platform.

1 Serial Implementation

1.1 Cooley-Tukey 1D-FFT Algorithm

We decided to use the Cooley-Tukey iterative algorithm to compute the FFT because is the most common one and much documentation about it is available for free online.
The algorithm is divided into two different part:

Bit Reversal: The algorithm requires the division of signal elements into even and odd indices, the bit reverse operation is a smart way to execute this operation (example: $001 \rightarrow 100$).

1D-FFT computation:

```
1  // Function to perform Cooley-Tukey FFT
2  // x = input vector
3  // N = size of the input vector
4  // Forwards if inverse = 0, backwards if inverse = 1
5  void cooley_tukey_fft(cplx x[], int N, int inverse) {
6      // Iterative FFT or IFFT
7      double sign = (inverse) ? 1.0 : -1.0; // Sign for IFFT
8      for (int s = 1; s <= log2(N); s++) {
9          int m = 1 << s; // Subproblem size
10         cplx omega_m = cexp(sign * I * 2.0 * PI / m);
11
12         for (int k = 0; k < N; k += m) {
13             cplx omega = 1.0;
14
15             for (int j = 0; j < m / 2; j++) {
16                 cplx t = omega * x[k + j + m / 2];
17                 cplx u = x[k + j];
18                 x[k + j] = u + t;
19                 x[k + j + m / 2] = u - t;
20                 omega *= omega_m;
21             }
22         }
23     }
24 }
```

In line 7 we check whether we should perform the direct or inverse FFT. To understand the basic functioning of the rest of the code it is better to have a graphic reference, Fig.[1].
The first for loop tells us how many `s` steps we have to take, in the example $\mathtt{s} = log_2(\mathtt{N}) = 2$, `m` tells

us how many arrows cross occur in each sub-problem for each step, in the example we have m=2 in step 1 (the red and orange arrows) and m=4 in step 2 (the blue arrows). The rest of the loops compute the twiddle factors $W_N^k = e^{-(j2\pi k)/N}$, where $k = 0, \ldots, (m/2) - 1$ for each step, and then execute the summation between the elements following the arrows direction. (More mathematical details in Appendix 1)

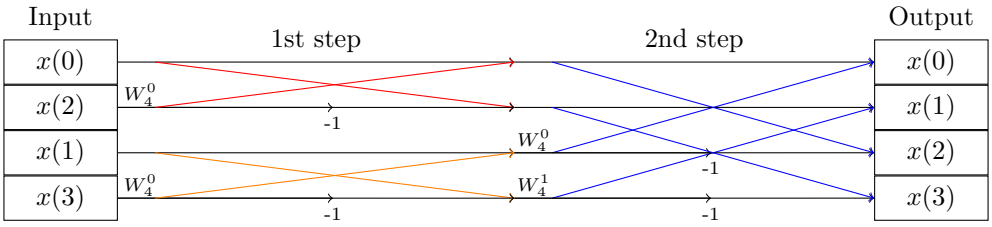


Figure 1: Basic FFT scheme for a N-size signal (N=4).

1.2 Serial Code Analysis

The FFT function cannot work with images of any size, an a-priori operation is required to use it correctly: bring the image to a size $2^n \times 2^n$ pixels.
That being said, let's see a detailed description of the Fig.[2] steps:

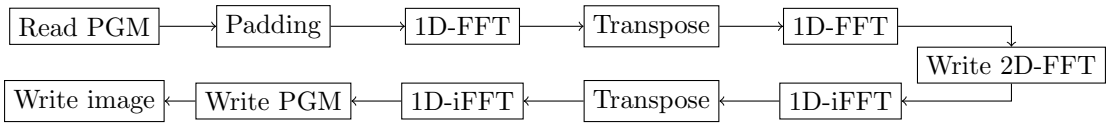


Figure 2: Data diagram of the serial process.

