LARGE INTEGER MULTIPLICATION WITH FAST FOURIER TRANSFORM & SCHÖNHAGE-STRASSEN

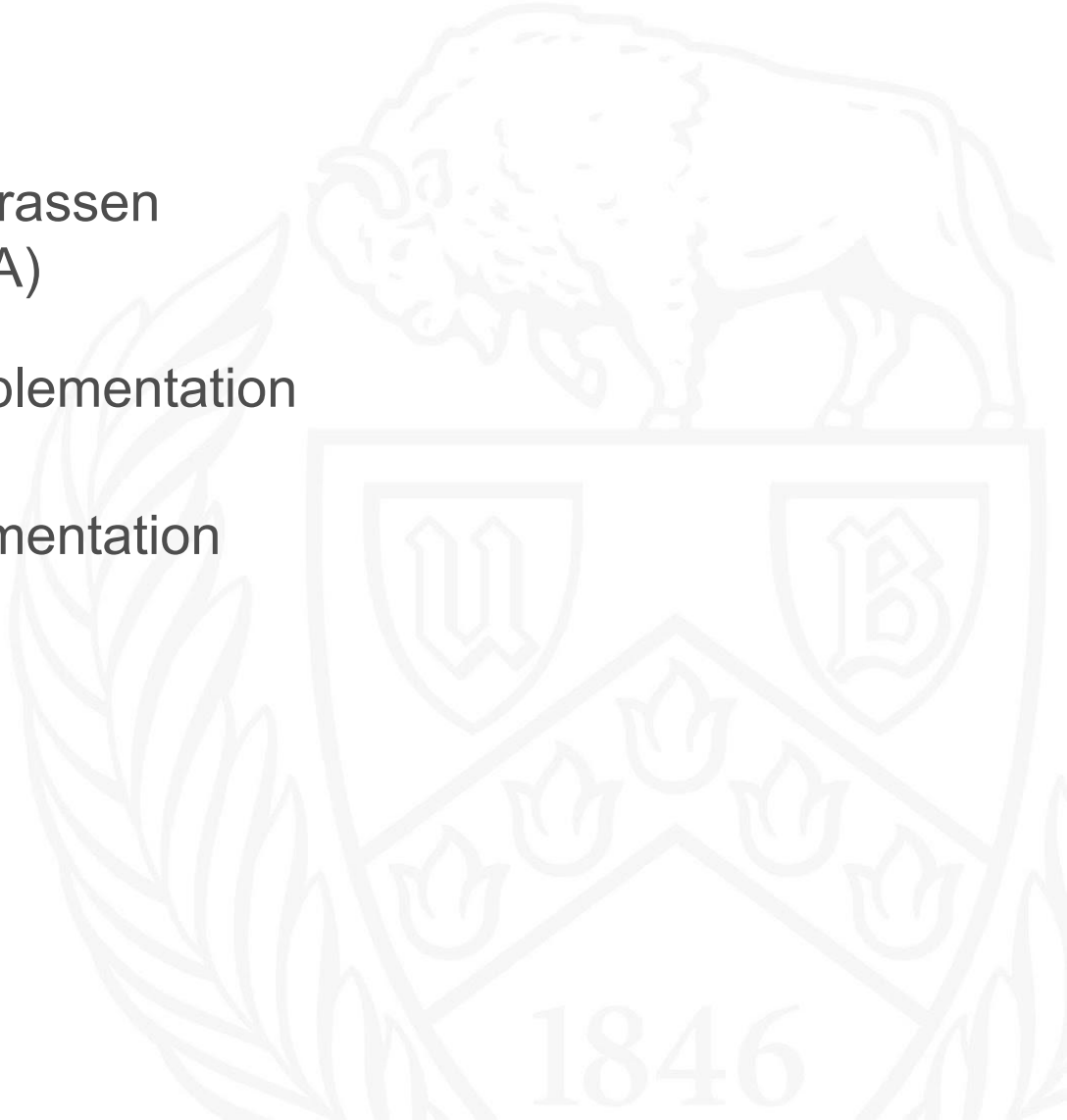
Peyton Joyce

CSE 633 · Parallel Algorithms

University at Buffalo The State University of New York

Overview

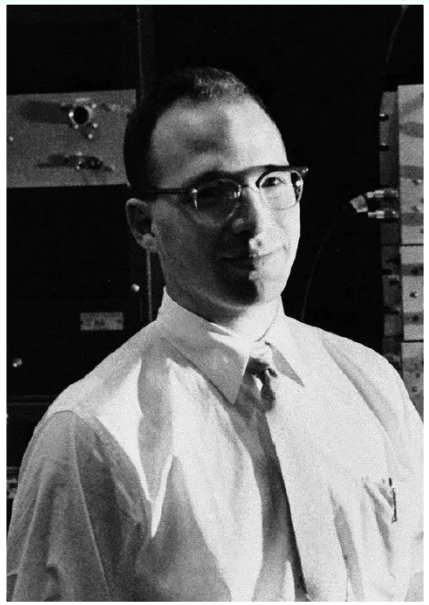
1. History & Applications
   - A. *Fourier Transform*
   - B. *Schönhage & Strassen*
2. The Four Fouriers
   - A. *Fourier Transform*
   - B. *Discrete Fourier Transform*
   - C. *Inverse Discrete Fourier Transform*
   - D. *Fast Fourier Transform (FFT)*
3. Integer Multiplication with FFT
4. Schönhage-Strassen Algorithm (SSA)
5. Sequential Implementation
6. Parallel Implementation
7. Results
8. Future Work

