CSCD71 Final Project

Aryan Jain

The Project is based on building a parallel code for the Fast Fourier Transform and using it to De-noise signals. We project is based on using both shared memory parallelization using OPENMP and hybrid memory parallelization using MPI+OPENMP .We present scaling analysis reports for both of the parallel implementations.

Fast Fourier Transform

The Fast Fourier transform is an mathematical algorithm that is used to compute the DFT - Discrete Fourier Transform. The DFT is an algorithm that convert signals from their original domain to their frequence domain. In this project we build a parallel FFT Algorithm that is used to denoise signals.

The Fourier Transform

The Discrete Fourier Transform can be expressed as

$$\mathbf{F}(n) = \sum_{k=0}^{N-1} f(k)\, e^{-j2\pi nk/N} \qquad (n = 0..1 \;:N-1)$$

The relevant inverse Fourier Transform can be expresse

$$f(k) = \frac{1}{N} \sum_{n=0}^{N-1} \mathbf{F}(n)\, e^{j2\pi nk/N} \qquad (k = 0, 1, 2\ldots, N-1)$$

Discrete Fourier Transform

$$\hat{f}_k = \sum_{j=0}^{n-1} f_j e^{-i2\pi jk/n}, \tag{2.26}$$

and the inverse discrete Fourier transform (iDFT) is given by:

$$f_k = \frac{1}{n} \sum_{j=0}^{n-1} \hat{f}_j e^{i2\pi jk/n}. \tag{2.27}$$

Thus, the DFT is a linear operator (i.e., a *matrix*) that maps the data points in f to the frequency domain $\hat{\mathbf{f}}$:

$$\{f_1, f_2, \cdots, f_n\} \quad \stackrel{\text{DFT}}{\Longrightarrow} \quad \{\hat{f}_1, \hat{f}_2, \cdots \hat{f}_n\}. \tag{2.28}$$

For a given number of points n, the DFT represents the data using sine and cosine functions with integer multiples of a fundamental frequency, $\omega_n = e^{-2\pi i/n}$. The DFT may be computed by matrix multiplication:

$$\begin{bmatrix} \hat{f}_1 \\ \hat{f}_2 \\ \hat{f}_3 \\ \vdots \\ \hat{f}_n \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_n \end{bmatrix}. \tag{2.29}$$

Fast Fourier Transform

$$\hat{\mathbf{f}} = \mathbf{F}_{1024}\mathbf{f} = \begin{bmatrix} \mathbf{I}_{512} & -\mathbf{D}_{512} \\ \mathbf{I}_{512} & -\mathbf{D}_{512} \end{bmatrix} \begin{bmatrix} \mathbf{F}_{512} & \mathbf{0} \\ \mathbf{0} & \mathbf{F}_{512} \end{bmatrix} \begin{bmatrix} \mathbf{f}_{\text{even}} \\ \mathbf{f}_{\text{odd}} \end{bmatrix}, \tag{2.30}$$

where $\mathbf{f}_{\text{even}}$ are the even index elements of $\mathbf{f}$, $\mathbf{f}_{\text{odd}}$ are the odd index elements of $\mathbf{f}$, $\mathbf{I}_{512}$ is the 512×512 identity matrix, and $\mathbf{D}_{512}$ is given by

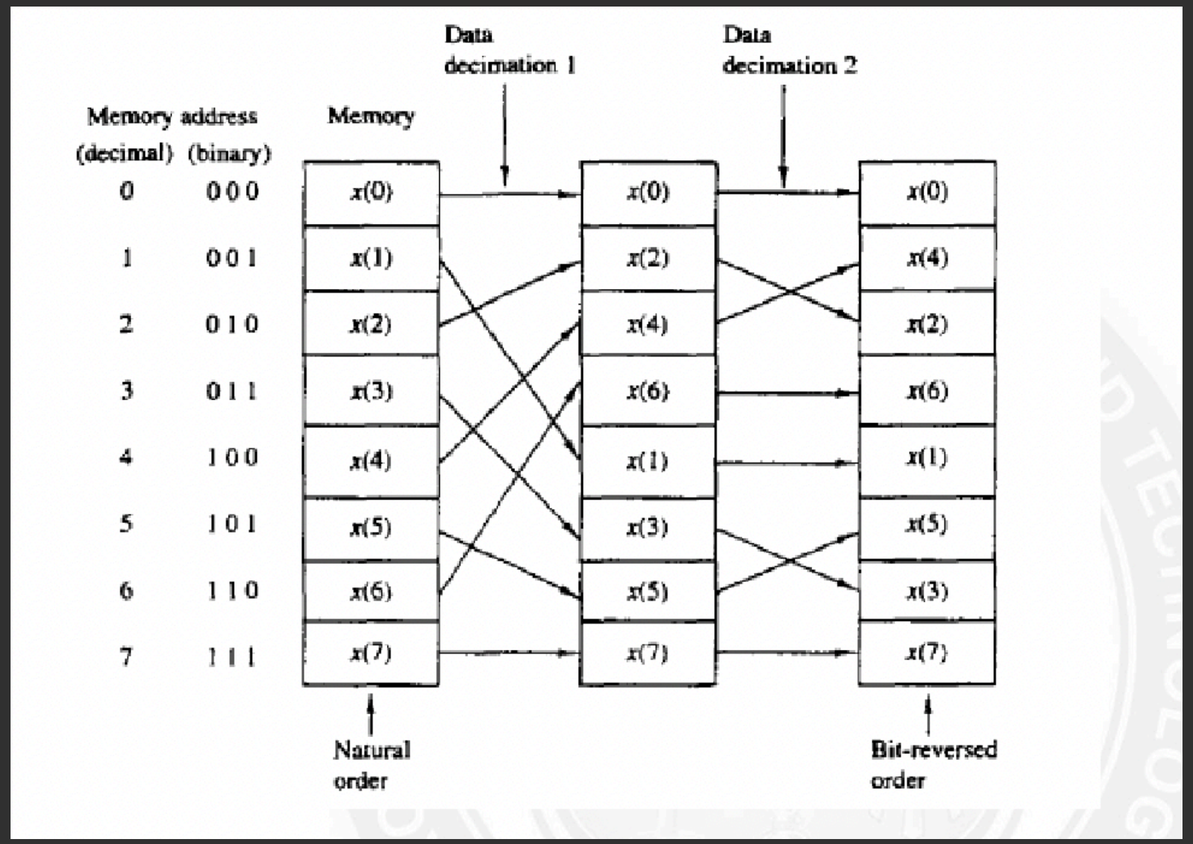
$$\mathbf{D}_{512} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & \omega & 0 & \cdots & 0 \\ 0 & 0 & \omega^2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \omega^{511} \end{bmatrix}. \tag{2.31}$$