1. Firstly we read the PGM image and save its features (format, dimensions, maximum value and pixels value) in a structure.
2. Next we check if the image has the proper dimensions to be computed by the 1D-FFT and if not compute the zero-padding on it, which consists of adding 0-value pixels in the image in order to reach the desired dimensions.
3. We transform the padded-image matrix into a vector and through a loop feed the 1D-FFT with an amount of data equal to the row length per cycle.
4. Next we compute the transpose. This step is necessary in order to compute the second 1D-FFT on the columns in a memory-reading efficient way.
5. Compute again the 1D-FFT, obtaining the overall 2D-FFT.
6. Now we perform the circular shift on the transformed image in order to have a reordered output in term of frequencies, write a new PGM image containing 2D-FFT result and compute the inverse circular shift.
7. In the next three steps we compute the inverse 2D-FFT (2D-iFFT).
8. The last step is write a PGM image for the 2D-iFFT in order to check if the previous steps are correct (the final image does not contain the padding).

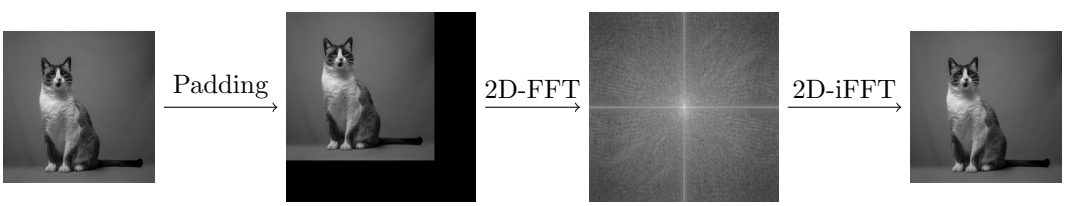


Figure 3: Image processing during code execution.

2 Parallel Implementation

2.1 Possible Strategies

We thought of two possible strategies to be implemented:

- The first idea is to divide the image into sub-images and perform the 1D-FFT algorithm in parallel on each of them. In Fig.[4] we can see the data flowchart of this strategy. (Implemented)
- The second idea is an improvement of the previous one, it consists in performing the transpose matrix without gathering together the sub-images after the first 1D-FFT.
  Unfortunately this idea has two drawbacks: (1) lot of communications between the processors are required; (2) the algorithm for a general number of processors is quite complex, so we prefer not to use it without fully understanding it.

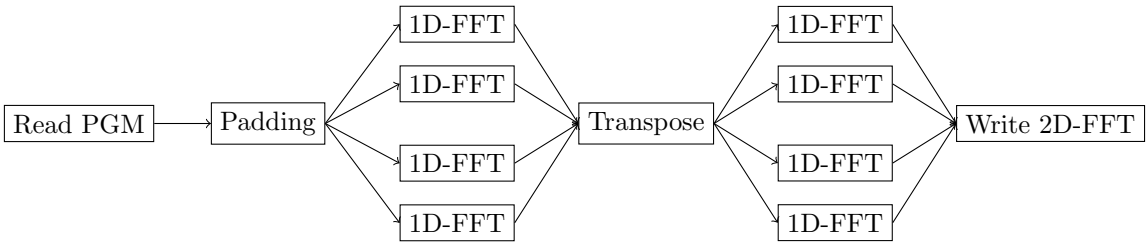


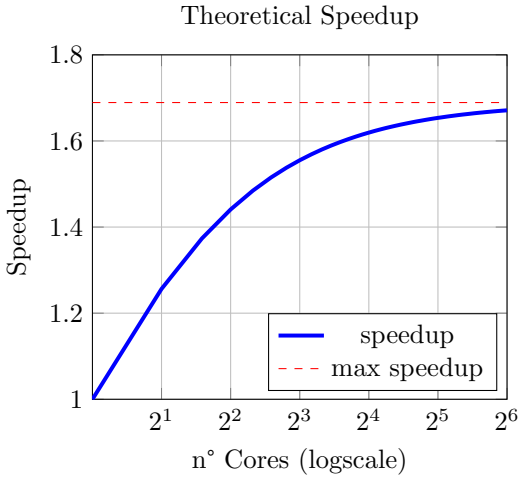
Figure 4: Data diagram of the implemented parallel process

2.2 A-priori Study of Parallelism

Using the default MacOS profiler we were able to discover the execution times of each function and compute the theoretical speedup, Fig.[5], through the Amdahl's law

$$\text{Speedup(N)} = \frac{1}{S + P/N} = \frac{1}{0.592 + 0.408/N} \tag{1}$$

where N is the number of cores, P is the fraction of code that can be parallelized and S is the serial fraction of code.