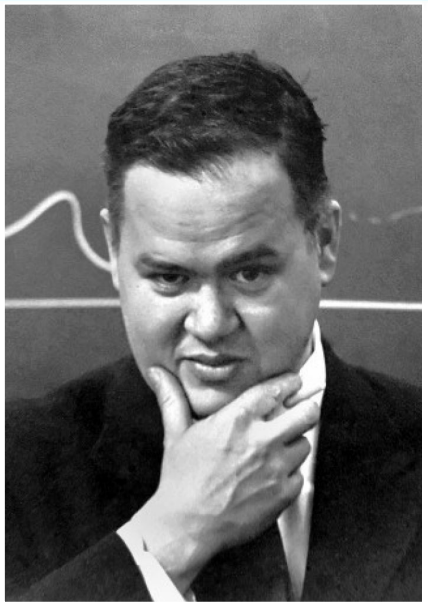


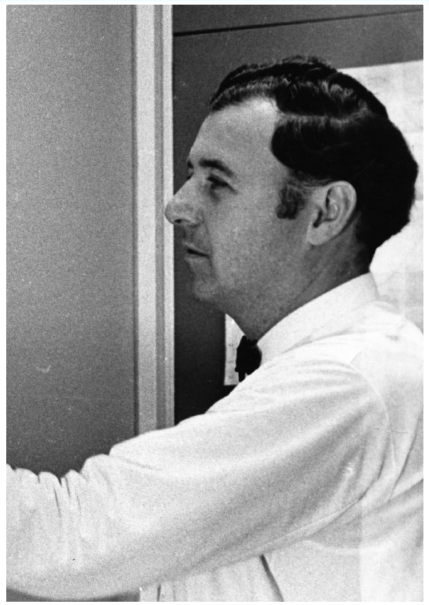
Joseph Fourier

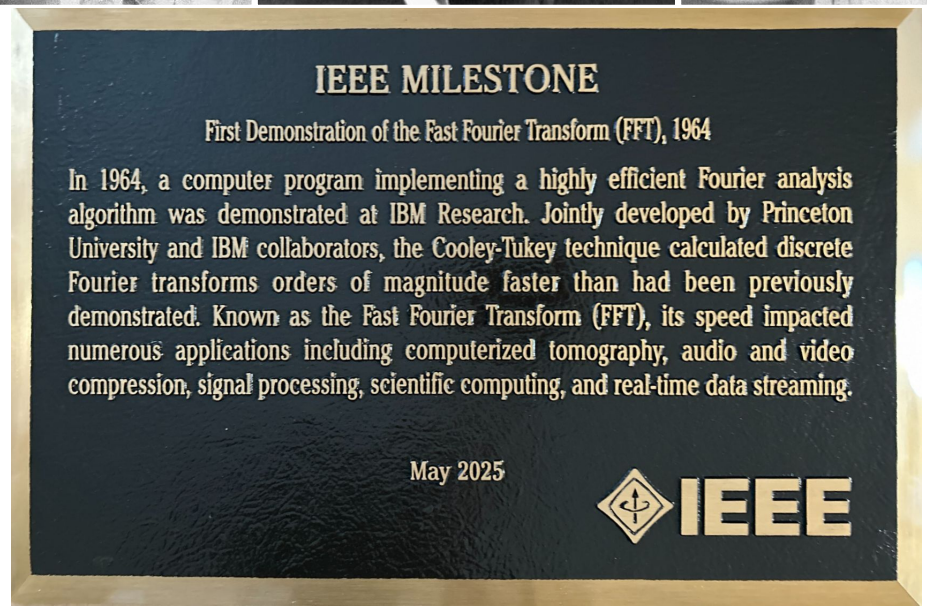
- French mathematician, physicist, governor, Egyptologist
- Scientific advisor to Napoleon
- Contributions:
  - Fourier series
  - Law of heat conduction
  - Heat equation
  - Greenhouse effect
- Namesakes:
  - Fourier method for separation of variables
  - Fourier-transform spectroscopy
  - **Fourier transform**
- Buried in Père Lachaise Cemetery

[1]









Garwin, Tukey, Cooley, and the Cold War

- **Dr. Richard Garwin** (IBM)
  - First hydrogen bomb design, first spy satellites, foundations of MRI, catalyst of FFT
- **Prof. John Tukey** (Princeton)
  - FFT, box plot, the words “bit” and “software”, pioneer of data science
- **Dr. James Cooley** (IBM)
  - FFT, early quantum computing, programming
- Devised to reliably detect underground nuclear testing by planting sensors and analyzing explosive vibrations

Originally discovered by Gauss in 1805!

Schönhage & Strassen

- **Dr. Arnold Schönhage**
  - German mathematician & computer scientist
  - SSA, multitape Turing machine
- **Dr. Volker Strassen**
  - German mathematician & statistician
  - Probability, complexity bounds, matrix multiplication
- **1971:** Schönhage & Strassen co-author asymptotically-fastest-known algorithm for large integer multiplication
  - Unbeaten until **2007**, and still widely used
  - GMP, NTL, MapReduce, OpenSSL & BIGNUM
  - Often slower than FFT in practice, but more accurate at large values

1979: *Strassen (left) and Schönhage (right) play a game of chess.*

Modern Applications

FAST FOURIER TRANSFORM

- Signal processing
- Digital recording
- Spectral analysis
- Audio/speech processing
- Image processing
- Communications
- Data compression/convolution
- MRI, CT, EEG, ECG
- 5G, Bluetooth, broadband internet

It's everywhere!

LARGE INTEGER MULTIPLICATION

- Cryptography
  - RSA primes
  - Diffie–Hellman key exchange
  - Elliptic-curve cryptography
  - Fully Homomorphic Encryption (FHE)
  - Zero-knowledge proofs
- High-precision arithmetic
  - Computational physics, engineering
  - Big data analysis

Fourier Transform

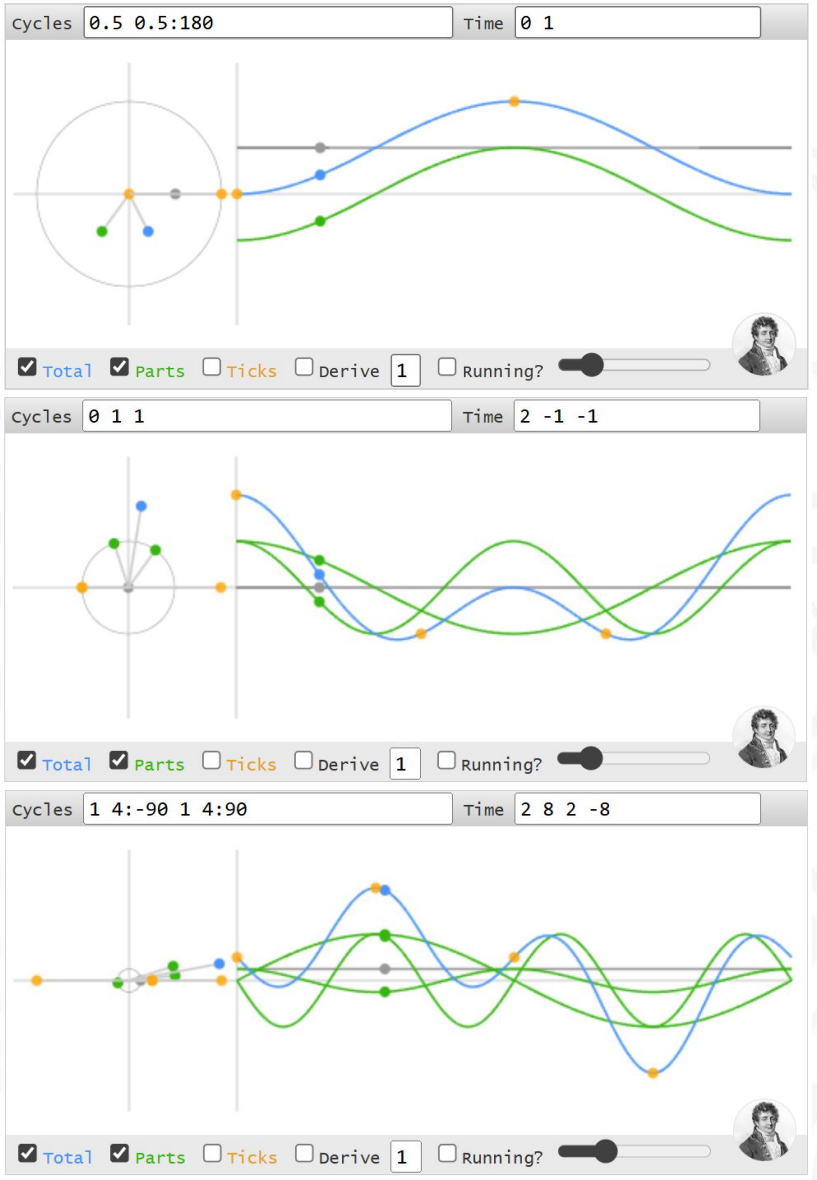
An integral transform that maps a function f(**x**) from one domain to another (**ξ**):

$$\widehat{f}(\xi) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i \xi x} dx, \ \forall \xi \in R$$