Bit Reversal to Find F(even) and F(odd)

0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

The indices of the input complex numbers are bit reversed to rearrange the input vector into f(even) and f(odd) indices .

```
int reverse_bit(int value, int N) {

    int ret = 0;
    int i = 0;

    while (i < N) {
        ret <<= 1;
        ret |= (value>>i) & 1;
        i++;
    }

    return ret;
}


// Calc WN[], with N = input_N
complex* Calc_WN(int N) {

    cout << "Calculating WN[] of N = " << N << " ... \t";
    complex* WN = new complex[N];

    complex WN_unit;
        WN_unit.re = cos(2*PI/N);
        WN_unit.im = -sin(2*PI/N);
    WN[0].re=1; WN[0].im=0;
    #pragma omp parallel for
    for (int i = 1; i < N; ++i)
    {
        WN[i] = ComplexMul(WN[i-1], WN_unit);
    }

    return WN;
}
```



We use Decimate in Time FFT Algorithm which uses bit reversal to calculate the DFT matrix.

- Divide and Conquer:
- DFT matrix is decomposed into smaller sub-problems.
- Splits input into even and odd-indexed components recursively.
- Butterfly Operations:
- Combine smaller DFT results using "butterfly" computations.
- Utilize twiddle factors WNk=e−2πik/NW_N^k = e^{-2\pi i k / N}WNk=e−2πik/N.
- Bit-Reversal Permutation:
- Rearrange the input sequence indices in bit-reversed order.
- Ensures efficient access to data for the recursive computation.

Recursive FFT Algorithm

```
X_0,...,N-1 ← ditfft2(x, N, s):              DFT of (x_0, x_s, x_2s, ..., x_(N-1)s):
    if N = 1 then
        X_0 ← x_0                                    trivial size-1 DFT base case
    else
        X_0,...,N/2-1 ← ditfft2(x, N/2, 2s)          DFT of (x_0, x_2s, x_4s, ...)
        X_N/2,...,N-1 ← ditfft2(x+s, N/2, 2s)        DFT of (x_s, x_s+2s, x_s+4s, ...)
        for k = 0 to N/2-1                           combine DFTs of two halves into full D
            t ← X_k
            X_k ← t + exp(-2πi k/N) X_k+N/2
            X_k+N/2 ← t - exp(-2πi k/N) X_k+N/2
        endfor
    endif
```

The recursive code has an underlying problem of being difficult to parallelize using openmpi this is because the recursive calls have a lot of parallel overhead. So in this project we compare the parallel recursive code with the parallel iterative code and (MPI+OPENMPI) iterative code.

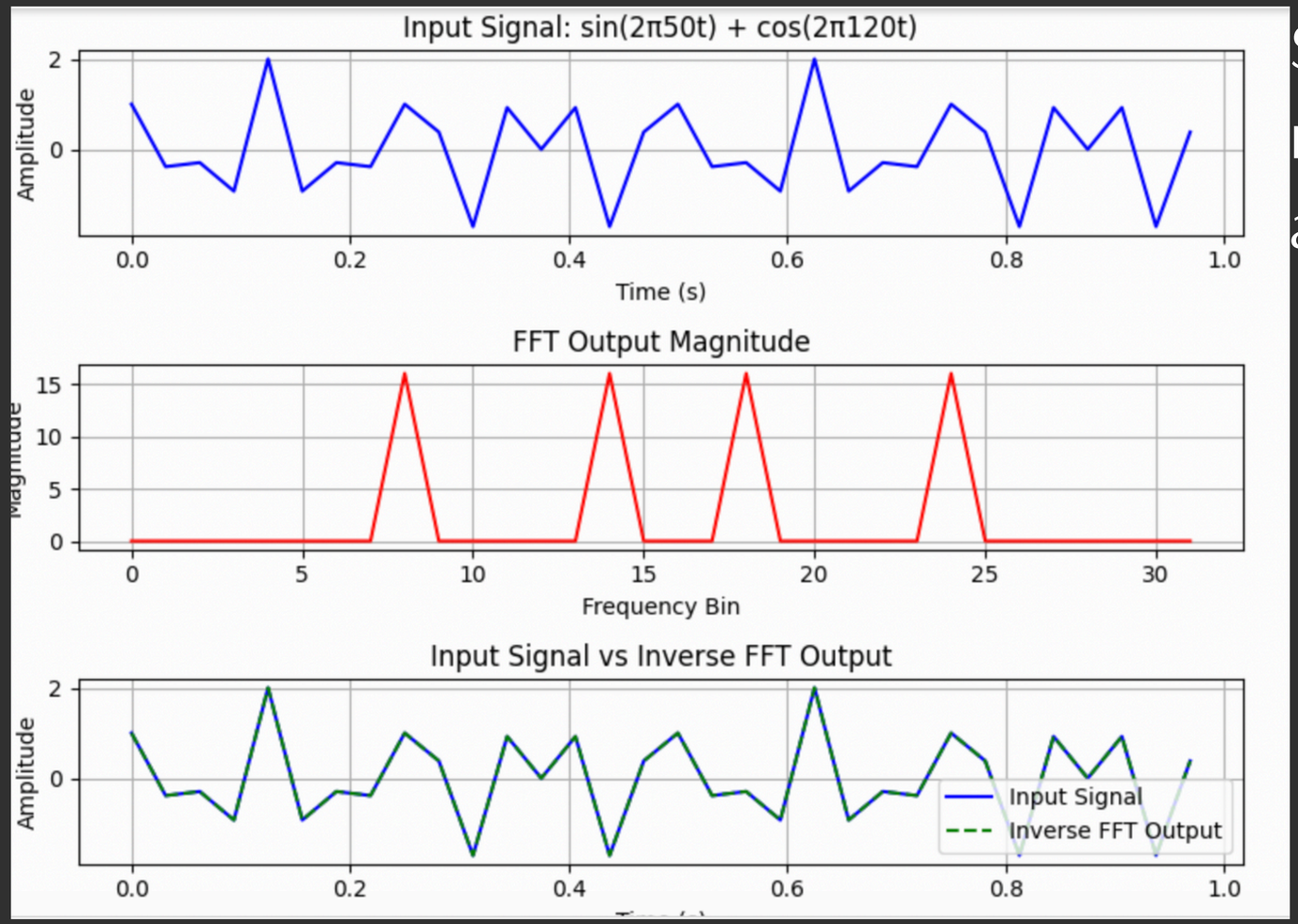
```cpp
complex* DIT_FFT_iterative(complex input_seq[], int N, complex WN[]) {
    complex* return_seq = new complex[N];
    complex* current_seq = input_seq;

    int stages = static_cast<int>(log2(N));

    for (int stage = 1; stage <= stages; ++stage) {
        int m = 1 << stage;  // 2^stage
        int half_m = m / 2;

        #pragma omp parallel for schedule(dynamic)
        for (int i = 0; i < N / 2; ++i) {
            int k = (i / half_m) * m;
            int j = i % half_m;
            int idx1 = k + j;
            int idx2 = k + j + half_m;

            complex W = WN[(j * N) / m];
            complex t = ComplexMul(current_seq[idx2], W);
            return_seq[idx1] = ComplexAdd(current_seq[idx1], t);
            return_seq[idx2] = ComplexSub(current_seq[idx1], t);
        }

        std::swap(current_seq, return_seq);
    }

    return current_seq;
}
```

