Fast Fourier Transform (FFT)

Dylan Zinsley

What is a (Fourier) transform?

- “relate a function in one domain to another function in a second domain”
- (Fourier) takes a function and outputs the frequencies present in the original function

$$e^{ix} = \cos x + i \sin x,$$

Fourier transform

$$\widehat{f}(\xi) = \int_{-\infty}^{\infty} f(x)\, e^{-i2\pi\xi x}\, dx, \quad \forall \xi \in \mathbb{R}. \qquad \textbf{(Eq.1)}$$

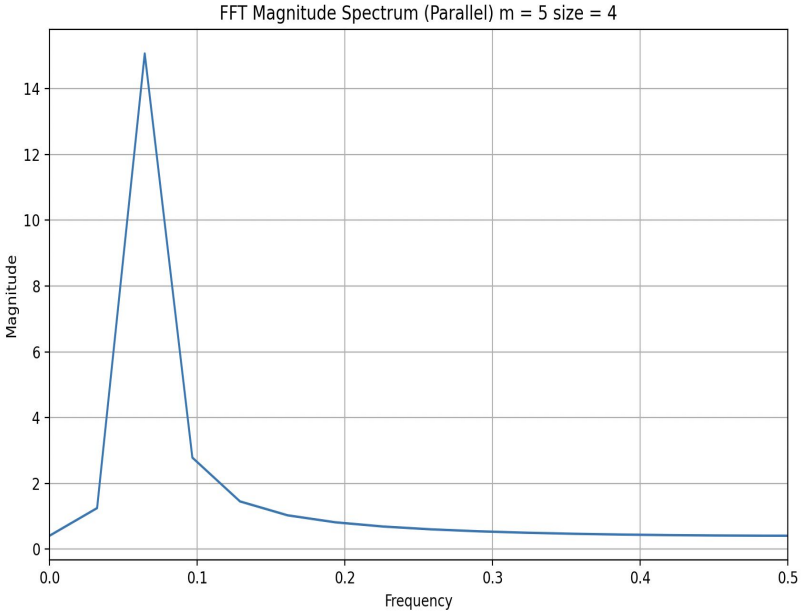
Applications

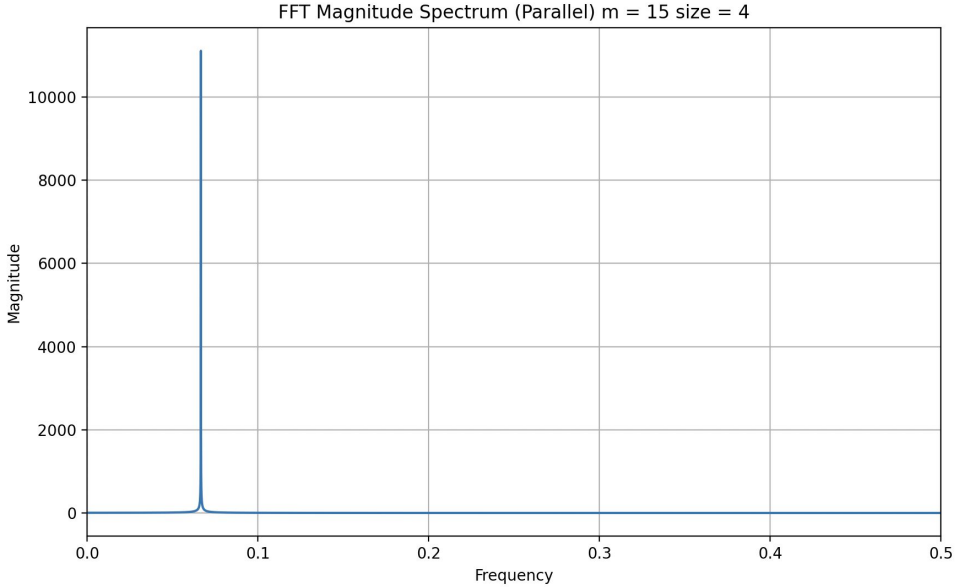
- Signal processing, image processing
  - Filtering, analysis, etc.
- Fourier transform was used for thermodynamics
  - FFT (algorithm) was used to detect nuclear explosions
- Machine learning and matrix operations
- Many more :)

Discrete fourier transform

- What if we don’t have this function, but can only sample magnitudes at different times?
- We can!
  - And this can be represented by a matrix

More runtime == better accuracy





Math break!

$$\omega_N = e^{-i2\pi/N}$$

$$w_{2n}^2 = w_n$$

DFT matrix

$$\mathbf{F} = \begin{bmatrix} \omega_N^{0\cdot 0} & \omega_N^{0\cdot 1} & \cdots & \omega_N^{0\cdot (N-1)} \\ \omega_N^{1\cdot 0} & \omega_N^{1\cdot 1} & \cdots & \omega_N^{1\cdot (N-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_N^{(N-1)\cdot 0} & \omega_N^{(N-1)\cdot 1} & \cdots & \omega_N^{(N-1)\cdot (N-1)} \end{bmatrix}$$

Fast Fourier Transform (FFT)