Often, we map from time (**t**) to frequency (*f*):

$$F(\omega) = \widehat{f}\left(\frac{\omega}{2\pi}\right) = \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt$$

[5][6][7][8]



Discrete Fourier Transform (DFT)

Fourier transform applied to discrete waves (data points at specific instances in time):

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-2\pi i \frac{k}{N} n}$$

- **k ∈ [N]** frequency bins, evaluated at **n** samples
- More applicable to real-world data
- **O(N²)** runtime

SOLVING THE DFT

1. Convert the exponential term using Euler's formula:

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

2. Calculate the summation. Each term will be a complex number of form $\mathbf{X_k = a_k + b_k i}$
3. Plot $(\mathbf{a_k}, \mathbf{b_k})$ vectors on the complex plane. Calculate:
   a. *Amplitude* of k^{th} bin as the vector's magnitude (Pythagorean theorem)
   b. *Phase* of k^{th} bin as the vector's angle (arctan)
4. Apply final steps (Nyquist frequency rule, averaging by n, plotting & phase-shifting)
5. Et Voilà

Inverse Discrete Fourier Transform (IDFT)

Exactly the reverse of the DFT:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{2\pi i \frac{k}{N} n}$$

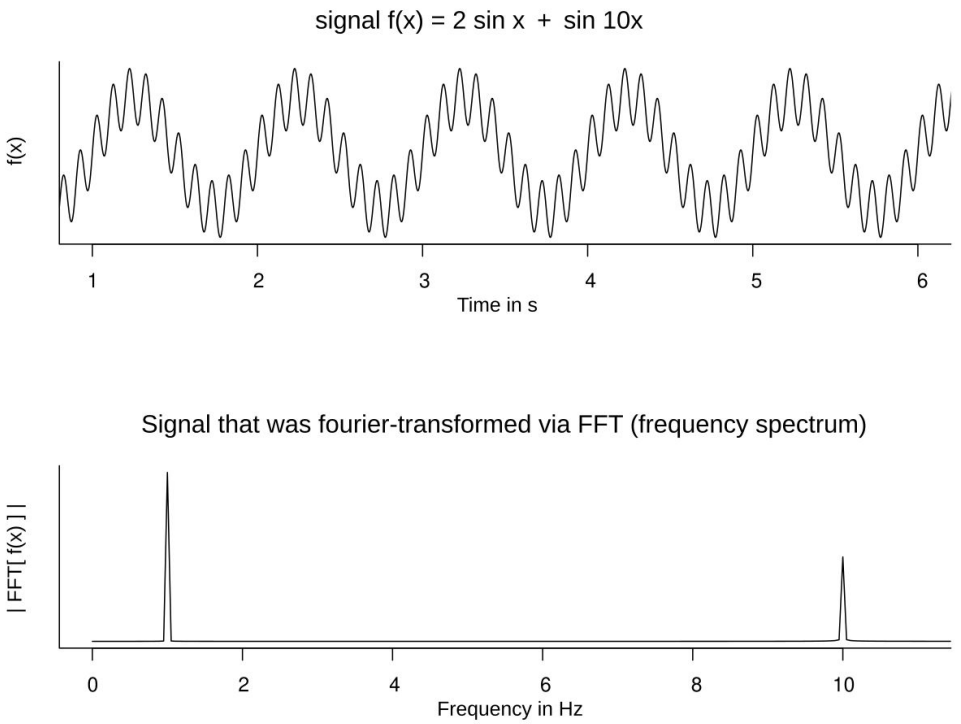
- Converts from frequency domain back to time
- Only need to flip the sign of the imaginary number and divide by N when complete

Fast Fourier Transform (FFT)

Any *algorithm* that solves the DFT in **O(n log n)** time.

COOLEY-TUKEY ALGORITHM (RADIX-2 DIT)

- **Split in half:** even-indexed & odd-indexed inputs
- Recursively solve on each half, then combine
- Use N as a power of 2 for easy dividing
- Use **roots of unity** to reduce computations



[3] [4] [11] [12]

Integer Multiplication with FFT

PROBLEM

- **Input:** Two large integers, **a** and **b**
- **Output:** The product **c = a × b**

SOLUTIONS

- **Naive Algorithm:** $O(n^2)$
- **DFT Algorithm:** $O(n^2)$
- **FFT Algorithm:** $O(n \log n)$

Generalizing to Polynomial Multiplication

- Any integer can be written as a polynomial
  - ex. 4321 ⇒ $4x^3 + 3x^2 + 2x + 1$, where x = 10
- Any polynomial can be written as a list of its coefficients ("coefficient form")
  - ex. $4x^3 + 3x^2 + 2x + 1$ ⇒ [1, 2, 3, 4],
    where the index of each term corresponds to its degree
- Any polynomial of degree **d** can be *uniquely represented* by **N ≥ d + 1** points ("value form")
  - Choose **N** to be a power of 2 for convenience
  - Pointwise multiplication of two polynomials in value form takes **linear time!**
  - Need a way to cut our calculations of **N** points in half at each step

Choosing N Points

- For any polynomial **P(x)** with only *even* terms, **P(−x) = P(x)**
- For any polynomial **P(x)** with only *odd* terms, **P(−x) = −P(x)**

If we split our polynomial into even ($\mathbf{P_e(x)}$) and odd ($\mathbf{P_o(x)}$) terms, and factor one **x** from $\mathbf{P_o(x)}$, we can use the same calculation to get two points from one **x** (**+/−** point pairs).

PROBLEM: In **P(x)**, we have positive *and* negative **x** terms. When these terms are squared in $\mathbf{P_e(x)}$ and $\mathbf{P_o(x)}$, they become exclusively positive, and we no longer have point pairs.

SOLUTION: Use **complex numbers** and **roots of unity**.

Roots of Unity

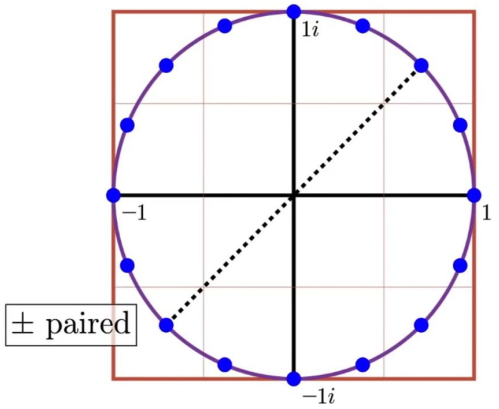
For a positive integer **n**, an **nth** root of unity is any number **z** such that **z^n = 1**.