Iterative FFT

```cpp
complex* DIT_FFT(complex input_seq[], int N, complex WN[], int recur_time_count) {
    complex* return_seq = new complex[N];

    if (N == 2) {
        // Base case: simple FFT computation
        return_seq[0] = ComplexAdd(input_seq[0], ComplexMul(input_seq[1], WN[0]));
        return_seq[1] = ComplexAdd(input_seq[0], ComplexMul(ReverseComplex(input_seq[1]), WN
        return return_seq;
    }

    // Split the input sequence into halves
    int k = pow(2, recur_time_count);
    complex* first_half_input_seq = new complex[N / 2];
    complex* second_half_input_seq = new complex[N / 2];

    #pragma omp parallel for
    for (int i = 0; i < N / 2; ++i) {
        first_half_input_seq[i] = input_seq[i];
        second_half_input_seq[i] = input_seq[i + N / 2];
    }

    complex* DFTed_first_half_seq = nullptr;
    complex* DFTed_second_half_seq = nullptr;

    #pragma omp taskgroup
    {
        // Create tasks for the recursive FFT calls
        #pragma omp task shared(DFTed_first_half_seq)
        DFTed_first_half_seq = DIT_FFT(first_half_input_seq, N / 2, WN, recur_time_count + 1

        #pragma omp task shared(DFTed_second_half_seq)
        DFTed_second_half_seq = DIT_FFT(second_half_input_seq, N / 2, WN, recur_time_count +
    }  // Implicit barrier: waits for all tasks to complete

    complex* output_first_half_seq = new complex[N / 2];
    complex* output_second_half_seq = new complex[N / 2];
```

Recursive FFT

Output of The Algorithm



Since the input signal and inverse FFT signal match we get a way to verify that our FFT and inverse FFT algorithm is correct.

We use Two implementation methods to build the FFT and Inverse FFT Algorithm. The first Algorithm is recursive and the other is iterative.

The Algorithm computes both FFT and inverse FFT for sampling input of the form (2^n)

Serial Scaling Analysis

```
Serial Scaling Analysis Results
k       N       Time (ms)
Running for N = 256 (k = 8)...
Reorder the sequence ...        Calculating WN[] of N = 256 ...         Reorder the sequence ...        Run time = 0 ms
Running for N = 512 (k = 9)...
Reorder the sequence ...        Calculating WN[] of N = 512 ...         Reorder the sequence ...        Run time = 0 ms
Running for N = 1024 (k = 10)...
Reorder the sequence ...        Calculating WN[] of N = 1024 ...        Reorder the sequence ...        Run time = 0 ms
Running for N = 2048 (k = 11)...
Reorder the sequence ...        Calculating WN[] of N = 2048 ...        Reorder the sequence ...        Run time = 0 ms
Running for N = 4096 (k = 12)...
Reorder the sequence ...        Calculating WN[] of N = 4096 ...        Reorder the sequence ...        Run time = 0 ms
Running for N = 8192 (k = 13)...
Reorder the sequence ...        Calculating WN[] of N = 8192 ...        Reorder the sequence ...        Run time = 0 ms
Running for N = 16384 (k = 14)...
Reorder the sequence ...        Calculating WN[] of N = 16384 ...       Reorder the sequence ...        Run time = 0 ms
Running for N = 32768 (k = 15)...
Reorder the sequence ...        Calculating WN[] of N = 32768 ...       Reorder the sequence ...        Run time = 10 ms
Running for N = 65536 (k = 16)...
Reorder the sequence ...        Calculating WN[] of N = 65536 ...       Reorder the sequence ...        Run time = 10 ms
Running for N = 131072 (k = 17)...
Reorder the sequence ...        Calculating WN[] of N = 131072 ...      Reorder the sequence ...        Run time = 40 ms
Running for N = 262144 (k = 18)...
Reorder the sequence ...        Calculating WN[] of N = 262144 ...      Reorder the sequence ...        Run time = 60 ms
Running for N = 524288 (k = 19)...
Reorder the sequence ...        Calculating WN[] of N = 524288 ...      Reorder the sequence ...        Run time = 140 ms
Running for N = 1048576 (k = 20)...
Reorder the sequence ...        Calculating WN[] of N = 1048576 ...     Reorder the sequence ...        Run time = 440 ms
Running for N = 2097152 (k = 21)...
Reorder the sequence ...        Calculating WN[] of N = 2097152 ...     Reorder the sequence ...        Run time = 1170 ms
Running for N = 4194304 (k = 22)...
Reorder the sequence ...        Calculating WN[] of N = 4194304 ...     Reorder the sequence ...        Run time = 2520 ms
Running for N = 8388608 (k = 23)...
Reorder the sequence ...        Calculating WN[] of N = 8388608 ...     Reorder the sequence ...        Run time = 5390 ms
Running for N = 16777216 (k = 24)...
Reorder the sequence ...        Calculating WN[] of N = 16777216 ...    Reorder the sequence ...        Run time = 11560 ms
Running for N = 33554432 (k = 25)...
Reorder the sequence ...        Calculating WN[] of N = 33554432 ...    Reorder the sequence ...        Run time = 22440 ms
Running for N = 67108864 (k = 26)...
Reorder the sequence ...        Calculating WN[] of N = 67108864 ...    Reorder the sequence ...        Run time = 45970 ms
Running for N = 134217728 (k = 27)...
Reorder the sequence ...        Calculating WN[] of N = 134217728 ...   Reorder the sequence ...        Run time = 74890 ms
```