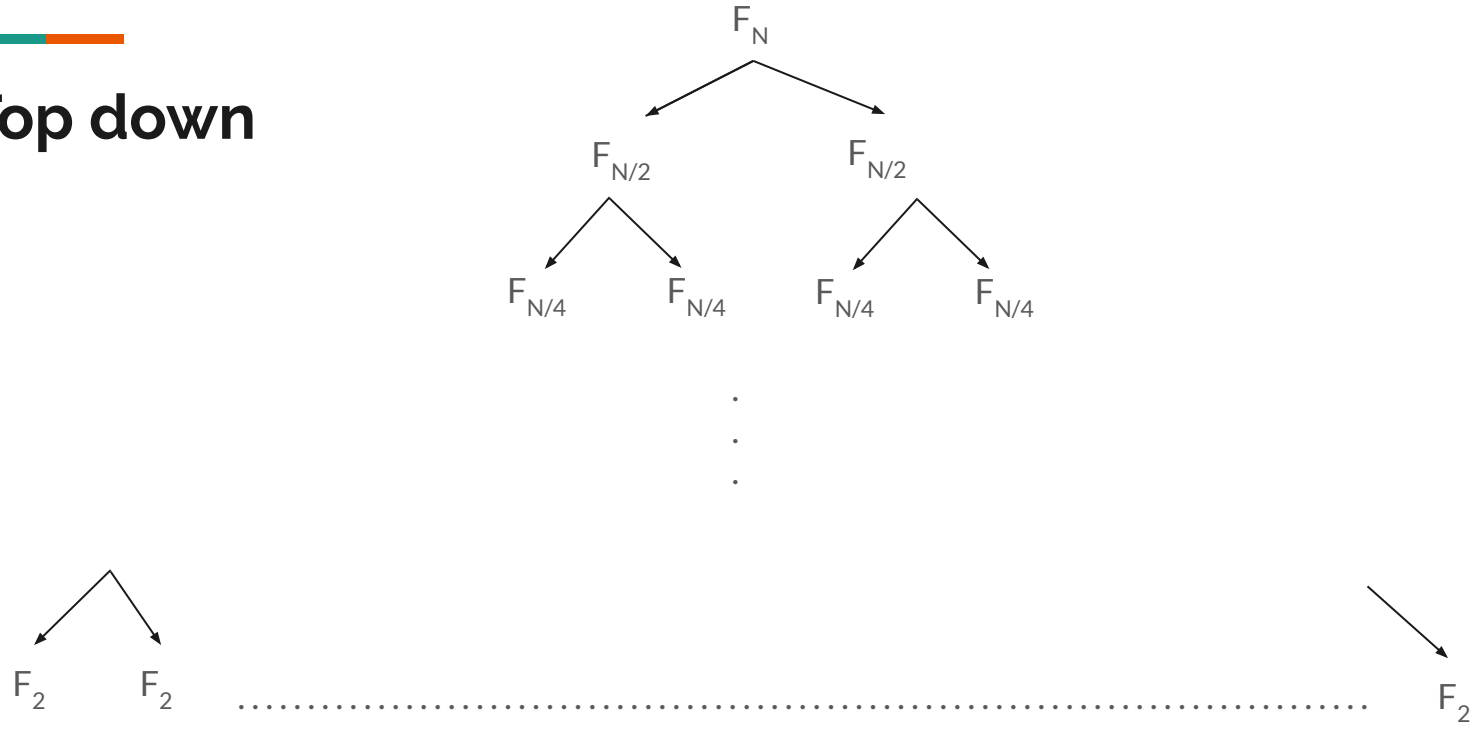
- An algorithm for computing the DFT
- Based on matrix decomposition
- O(N^2) => O(Nlog(N)

Sparse product matrix

$$D = \begin{bmatrix} 1 & & & & \\ & w & & & \\ & & w^2 & & \\ & & & \ddots & \\ & & & & w^{n-1} \end{bmatrix}.$$

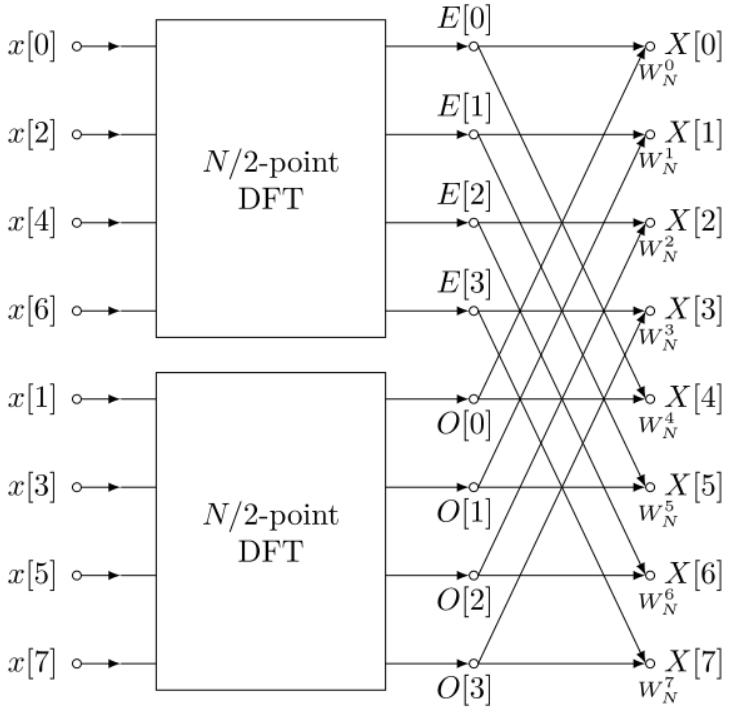
$$F_{2n} = \begin{bmatrix} I & D \\ I & -D \end{bmatrix} \begin{bmatrix} F_n & 0 \\ 0 & F_n \end{bmatrix} P,$$