If we include complex numbers, by Euler's formula, we have roots of unity ω:

$$\omega^k = e^{2\pi i \frac{k}{n}} = cos(\frac{2\pi k}{n}) + i\, sin(\frac{2\pi k}{n}) \; \forall \; k = 0, 1, \ldots, n-1$$

- For any ω^j, $\omega^{j + N/2} = -\omega^j$ is exactly 180° opposite on the unit circle (+/− point pairs)
- During recursion, ω^j and $\omega^{j + N/2}$ are squared, effectively becoming the same value and giving us the **N/2th** roots of unity, or **N/2** points (cut the problem in half)

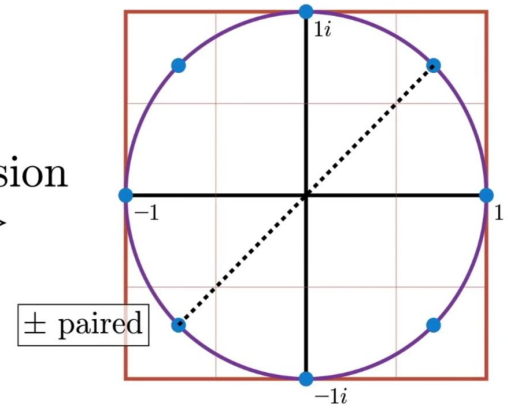
$\omega^{j+n/2} = -\omega^j \rightarrow (\omega^j, \omega^{j+n/2})$ are ± paired



Evaluate $P(x)$ at $[1, \omega^1, \omega^2, \ldots, \omega^{n-1}]$

n roots of unity

Recursion ⟹



Evaluate $P_e(x^2)$ and $P_o(x^2)$ at $[1, \omega^2, \omega^4, \ldots, \omega^{2(n/2-1)}]$

$(n/2)$ roots of unity

Recursive Pseudocode

```
func multiply_with_fft(a, b):
    convert a into a_coeff
    convert b into b_coeff
    // where N = 2^k for k ∈ Z:
    define N ≥ (len(a_coeff) + len(b_coeff) + 1)
    a_value = FFT(a_coeff, N)
    b_value = FFT(b_coeff, N)
    c_value = pointwise multiply a_value × b_value
    conjugate each complex number in c_value
    c_coeff = FFT(c_value, N)
    divide all values in c_coeff by N
    propagate carry over c_coeff
    convert c_coeff into c
    return c
```

```
func FFT(P, N):
    if N == 1:
        return P
    P_e = P[::2]   // even-degree indices
    P_o = P[1::2]  // odd-degree indices
    y_e = FFT(P_e, N/2)
    y_o = FFT(P_o, N/2)
    y = [0] × N
    for j in range(N/2):  // get +/− point pairs
        y[j] = y_e[j] + ω^j × y_o[j]
        y[j + n/2] = y_e[j] − ω^j × y_o[j]
    return y
```

[3][4][11][12]

Iterative Pseudocode

```
func multiply_with_fft(a, b, N):
    shuffle(a)  // Bit-reversed indices
    FFT(a, N)   // In-place butterfly
    shuffle(b)
    FFT(b, N)
    c = pointwise multiply a × b
    conjugate each complex number in c
    shuffle(c)
    FFT(c, N)
    divide all values in c by N
    round to int, propagate carry
    return c
```

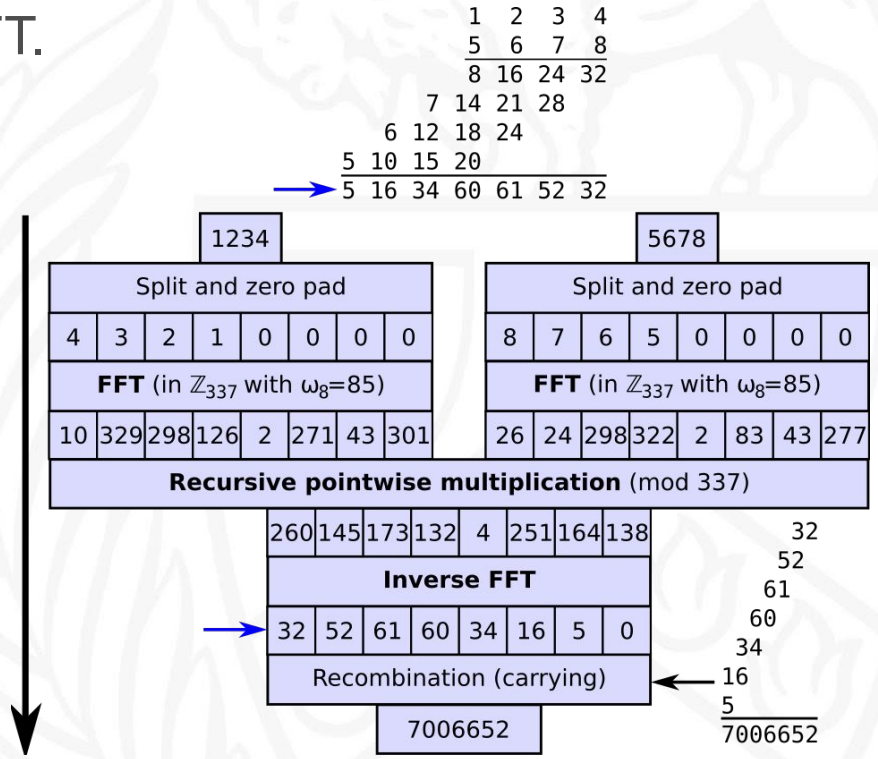
```
func FFT(P, N):
    for (size = 2; size ≤ N; size *= 2):
        calculate ω-base
        for (sub = 0; sub < N; sub += size):
            ω = 1.0 + 0.0i
            for (i = 0; i < (size/2); i++):
                y_e = P[i + sub]
                y_o = P[i + sub + (size/2)]

                P[i + sub]              = y_e + ω × y_o
                P[i + sub + (size/2)] = y_e − ω × y_o
                ω *= ω-base
```

Schönhage-Strassen & Modular Rings

SSA replaces **complex** roots of unity with **modular arithmetic** using a **Number Theoretic Transform** (NTT), a specialized DFT.

- **Modular Ring:** A finite set of integers {0, 1, …, n − 1} where addition & multiplication are performed modulo **n**
- **Modulo:** Define **M = 2^{2^k} + 1**, where **k** represents the "size" of numbers being handled at a level of recursion



Schönhage-Strassen & Modular Rings

SSA replaces **complex** roots of unity with **modular arithmetic** using a **Number Theoretic Transform** (NTT), a specialized DFT.

- **Root of Unity:** An $\mathbf{n}^{th}$ root of unity modulo **M** is an integer **ω** s.t. $\boldsymbol{\omega}^{\mathbf{n}} \equiv 1 \pmod{M}$, but $\boldsymbol{\omega}^{\mathbf{k}} \not\equiv 1$ for any smaller values of **k**
  - When we define $\mathbf{M = 2^{2\wedge k} + 1}$, powers of 2 act as roots of unity
  - ex. to get a 4^{th} RoU, **M** = 17. $4^2 = 16 \equiv \mathbf{-1} \pmod{17}$, and $4^4 = 256 \equiv \mathbf{1} \pmod{17}$

Runtime: $O(n \times \log n \times \log \log n)$

- Often slower than FFT
- No floating-point precision errors!