Function Name	Runtime%
cooley_tukey_fft	40.8%
pgm_write_fft	31.5%
pgm_write	11.9%
pgm_read	4.4%
transpose	3.9%
vet2mat	2.0%
mat2vet	0.8%
zeroPadding	0.7%
fftshift	0.6%
others	3.4%

Figure 5: Theoretical speedup computed through the Amdahl's law (right) and MacOS profiler results (left).

The maximum theoretical speedup is 1.689 but expectations are to have a lower outcome because of communication delays, so for a good result it is necessary to minimize the exchange of information between processors without increasing too much the performed operations and to balance the workload between processors.

2.3 MPI Implementation

In this section we describe the MPI implementation hot spots focusing on the main communication & synchronization aspects.
The main idea is to divide the computation of the 2D-FFT among all existing processors. We decided to let the processor 0 (the master) read the image, if needed make the padding, and broadcast the image metadata (width, height and pixels max value) to all the processors.

```
// Broadcast the usefull length information
MPI_Bcast(len_info, 5, MPI_INT, 0, MPI_COMM_WORLD);
```

Then we used this metadata in order to distribute the data among all the processors, taking into account the fact that data are not always divisible by the number of processors. We managed to solve this issue distributing extra rows to other processors starting from the master. It is important to notice that in this way each processor will have at most 1 more row than the processor with the least number of rows. The implementation is in the following piece of code.

```
// Find the number of rows per processor
int rows_per_processor = len_info[1] / size;
int remainder = len_info[1] % size;

// Find the number of processors that will receive an extra row
int processors_with_extra_rows = (rank < remainder) ? 1 : 0;

// Find the number of rows to send to this processor
int my_num_rows = rows_per_processor + processors_with_extra_rows;

// Calculate the displacements for MPI_Scatterv
int* displacements = (int*)malloc(size * sizeof(int));
int* recvcounts = (int*)malloc(size * sizeof(int));

// Calculate the displacements and the recvcounts
int displacement = 0;
for (int i = 0; i < size; i++) {
    recvcounts[i] = rows_per_processor * len_info[0];
    if (i < remainder) {
        recvcounts[i] += len_info[0]; // Distribute remaining rows
    }
    displacements[i] = displacement;
    displacement += recvcounts[i];
}
```

After calculating `my_num_rows` and `displacements` we used them as argument for `MPI_Scatterv()`; here the displacement is the index from where the data will be sent.

```
// Scatter the data
MPI_Scatterv(v_send, recvcounts, displacements, MPI_C_DOUBLE_COMPLEX, v_revc,
    my_num_rows * len_info[0], MPI_C_DOUBLE_COMPLEX, 0, MPI_COMM_WORLD);
```

Now, after having sent the portion of data to the processors, it is time to perform the 1D-FFT on each row and then to collect the data using `MPI_Gatherv()` in order to make the transpose of the image using processor 0. Then we scatter again the data and perform the 1D-FFT on rows.

```
// Gather the data
MPI_Gatherv(v_revc, my_num_rows * len_info[0], MPI_C_DOUBLE_COMPLEX, v_send,
    recvcounts, displacements, MPI_C_DOUBLE_COMPLEX, 0, MPI_COMM_WORLD);
```

After that, we have our 2D-FFT to be written by processor 0 in *fft.pgm*. Finally we run the 2D-iFFT using the same pipeline as before and, collecting the data in processor 0, divide the data by the total number of data. Eventually 2D-iFFT is written in *ifft.pgm* by processor 0.