Top down



Butterfly pattern

$$\omega_N^{(N/2+k)} = -\omega_N^{k}$$



Input

Butterfly walkthrough

x(0) = 0

x(1) = 1

x(2) = 2

x(3) = 3

x(4) = 4

x(5) = 5

x(6) = 6

x(7) = 7

Input

x(0) = 0

x(1) = 1

x(2) = 2

x(3) = 3

x(4) = 4

x(5) = 5

x(6) = 6

x(7) = 7

Input

x(0) = 0

x(1) = 1

x(2) = 2

x(3) = 3

x(4) = 4

x(5) = 5

x(6) = 6

x(7) = 7

Bit reversal

x(0) => 000

x(1) => 001

x(2) => 010

x(3) => 011

x(4) => 100

x(5) => 101

x(6) => 110

x(7) => 111

Input

Bit reversal

x(0) = 0	x(0) => 000 => 000
x(1) = 1	x(1) => 001 => 100
x(2) = 2	x(2) => 010 => 010
x(3) = 3	x(3) => 011 => 110
x(4) = 4	x(4) => 100 => 001
x(5) = 5	x(5) => 101 => 101
x(6) = 6	x(6) => 110 => 011
x(7) = 7	x(7) => 111 => 111

Input

Bit reversal

x(0) = 0	x(0) => 000 => 000 => 0
x(1) = 1	x(1) => 001 => 100 => 4
x(2) = 2	x(2) => 010 => 010 => 2
x(3) = 3	x(3) => 011 => 110 => 6
x(4) = 4	x(4) => 100 => 001 => 1
x(5) = 5	x(5) => 101 => 101 => 5
x(6) = 6	x(6) => 110 => 011 => 3
x(7) = 7	x(7) => 111 => 111 => 7