Iterative Code

```
Serial Scaling Analysis Results
k       N       Time (ms)
Running for N = 256 (k = 8)...
Calculating WN[] of N = 256 ...         Reorder the sequence ...        Reorder the sequence ...        Run time = 0 ms
Running for N = 512 (k = 9)...
Calculating WN[] of N = 512 ...         Reorder the sequence ...        Reorder the sequence ...        Run time = 0 ms
Running for N = 1024 (k = 10)...
Calculating WN[] of N = 1024 ...        Reorder the sequence ...        Reorder the sequence ...        Run time = 0 ms
Running for N = 2048 (k = 11)...
Calculating WN[] of N = 2048 ...        Reorder the sequence ...        Reorder the sequence ...        Run time = 0 ms
Running for N = 4096 (k = 12)...
Calculating WN[] of N = 4096 ...        Reorder the sequence ...        Reorder the sequence ...        Run time = 0 ms
Running for N = 8192 (k = 13)...
Calculating WN[] of N = 8192 ...        Reorder the sequence ...        Reorder the sequence ...        Run time = 0 ms
Running for N = 16384 (k = 14)...
Calculating WN[] of N = 16384 ...       Reorder the sequence ...        Reorder the sequence ...        Run time = 10 ms
Running for N = 32768 (k = 15)...
Calculating WN[] of N = 32768 ...       Reorder the sequence ...        Reorder the sequence ...        Run time = 40 ms
Running for N = 65536 (k = 16)...
Calculating WN[] of N = 65536 ...       Reorder the sequence ...        Reorder the sequence ...        Run time = 80 ms
Running for N = 131072 (k = 17)...
Calculating WN[] of N = 131072 ...      Reorder the sequence ...        Reorder the sequence ...        Run time = 170 ms
Running for N = 262144 (k = 18)...
Calculating WN[] of N = 262144 ...      Reorder the sequence ...        Reorder the sequence ...        Run time = 360 ms
Running for N = 524288 (k = 19)...
Calculating WN[] of N = 524288 ...      Reorder the sequence ...        Reorder the sequence ...        Run time = 750 ms
Running for N = 1048576 (k = 20)...
Calculating WN[] of N = 1048576 ...     Reorder the sequence ...        Reorder the sequence ...        Run time = 1700 ms
Running for N = 2097152 (k = 21)...
Calculating WN[] of N = 2097152 ...     Reorder the sequence ...        Reorder the sequence ...        Run time = 3570 ms
Running for N = 4194304 (k = 22)...
Calculating WN[] of N = 4194304 ...     Reorder the sequence ...        Reorder the sequence ...        Run time = 8050 ms
Running for N = 8388608 (k = 23)...
Calculating WN[] of N = 8388608 ...     Reorder the sequence ...        Reorder the sequence ...        Run time = 16850 ms
Running for N = 16777216 (k = 24)...
Calculating WN[] of N = 16777216 ...    Reorder the sequence ...        Reorder the sequence ...        Run time = 35170 ms
Running for N = 33554432 (k = 25)...
Calculating WN[] of N = 33554432 ...    Reorder the sequence ...        Reorder the sequence ...        Run time = 73320 ms
Running for N = 67108864 (k = 26)...
./serial.sh: line 8: 17768 Killed               $EXECUTABLE $k 0
Running for N = 134217728 (k = 27)...
```

Recursive code

Both the iterative code and the recursive code have similar serial performance , The recursive code for k= 26 and above gets killed because of memory overload in the stack. But the iterative code runs till k=30 to overload the memory. Trying to parallelize the recursive doesn't help much this is because recursive stack calls leads to a lot of overhead memory that does optimize the performance of the most time consuming functions but and leads to a lot of communication between recursive calls and does not help much in performance.

Parallelizing Iterative FFT

we get the following profiling results :

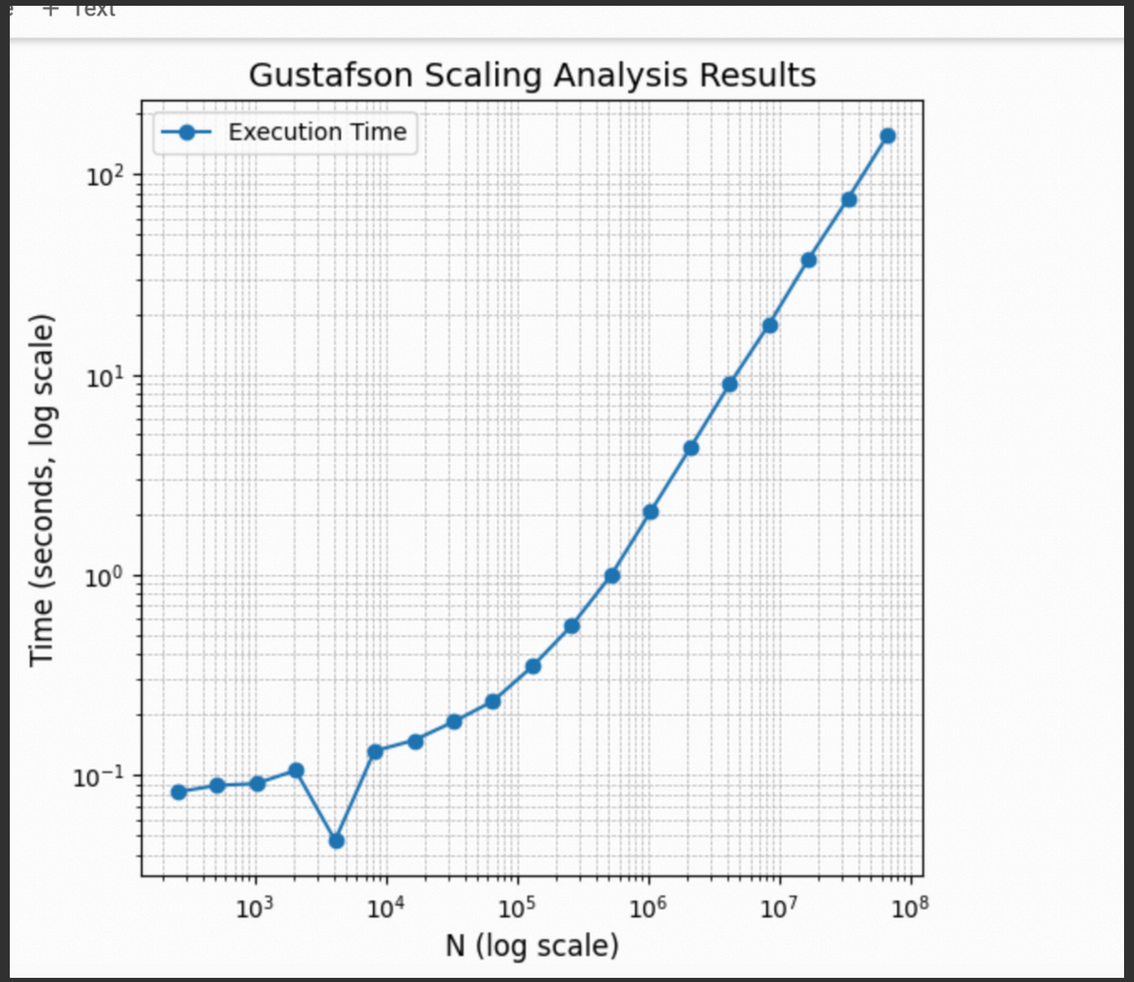
```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 69.51      4.25     4.25        1     4.25     5.13  DIT_FFT(Complex*, int, Complex*, int)
 14.39      5.13     0.88  4194301     0.00     0.00  append_seq(Complex*, Complex*, int)
 13.25      5.95     0.81        2     0.41     0.41  reorder_seq(Complex*, int)
  0.65      5.99     0.04                             save_to_file(char const*, Complex const*, int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&)
  0.65      6.03     0.04                             DIT_FFT_reordered(Complex*, int)
  0.65      6.07     0.04                             DIT_IFFT_reordered(Complex*, int)
  0.49      6.10     0.03        1     0.03     0.03  Calc_Inverse_WN(int)
  0.49      6.13     0.03        1     0.03     0.03  Calc_WN(int)
  0.00      6.13     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z10ComplexMul7ComplexS_
```