2.4 Test & Debugging

To carry out the tests we used 17 different size images in total: 8 taken from the internet [7], [8] and 9 generated by us. Internet images were used to debug the code, the sizes chosen cover a wide range from 24×7 to 5184×3456 pixels. Those generated by us represent only white noise, but this is not a problem since performances are related just to the size of the image and not to the pixels value. We were forced to make this choice because on the internet there are really few examples of PGM images and, for some of our tests, we needed precise image dimensions.

3 Performance and Scalability Analysis

To study scalability we used the Google Cloud Platform, thanks to it we were able to run our code in parallel on multiple virtual machines (VMs) set up by us.
We created two different types of clusters, a "fat cluster" consisting of a few but powerful VMs and a "light cluster" consisting of more but less powerful VMs. Another differentiation made on our clusters is between "intraregional", if all the VMs belong to the same region, or "infraregional" if different regions are involved. Our clusters were set as follow

- Fat Cluster - Intraregional: 2 VMs with 8 vCores each in region `us-central1`.
- Fat Cluster - Infraregional: 2 VMs with 8 vCores each, one in region `us-central1` and the other one in region `europe-west1`.
- Light Cluster - Intraregional: 8 VMs with 2 vCores each in region `us-central1`.
- Light Cluster - Infraregional: 8 VMs with 2 vCores each, four in region `us-central1` and four in region `europe-west1`.

3.1 Strong Scalability Analysis

As first we studied "strong scalability," that is how an increasing number of vCores perform on a fixed size problem; the image on which the tests were done had size 1920x1280 pixels.

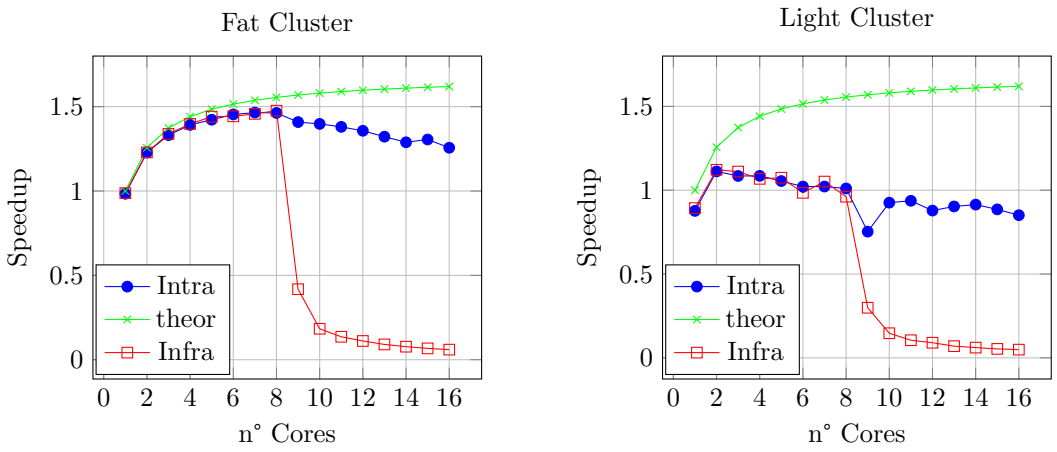


Figure 6: Fat and light cluster: intraregional, infraregional and theoretical speedup.

In Fig.[6] we can see that up to 8 cores the fat cluster performances are practically the same as the theoretical case and then decreases noticeably. This behavior is probably due to the fact that when more than 8 cores are required, communication between VMs begins and with it delays due to communication.
In the light cluster case the problem arises after just 2 cores, but this is not surprising given that every single VM has maximum this quantity.
The first surprising result is how bad the performance drops, compared to what we expected, when the communication happens between two different regions, this probably happens due to the large distance between regions added to the problems already encountered.