Illustrative Example

8

7

6

5

4

3

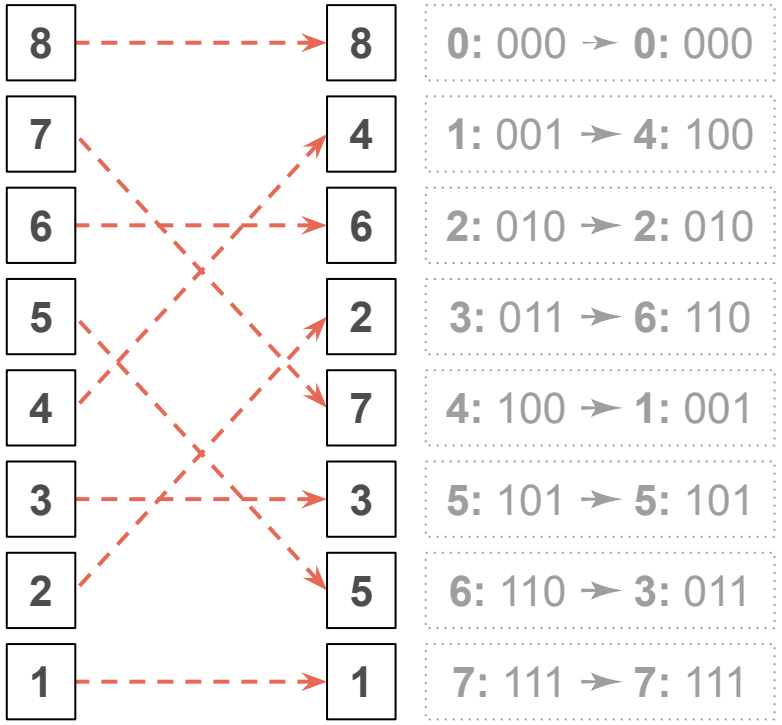
2

1

Let’s run FFT on the integer:

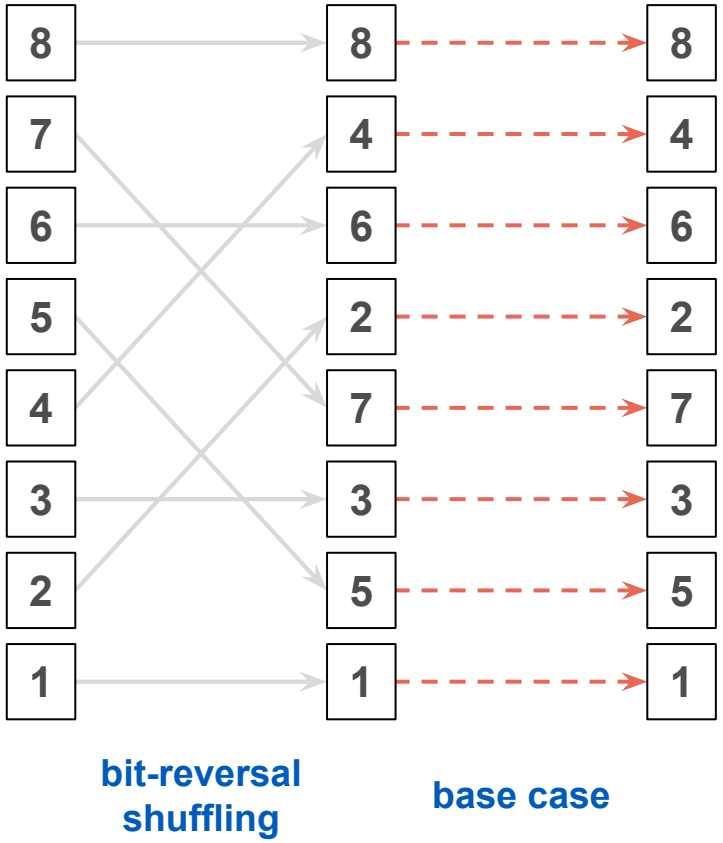
12345678

Illustrative Example



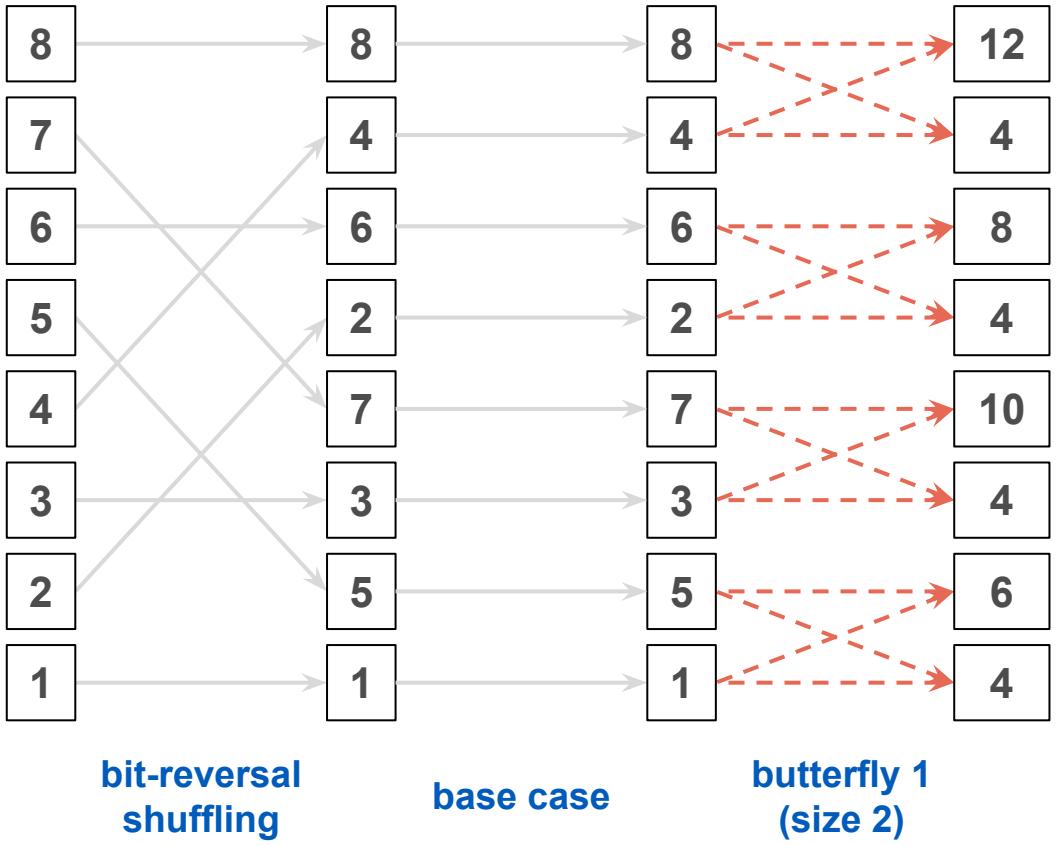
1) Shuffle the array based on *bit-reversed* indices