The function DIT_FFT which calculates the Fast Fourier Transform is the most expensive part of the code , we use OPENMP to parallelize the code by using threads within the code.

```bash
#!/bin/bash
#SBATCH --job-name=fft_scaling               # Job name
#SBATCH --output=fft_scaling_results_%j.out  # Output file for job results, %j is replaced with job ID
#SBATCH --error=fft_scaling_results_%j.err   # Error file for job errors
#SBATCH --ntasks=1                           # Single task (using OpenMP, not MPI)
#SBATCH --cpus-per-task=16

#SBATCH --time=01:00:00                      # Maximum job runtime

# Define the output results file
results_file="fft_scaling_results_$SLURM_JOB_ID.txt"

echo "Threads Runtime" > $results_file

threads_list=(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16)

for threads in "${threads_list[@]}"; do
    if [ $threads -le $SLURM_CPUS_PER_TASK ]; then
        export OMP_NUM_THREADS=$threads
        echo "Running with $threads thread(s)..."

        # Measure the runtime using `time` command or internal timing in code
        start_time=$(date +%s.%N)
        ./fft 22 4
        end_time=$(date +%s.%N)


        runtime=$(echo "$end_time - $start_time" | bc -l)

        # Save the results to the file
        echo "$threads $runtime" >> $results_file
    else
        echo "Skipping $threads threads as it exceeds allocated CPUs ($SLURM_CPUS_PER_TASK)"
    fi
done

echo "Scaling analysis completed. Results saved to $results_file."
```

Gustafson’s Law Analysis

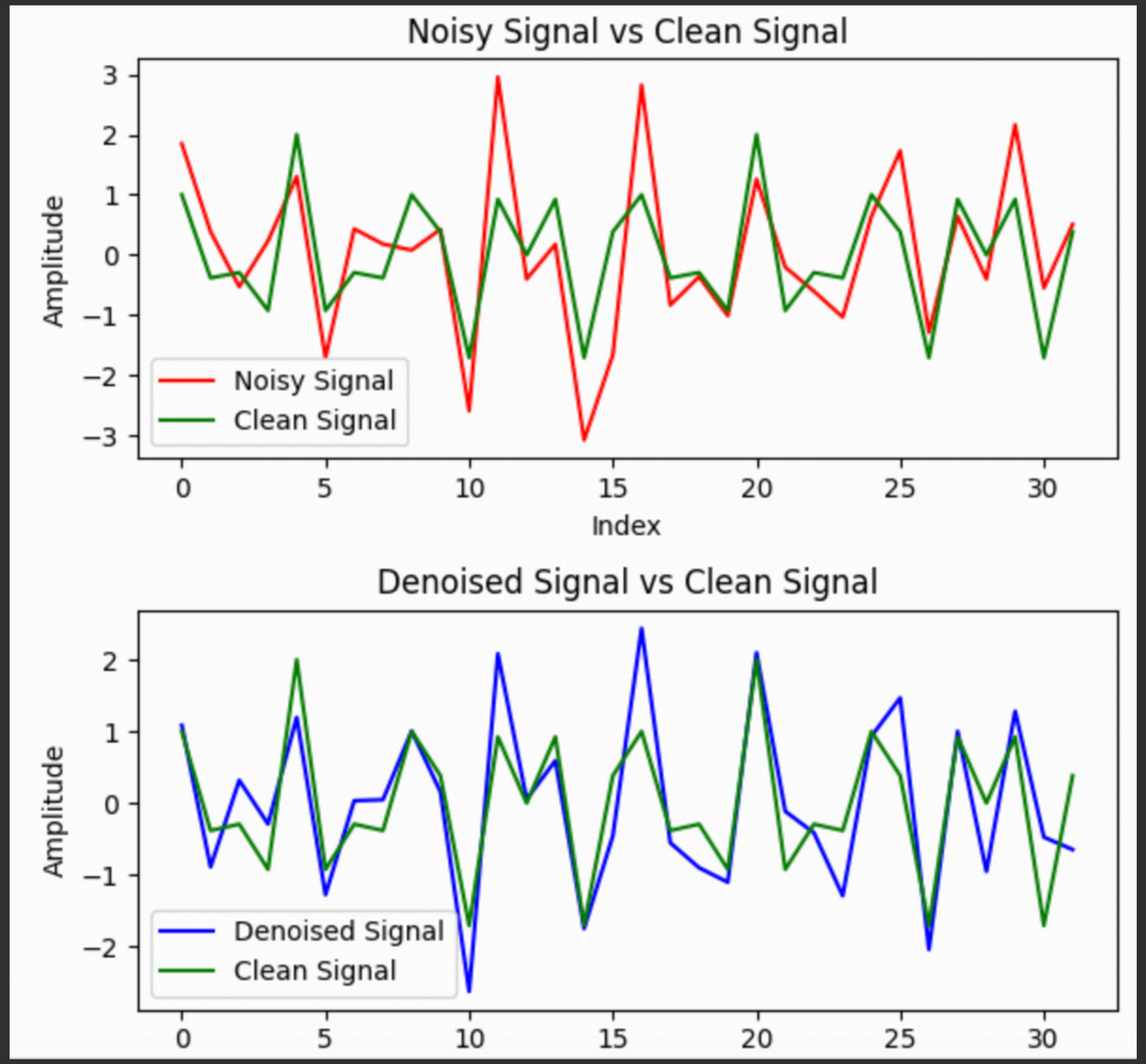
We run the code for varying input singal values and as the number of inputs increase we also increase the number of threads . Theoretically we should observe a straight line on the time vs number of inputs axis. Since the load per thread should remain constant.