Input

x(0) = 0

x(4) = 4

x(2) = 2

x(6) = 6

x(1) = 1

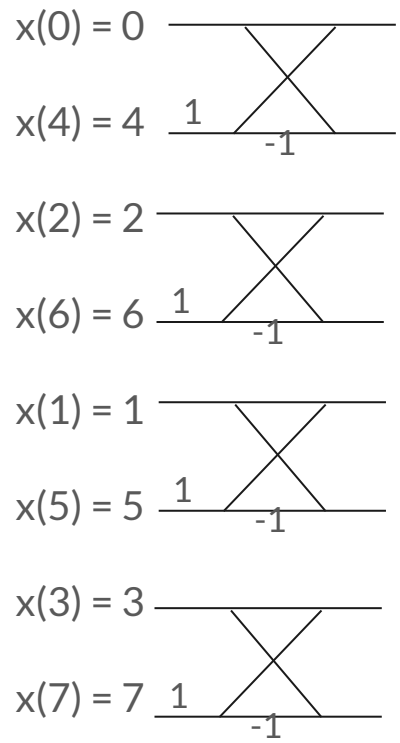
x(5) = 5

x(3) = 3

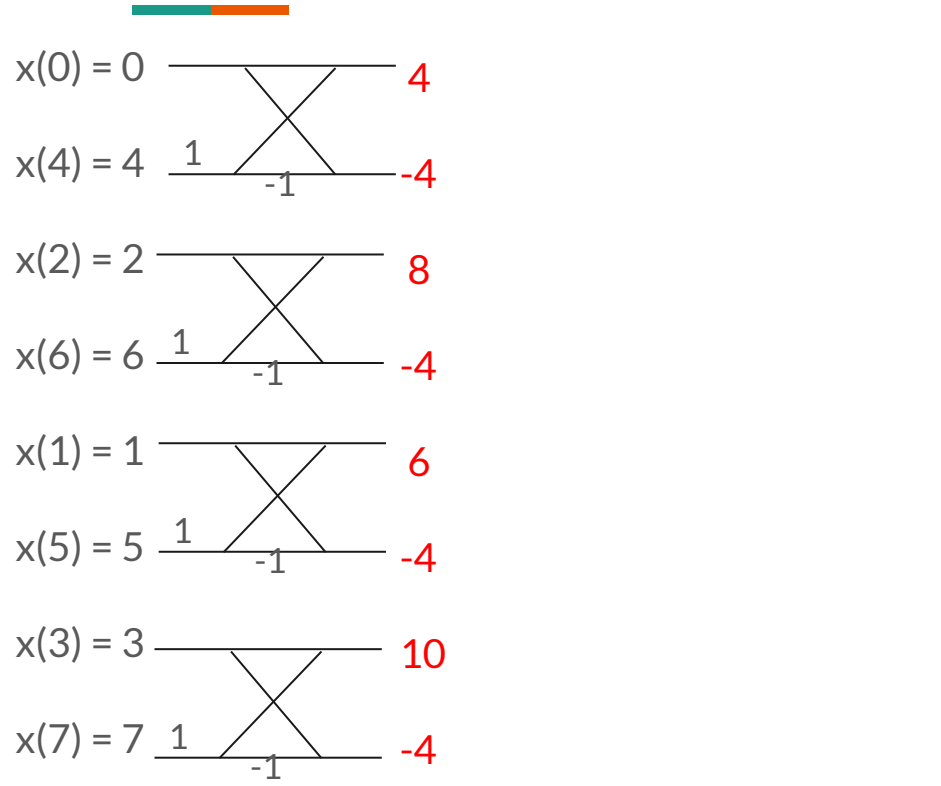
x(7) = 7

Input

$\omega_2^0 = 0$



Input $\omega_2^0=0$



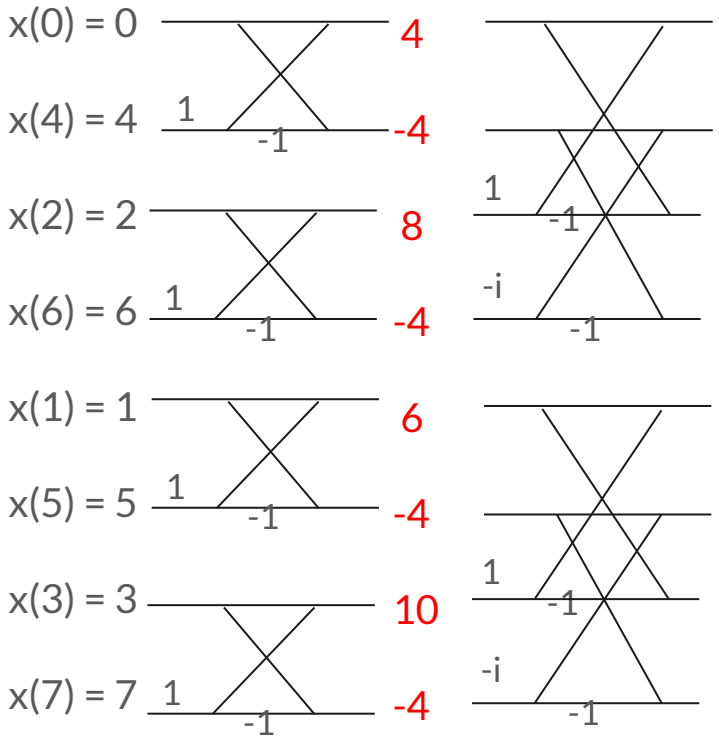
Stage 1

Stage 2

Input

$\omega_2^0=0$

$\omega_4^0=0,$
$\omega_4^1=-i$



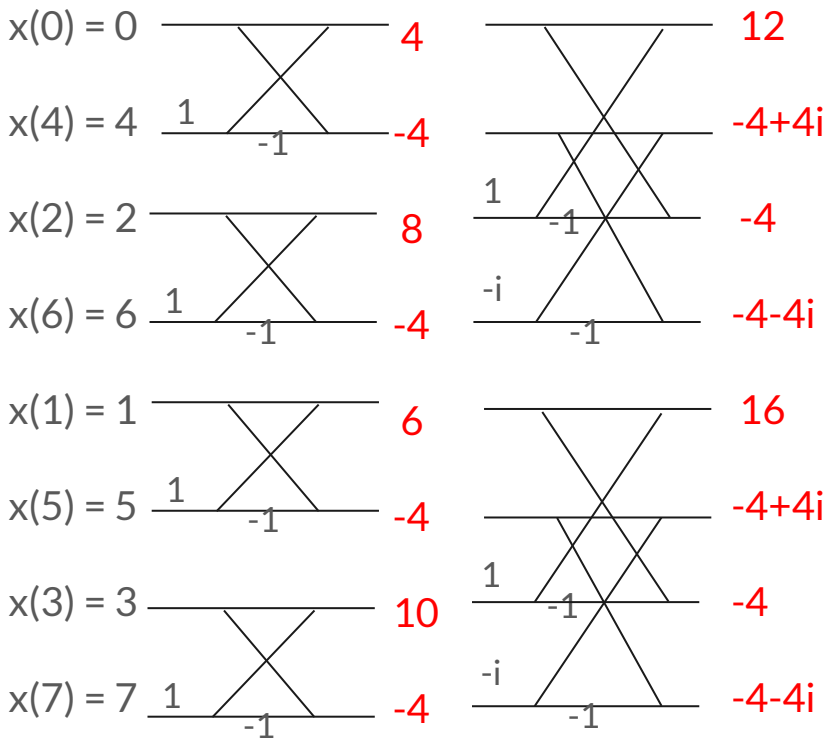
Stage 1

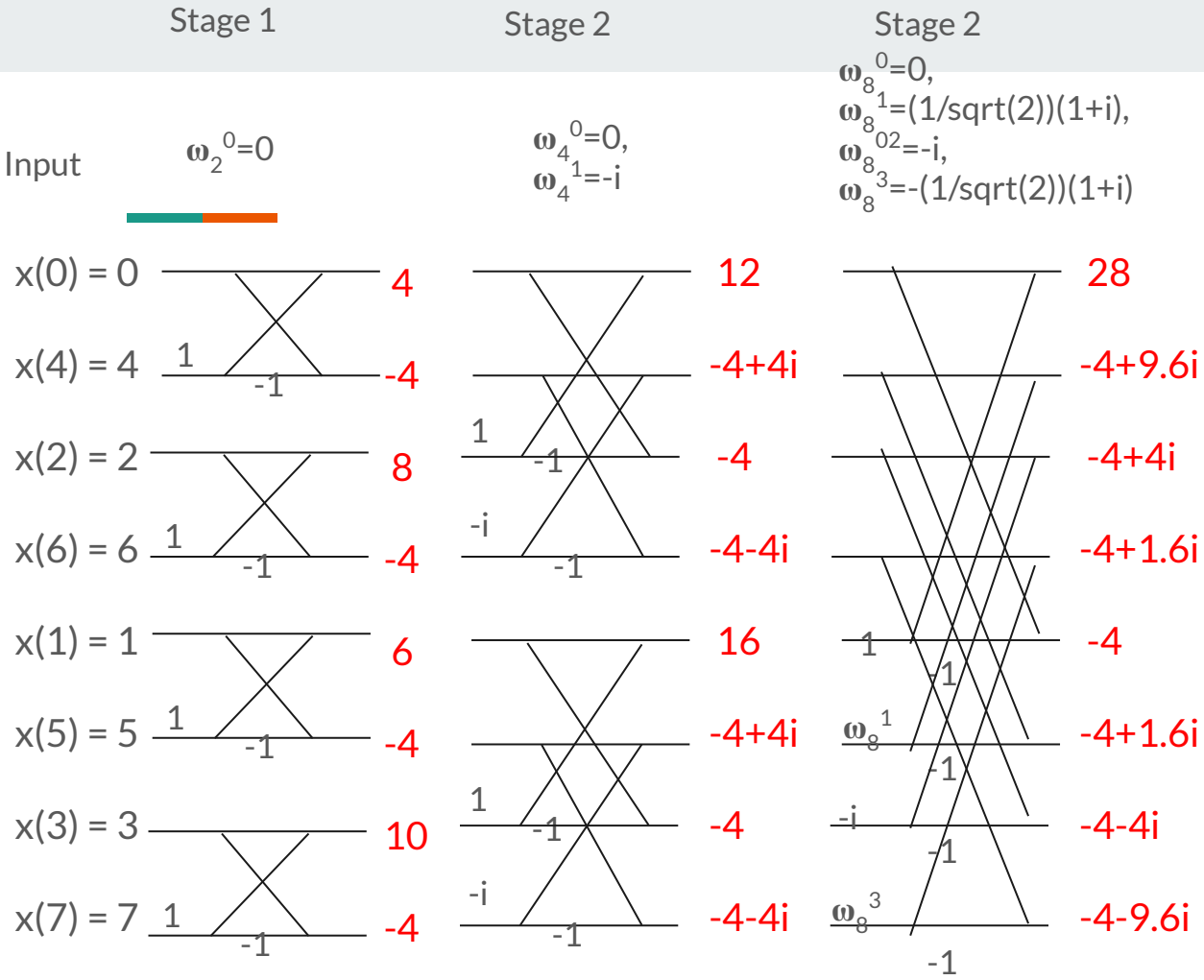
Stage 2

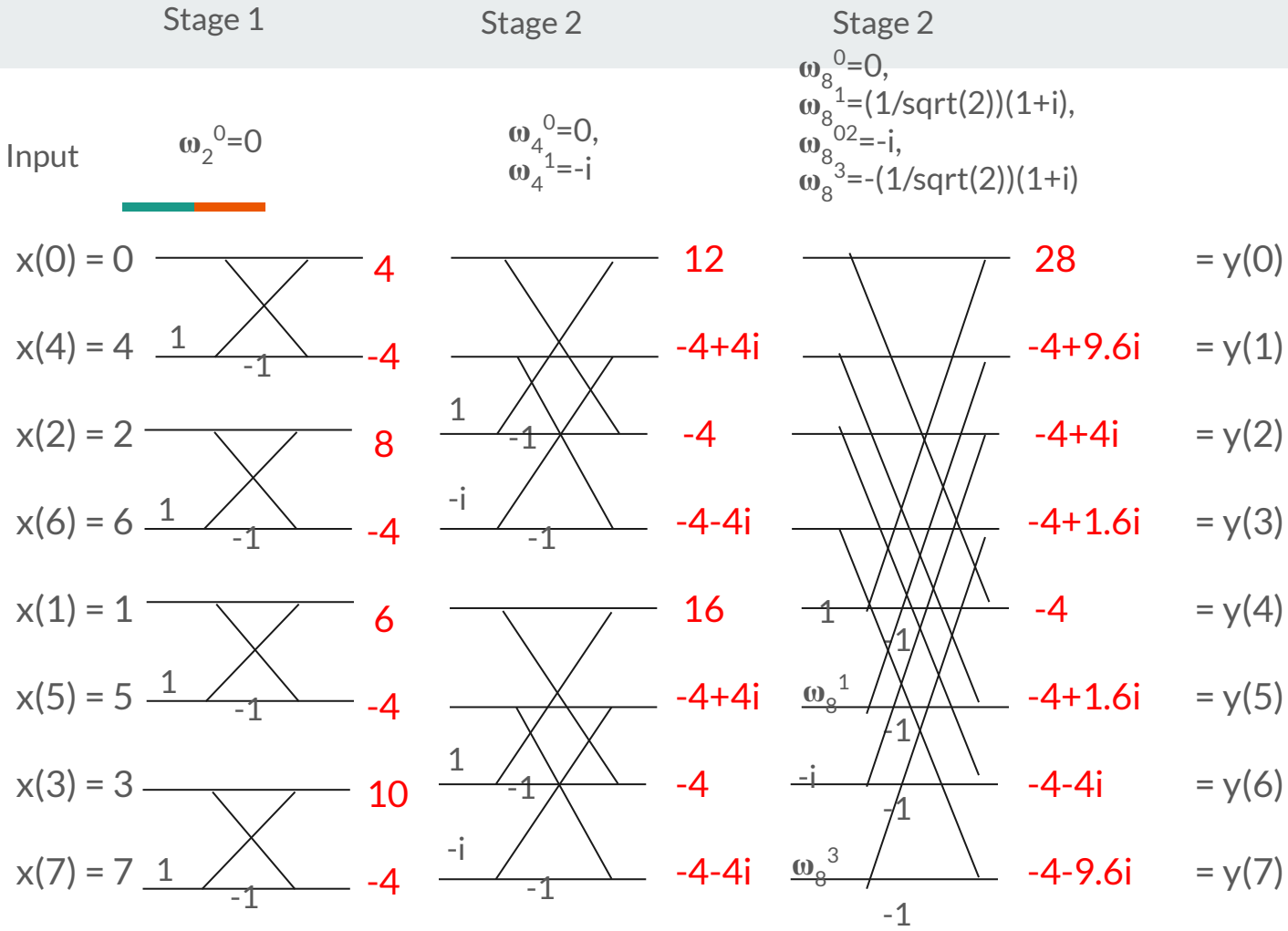
Input

$\omega_2^0=0$

$\omega_4^0=0,$
$\omega_4^1=-i$




Stage 1
Stage 2
Stage 2
Input
$\omega_2^0=0$
$\omega_4^0=0$,
$\omega_4^1=-i$
$\omega_8^0=0$,
$\omega_8^1=(1/\text{sqrt}(2))(1+i)$,
$\omega_8^{02}=-i$,
$\omega_8^3=-(1/\text{sqrt}(2))(1+i)$
x(0) = 0
x(4) = 4
x(2) = 2
x(6) = 6
x(1) = 1
x(5) = 5
x(3) = 3
x(7) = 7
1
-1
4
-4
8
-4
6
-4
10
-4
12
-4+4i
-4
-4-4i
16
-4+4i
-4
-4-4i
28
-4+9.6i
-4+4i
-4+1.6i
-4
-4+1.6i
-4-4i
-4-9.6i
ω_8^1
ω_8^3
-i