3.2 Weak Scalability Analysis

For the "weak scalability" analysis, that is increase the size of the problem proportionally to the computational power, we used the nine images created by us in order to have the correct size to respect the ratio between n° of cores and pixels (respected ratio: $n°Pixels/n°Cores = 10^4$).

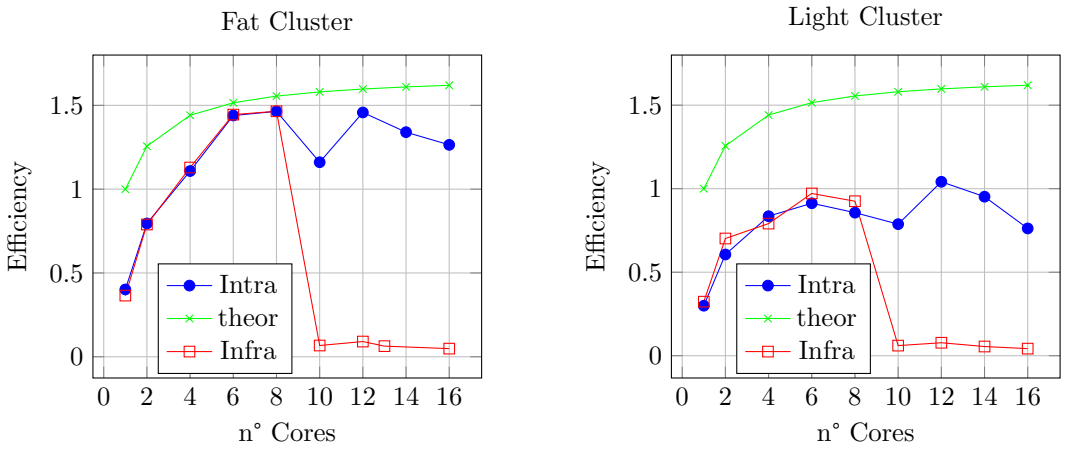


Figure 7: Fat and light cluster: intraregional & infraregional efficiency and theoretical speedup.

In the case of weak scalability, it makes more sense to talk about efficiency, i.e., how well the additional resources are exploited when the problem becomes larger, rather than speedup.
In Fig.[7] we can see that for the fat cluster results scale very well until 8 cores, where they settle down or, in the infraregional case, totally collapse.
In the light cluster design performances are similar to that of the fat cluster but with far worse results, expecially in the infraregional case.
Plausibly, the causes of performances degradation are the same as those hypothesized for strong scalability.

4 Conclusions

The implemented parallelization strategy provides good results using few powerful VMs in the same region. If it were possible to further reduce communications between machines, or speed them up, also infraregional performances should increase significantly.
From the code point of view, some possible improvements could be:

- Use only matrices, avoiding matrix-vector transformations and vice versa.
- Use optimized libraries to read and write images with the possibility of extending input formats.
- Parallelize other functions, such as the reading and writing ones.

Individual Contributions

Most of the code-writing work was done together, and it is not possible to split individual contributions; the same is true for the testing done on Google Cloud Platform. The only tasks attributable to individuals are:

- Bernardini P. Federico: Writing the report on LaTeX.
- Zamboni Fabio: Creating and maintaining the repository on GitHub.

Individual actions, while such, were still supervised by both.

Appendix 1

Mathematical Insight Of The FFT

We have added this extra section for people who want to delve into the idea behind the FFT algorithm from a mathematical point of view.

The main idea is to exploit the symmetry of the Fourier matrix to decompose it into increasingly sparse matrices, let's look more technically what we are talking about starting with the definition of the DFT in matrix form

$$X_n = F_n * x_n \tag{2}$$

where $X_n \in \mathbb{C}^n$ is the Fourier transformed elements vector, $x_n \in \mathbb{C}^n$ is the inputs vector and $F_n \in \mathbb{C}^{n\times n}$ is the Fourier matrix

$$F_n = \frac{1}{\sqrt{n}} \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega & \cdots & \omega^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \cdots & \omega^{(n-1)^2} \end{bmatrix} \tag{3}$$