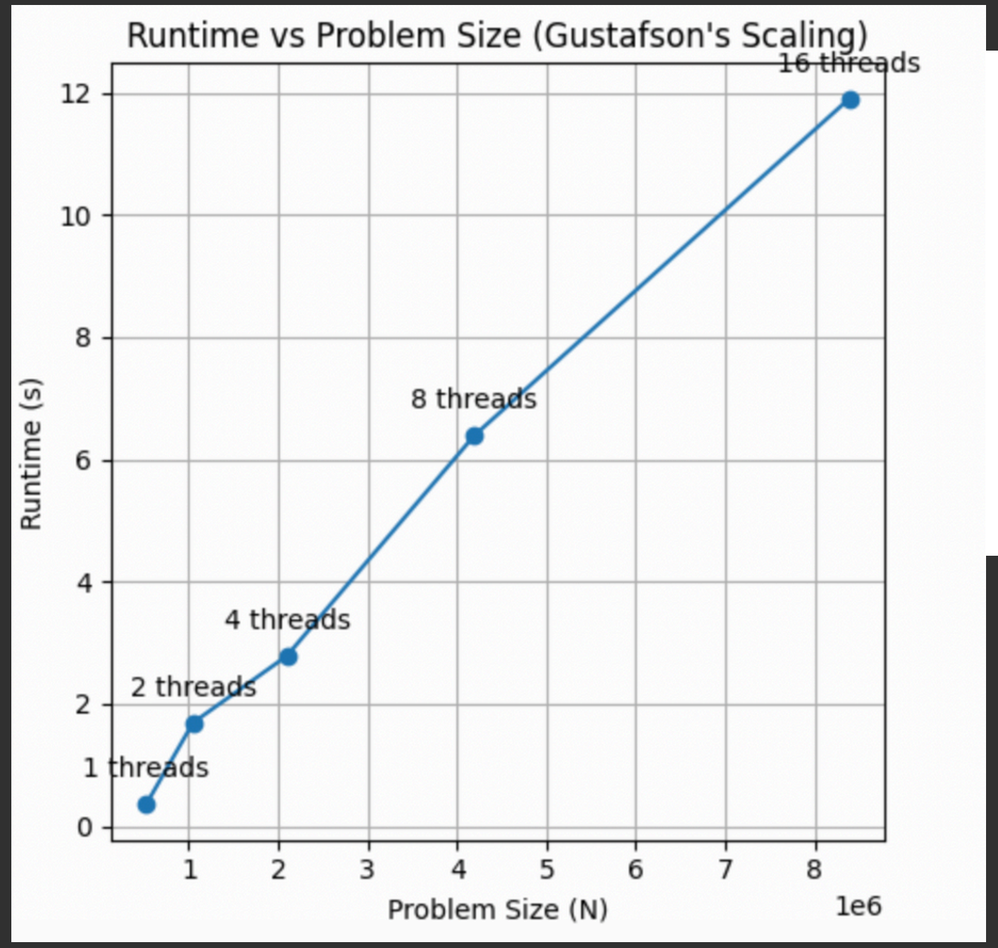
The code parallelises to give us an approximate straight line graph on Gustafson's scaling indicating that parallelisation aids in speeding up the code and also allows us to calculate the FFT and the inverse FFT. The timing output was taken both running both FFT and inverse FFT together.

Denoising Signals

The FFT algorithm has a great application in De-noising signals. Basically when we perform FFT we convert the signal in its frequency. If we analyse the signal in its frequency domain and use Power spectrum analysis to pick out the frequencies that have power density higher than a threshold. we then perform an Inverse fourier transform to get the De-noised signal back


Noisy Signal vs Clean Signal
Amplitude
Index
Noisy Signal
Clean Signal
Denoised Signal vs Clean Signal
Amplitude
Denoised Signal
Clean Signal