Illustrative Example



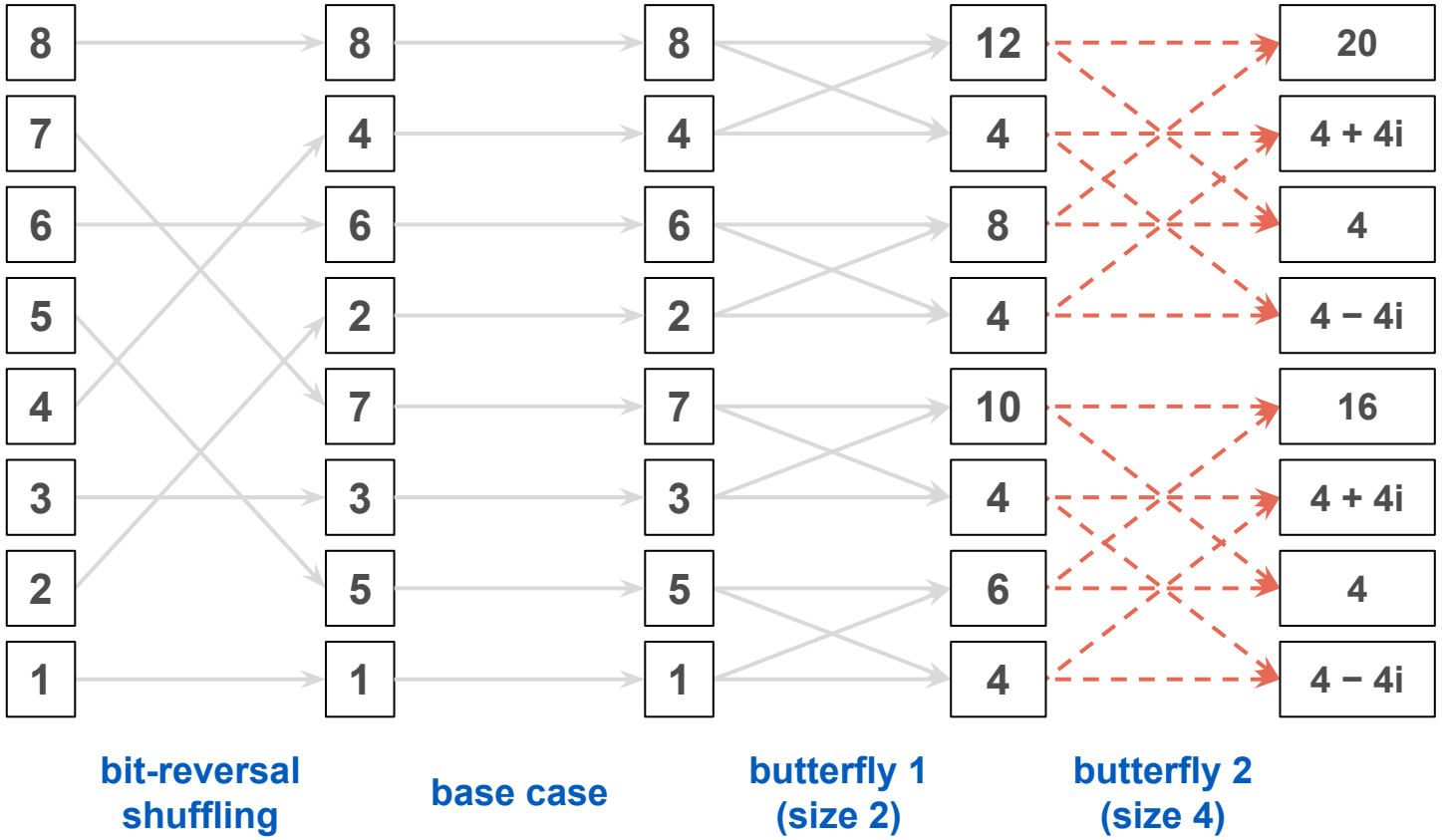
2) Base case, values carry over

Illustrative Example



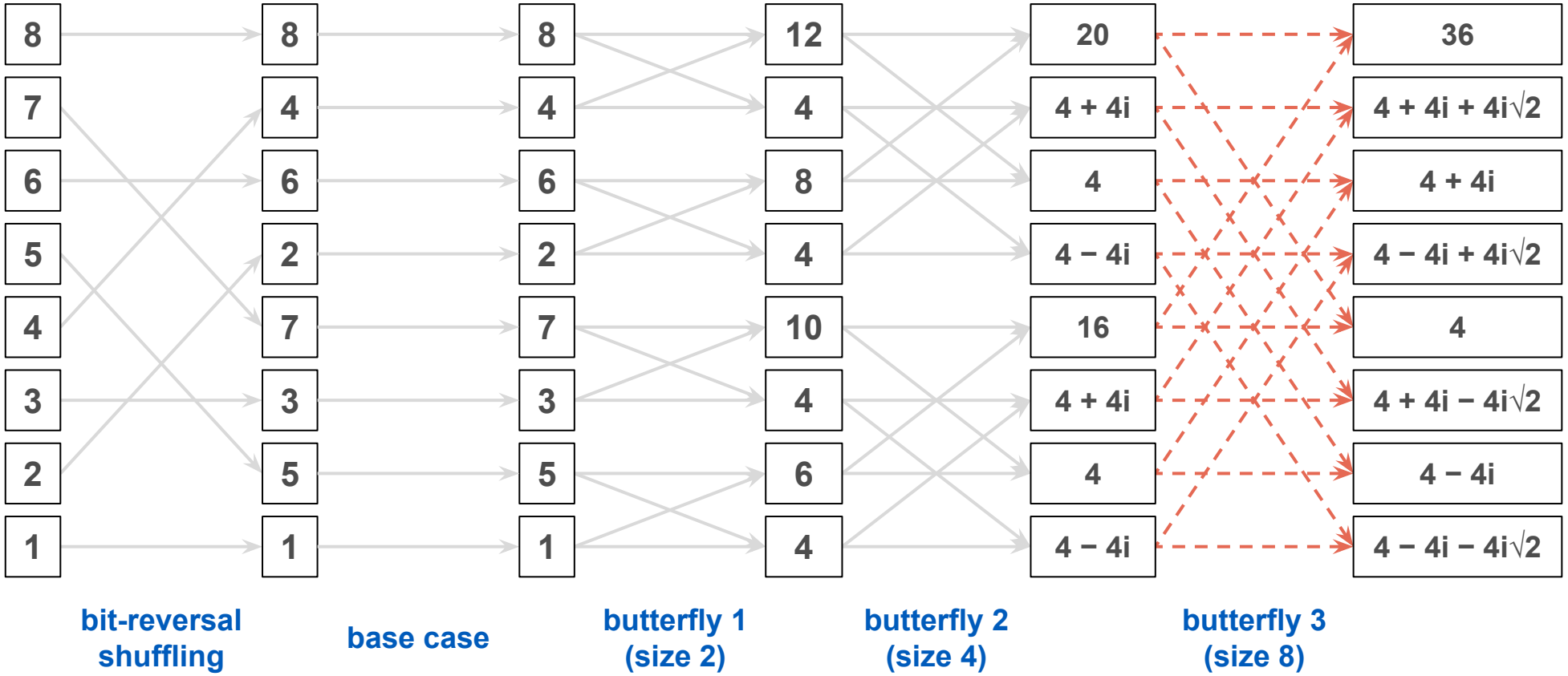
3) First butterfly, values *one* space apart