Stage 1
Stage 2
Stage 2
Input
$\omega_2^0=0$
$\omega_4^0=0$,
$\omega_4^1=-i$
$\omega_8^0=0$,
$\omega_8^1=(1/\text{sqrt}(2))(1+i)$,
$\omega_8^{02}=-i$,
$\omega_8^3=-(1/\text{sqrt}(2))(1+i)$
x(0) = 0
x(4) = 4
x(2) = 2
x(6) = 6
x(1) = 1
x(5) = 5
x(3) = 3
x(7) = 7
1
-1
4
-4
8
-4
6
-4
10
-4
12
-4+4i
-4
-4-4i
16
-4+4i
-4
-4-4i
28
-4+9.6i
-4+4i
-4+1.6i
-4
-4+1.6i
-4-4i
-4-9.6i
ω_8^1
ω_8^3
-i
= y(0)
= y(1)
= y(2)
= y(3)
= y(4)
= y(5)
= y(6)
= y(7)

Compute local FFT, if needed

Partner node should always be 2^(stage+1) away

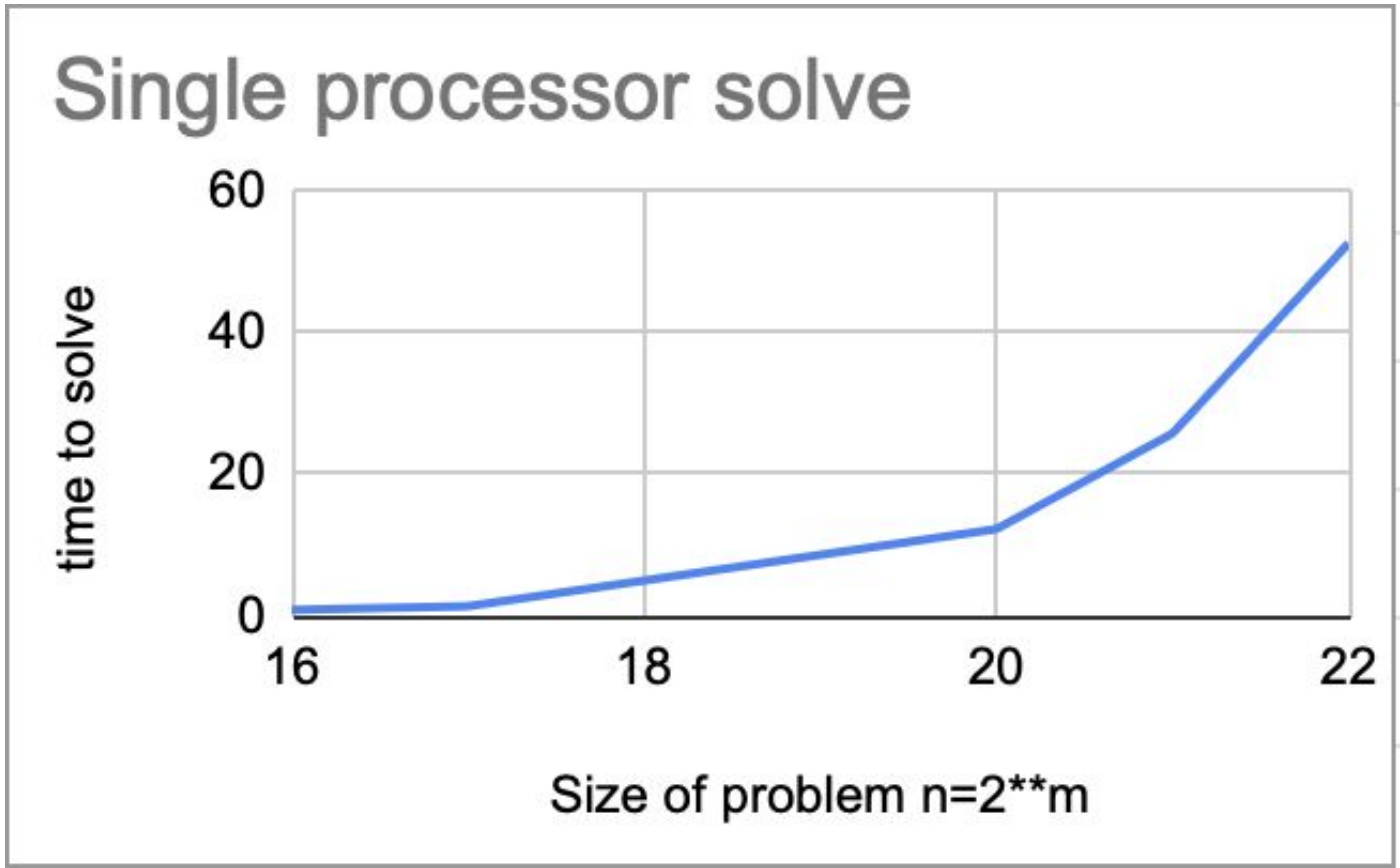
Extra offset calculation for if nodes hold >1 piece of data

Root of unity (twiddle factor)