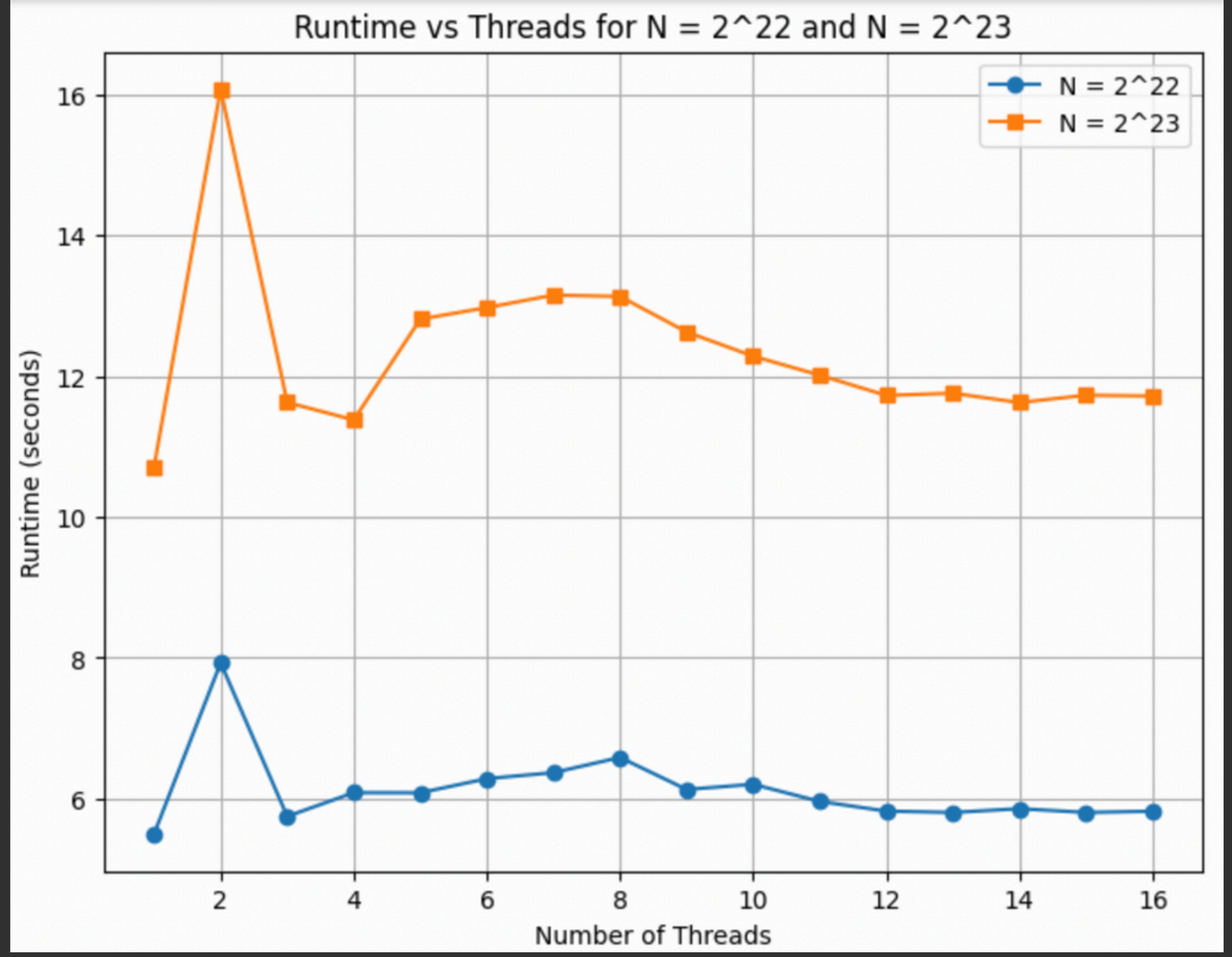
Scaling Analysis on De-noising Signal



N (2^k)	N	Threads	Runtime (s)
2^19	524288	1	0.354229
2^20	1048576	2	1.69135
2^21	2097152	4	2.78374
2^22	4194304	8	6.39024
2^23	8388608	16	11.902

Same result as before because we perform both FFT and inverse FFT

This is Amdhal's Scaling for N= 2^22 on the de noised signal

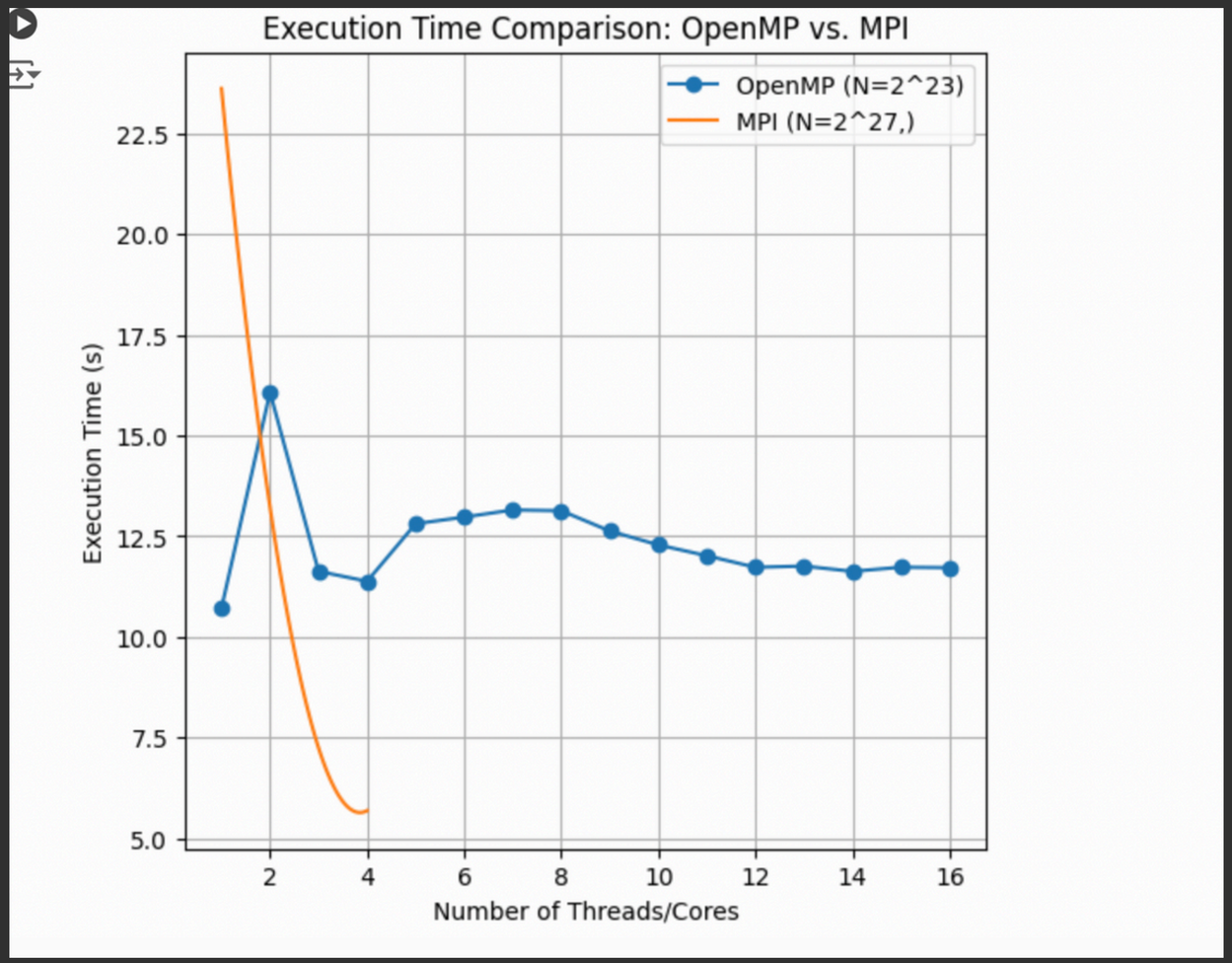


The graph suggests that as we keep the problem size constant(N= 2^23 and 2^22) and increase the number of threads the run time is more or less constant . The graph suggests that adding threads to the code does not increase its efficiency. The reason behind this is the fact that FFT is a memory bound algorithm rather than a compute bound algorithm so just increasing the number of does not increase the efficiency of the code because the code requires a lot data transfer from the RAM to the Cache which leads to higher Parallel Overhead.