where $\omega = e^{-j\frac{2\pi}{n}}$.
The actual benefit in the computations occurs in the following decomposition of the F_n matrix

$$F_n = \frac{1}{\sqrt{n}} \begin{bmatrix} I_{n/2} & D_{n/2} \\ I_{n/2} & -D_{n/2} \end{bmatrix} \begin{bmatrix} F_{n/2} & 0 \\ 0 & F_{n/2} \end{bmatrix} P_n \tag{4}$$

where $I_{n/2} \in \mathbb{R}^{n/2\times n/2}$ is the identity matrix, $D_{n/2} \in \mathbb{C}^{(n/2)\times(n/2)}$ is a diagonal matrix defined as follow and $P_n \in \mathbb{R}^{n\times n}$ is a permutation matrix that allow us to split the even and odd elements

$$D_{n/2} = diag\left\{1, \omega, \omega^2, \cdots, \omega^{(n/2)-1}\right\} \qquad P_n = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 1 & 0 & 0 & \cdots & 0 & 1 \end{bmatrix}$$

It is possible to divide the F_n matrix until we reach the desired sparsity, in this way computational complexity decrease from $O(n^2)$ to $O(n * log_2(n))$, in Fig.[8] we can see how great is the benefit as n increases. Finally to perform the 2D-FFT we can just compute the 1D-FFT on each of the image dimensions one at a time.

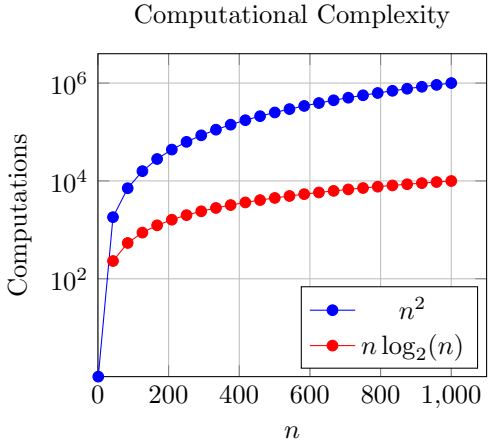


Figure 8: Computational complexity varying the signal size n.

References

GitHub Link

[1] Zamboni F. Bernardini F.P. "GitHub Repository". In: (). URL: https://github.com/zamb0/2D_Fast_Fourier_Transform_OpenMPI.git.

FFT Implementation & Code Sources

[2] Eleanor Chu and Alan George. *FFT algorithms and their adaptation to parallel processing*. Vol. 284. 1. International Linear Algebra Society (ILAS) Symposium on Fast Algorithms for Control, Signals and Image Processing. 1998, pp. 95–124. DOI: https://doi.org/10.1016/S0024-3795(98)10086-1. URL: https://www.sciencedirect.com/science/article/pii/S0024379598100861.

[3] *Fourier Transform (And Inverse) Of Images*. 2016. URL: https://blog.demofox.org/2016/07/28/fourier-transform-and-inverse-of-images/.

[4] *Iterative Fast Fourier Transformation for polynomial multiplication*. 2023. URL: https://www.geeksforgeeks.org/iterative-fast-fourier-transformation-polynomial-multiplication/.

[5] *OpenMPI Documentation*. URL: https://www.open-mpi.org/doc/current/.

[6] A. Walker R. Fisher S. Perkins and E. Wolfart. *Logarithm Operator*. 2003. URL: https://homepages.inf.ed.ac.uk/rbf/HIPR2/pixlog.htm.

[9] G. Strang. *26. Complex Matrices; Fast Fourier Transform*. 2005. URL: https://www.youtube.com/watch?v=M0Sa8fLOajA.

Test & Debug Materials

[7] First Image Source. URL: https://filesamples.com/formats/pgm?utm_content=cmp-true#google_vignette.

[8] Second Image Source. URL: https://people.sc.fsu.edu/~jburkardt/data/pgma/pgma.html.