Properly applying TF (and negation)

```python
def parallel_fft(x, comm):
    rank = comm.Get_rank()
    size = comm.Get_size()
    N = len(x) * size

    if not (size & (size - 1) == 0):
        raise ValueError("Number of processes must be power of 2")

    local_data = bit_reverse_list(x)

    local_data = local_fft(local_data)

    for stage in range(int(math.log2(size))):
        partner = rank ^ (1 << stage)

        send_data = local_data.copy()
        received_data = comm.sendrecv(send_data, dest=partner, source=partner)

        stage_size = len(local_data)
        for k in range(stage_size):
            tn = 2**(stage+1)*stage_size
            tk = (rank*stage_size+k) % (tn/2)
            tf = cmath.exp(-2j * cmath.pi * tk / tn)

            if rank & (1 << stage) == 0:
                local_data[k] = send_data[k] + tf * received_data[k]
            else: #sub bc lower
                local_data[k] = received_data[k] - tf * send_data[k]


    all_results = comm.gather(local_data, root=0)
```

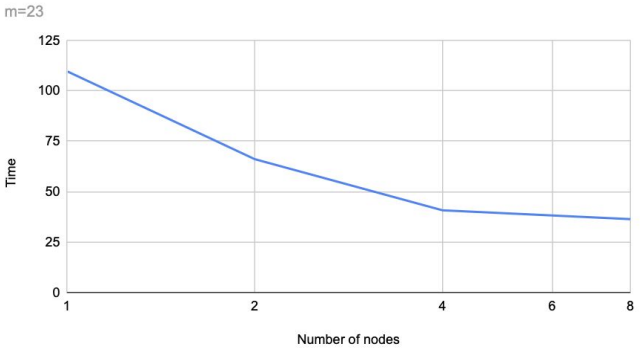
Results!


Single processor solve
60
40
20
0
time to solve
16
18
20
22
Size of problem n=2**m