MPI Parallelization of FFT

More advance computations of FFT such as compressing images require the use of 2-D FFT and in spectral analysis of real world signals the number of discrete samples taken can exceed 2^27 in those cases we start to exceed the memory of a single core on usual cluster machines and require the need of distributed parallelisation. We also use the hybrid parallelization approach because just using openmp to parallelise our code fails in increasing the performance of our code.

We run the Code for N= 2^27 which is About 138 GB of data. The Data is generated by sampling points from a periodic function. We then run a scaling analysis on the data and find its FFT and Inverse FFT for increasing number of cores and compare the results to N=2^23 scaling analysis of FFT using Threads.



What we find is that the execution tome of the code with Threads Is more or less constant for a constant problem size , but when we use the Hybrid approach of Using MPI+OPENMP to parallelise our code we get a great reduction in execution time . The Hybrid code also works significantly better with larger data sets unlike the code Parallelised using just OPENMP.

Conclusions

Memory-Bound Execution:

- Increasing threads doesn't help if memory bandwidth is maxed out. The FFT's large data set leads to frequent cache misses and heavy memory traffic, creating a performance bottleneck that more threads can't overcome.

Limited Memory Bandwidth:

- More threads generate more simultaneous memory requests. The available DRAM bandwidth saturates quickly, preventing runtime improvement as thread count grows.

Poor Data Locality & Cache Behavior:

- The FFT's "butterfly" pattern involves non-contiguous memory accesses, causing limited cache reuse and increased data movement.
- Synchronization Overhead:
- Each FFT stage requires barriers and scheduling decisions. Increasing threads adds synchronization costs, reducing efficiency gains.

Comparison with MPI:

- MPI processes run on separate nodes or NUMA regions, each with dedicated memory bandwidth, allowing better scaling than a single shared-memory system with the same total number of cores.

Further Applications of FFT

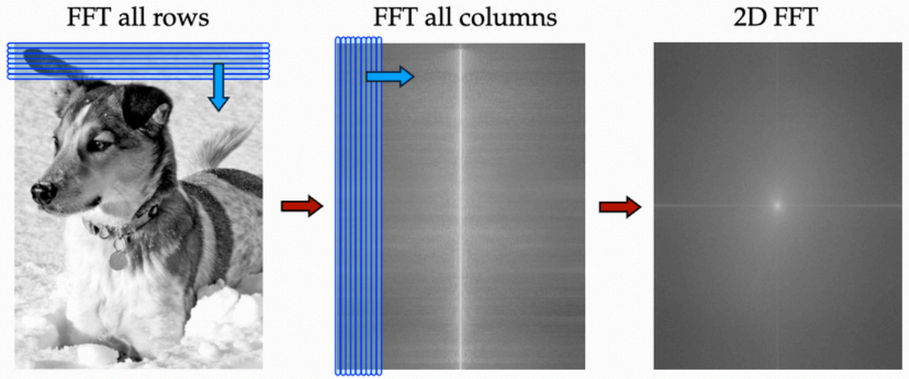


Figure 2.25: Schematic of 2D FFT. First, the FFT is taken of each row, and then the FFT is taken of each column of the resulting transformed matrix.

Compressing Images and Videos

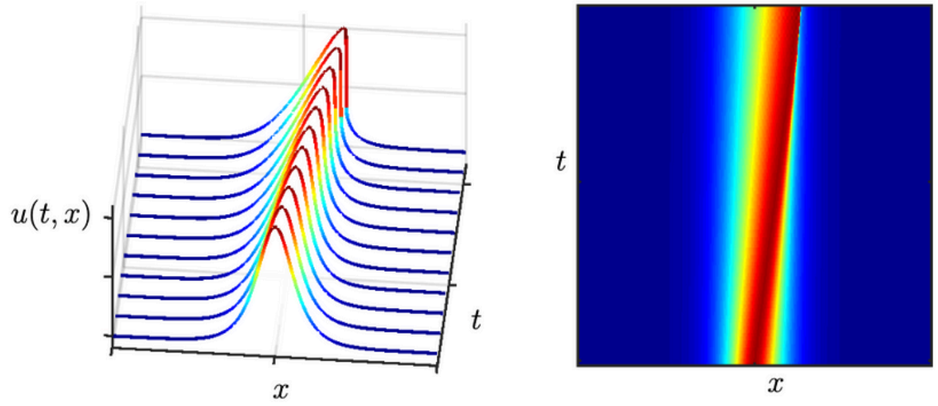


Figure 2.18: Evolution of Burgers' equation in time, illustrated by a waterfall plot (left) and an x-t diagram (right).

Solving Quasi-Linear PDEs

Derivatives of functions. The Fourier transform of the derivative of a function is given by:

$$\mathcal{F}\left(\frac{d}{dx}f(x)\right) = \int_{-\infty}^{\infty} \overbrace{f'(x)}^{dv}\overbrace{e^{-i\omega x}}^{u}\, dx \tag{2.19a}$$

$$= \Big[\underbrace{f(x)e^{-i\omega x}}_{uv}\Big]_{-\infty}^{\infty} - \int_{-\infty}^{\infty} \underbrace{f(x)}_{v}\Big[\underbrace{-i\omega e^{-i\omega x}}_{du}\Big]\, dx \tag{2.19b}$$

$$= i\omega \int_{-\infty}^{\infty} f(x)e^{-i\omega x}\, dx \tag{2.19c}$$

$$= i\omega\mathcal{F}(f(x)). \tag{2.19d}$$

This is an extremely important property of the Fourier transform, as it will allow us to turn PDEs into ODEs, closely related to the separation of variables:

Thank you !