Illustrative Example



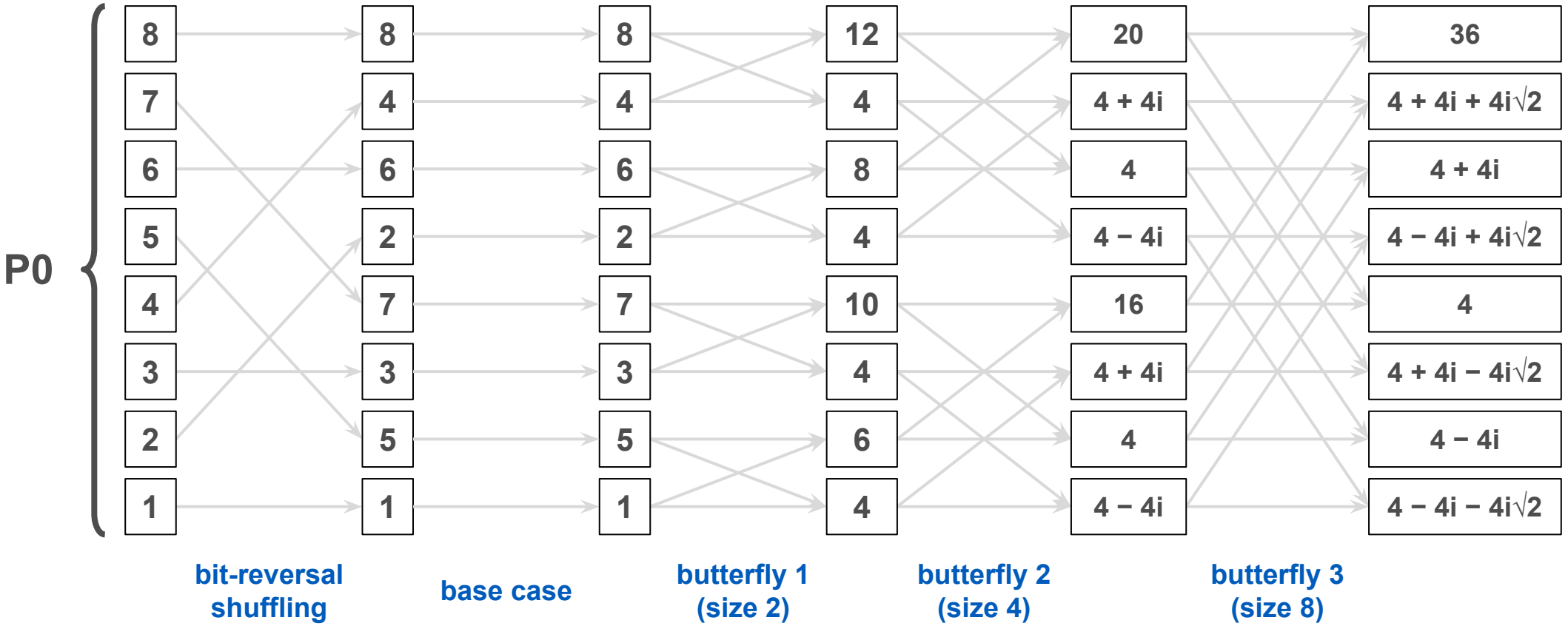
4) Second butterfly, values *two* spaces apart