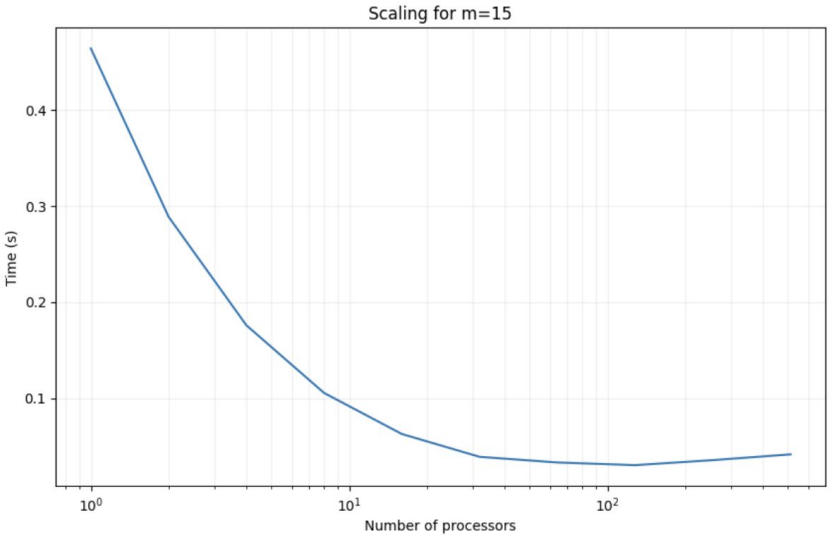
Strong scaling

~1.7x speedup

Nodes vs time

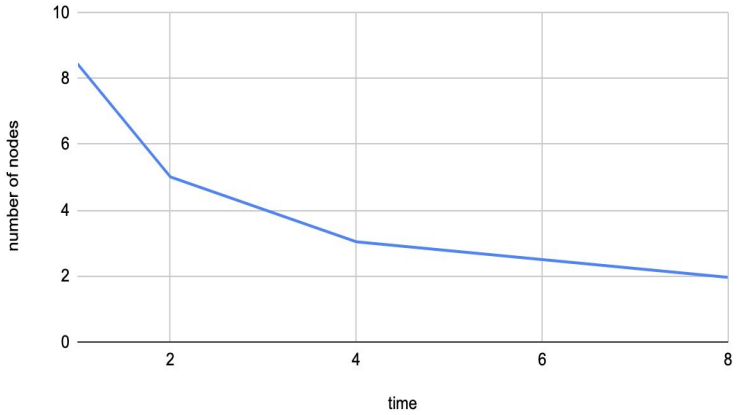


Nodes vs time



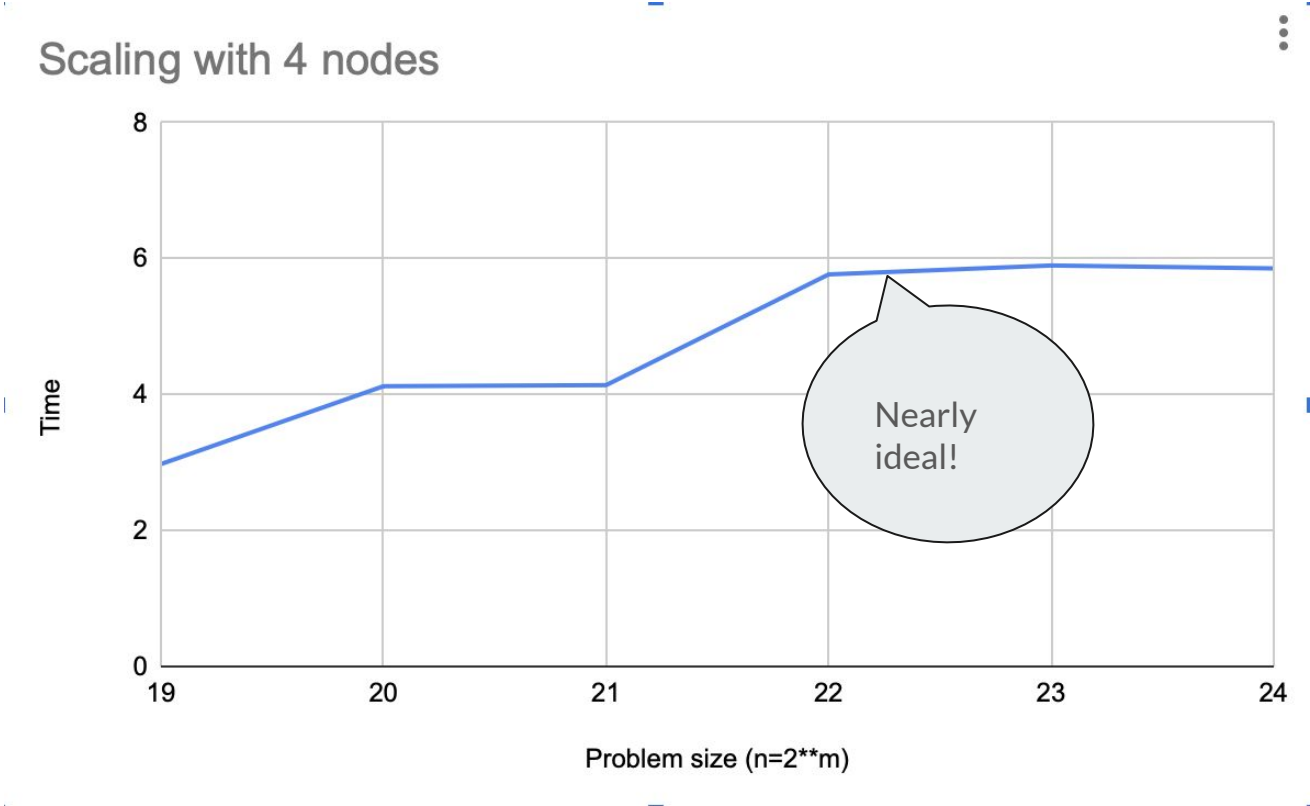
~1.7x speedup

Nodes vs time



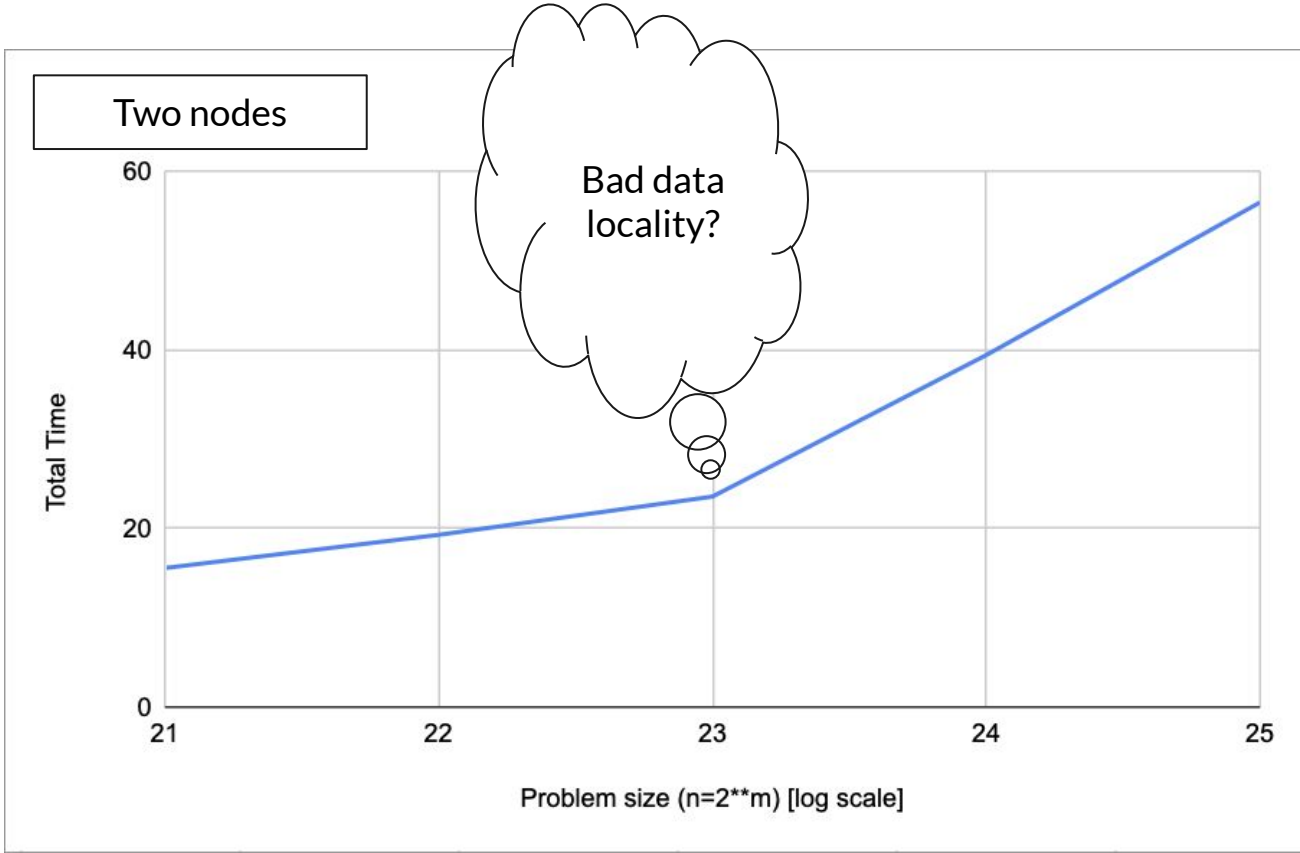
Weak Scaling:

32x problem size only loses a factor of 2!



Weak scaling:

Note: the dense FFT matrix for m=25 would have 1 quadrilion entries!!!



Future work (if i had had more time)

- Cuda :(
  - (i will do this someday)

Thank you