Illustrative Example

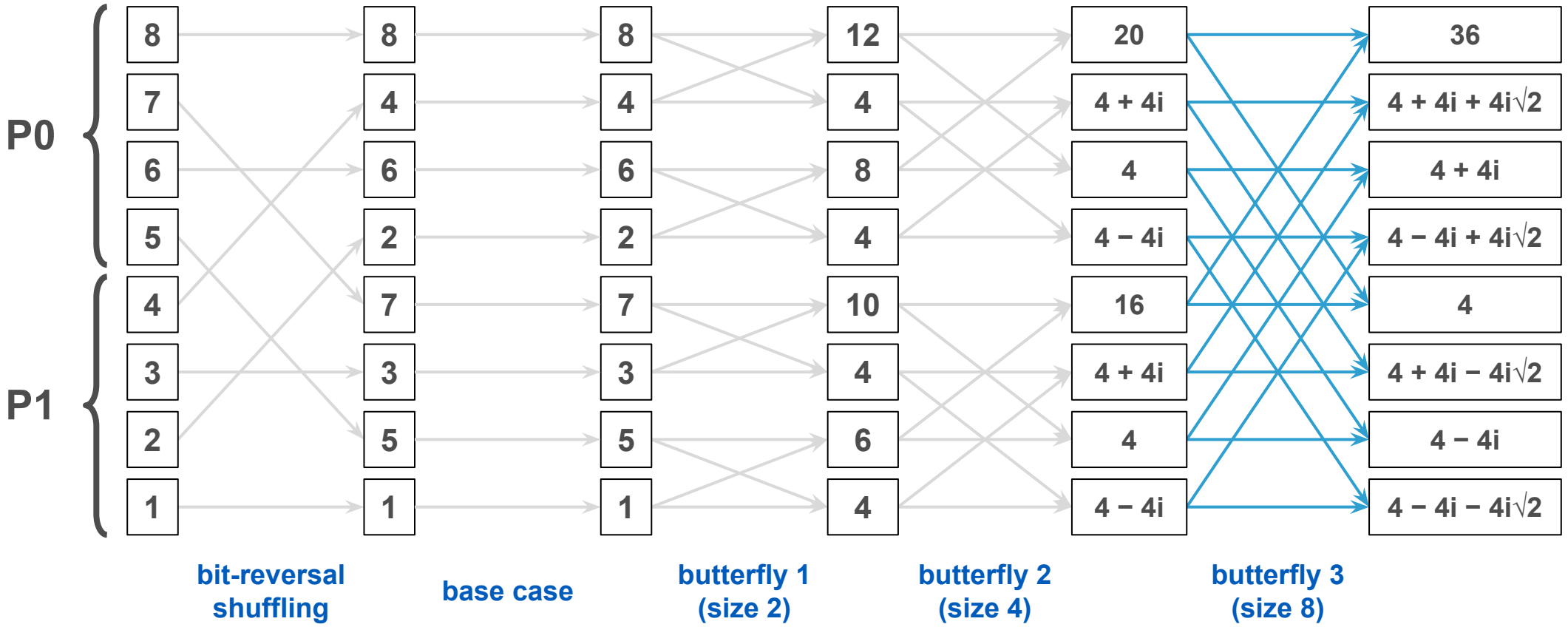


5) Third butterfly, values *four* spaces apart

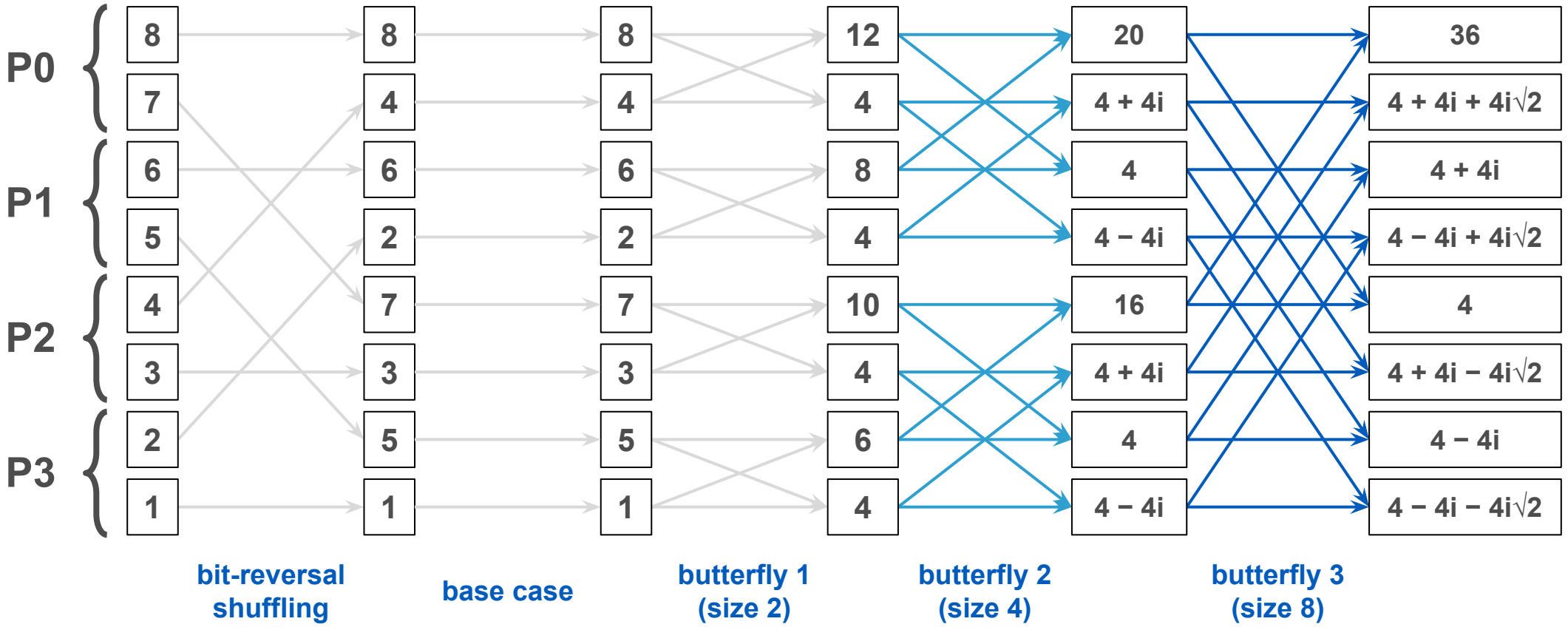
Sequential Implementation



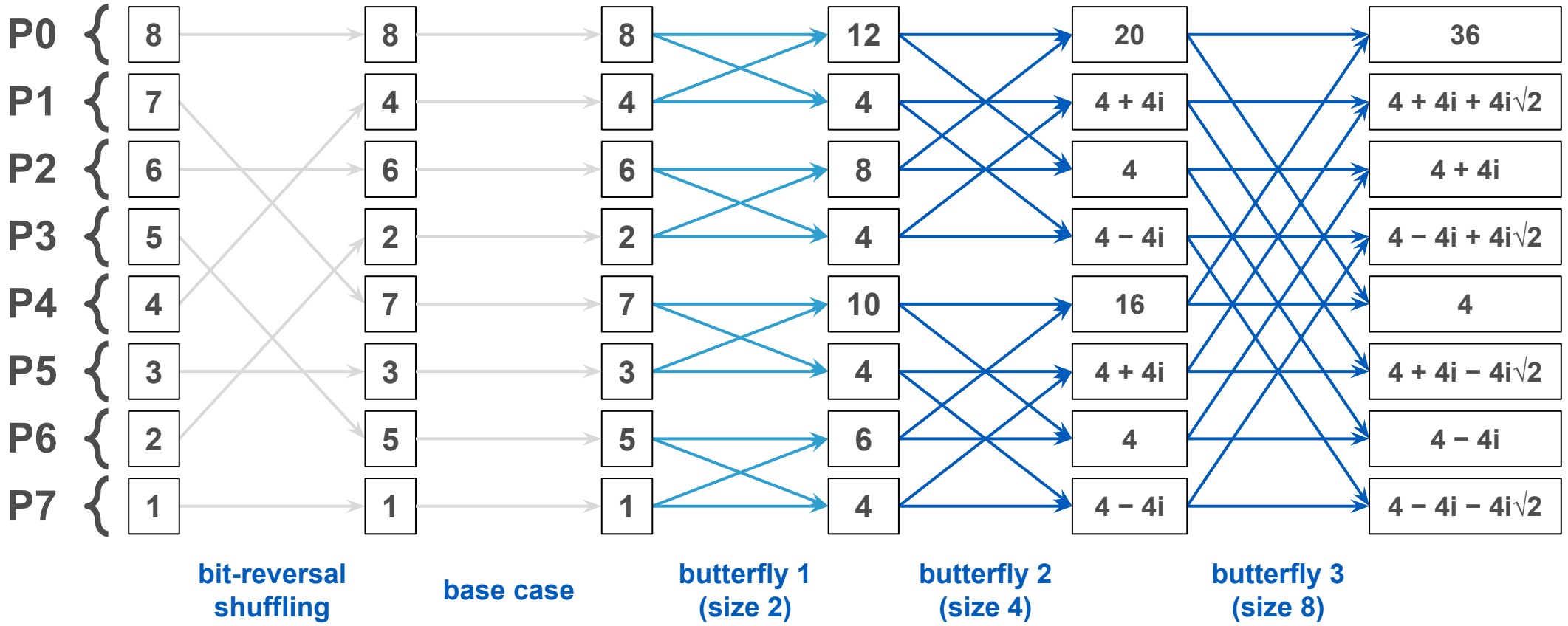
Parallel Implementation · 2P



Parallel Implementation · 4P



Parallel Implementation · 8P



Parallel Details

Each processor performs a **local** shuffle & butterfly, then a **parallel** butterfly.

BUDDY SYSTEM

In each parallel butterfly, $\mathbf{P_i}$ and $\mathbf{P_j}$ (**i < j**) must trade data. For a processor $\mathbf{P_x}$ and size **k**:

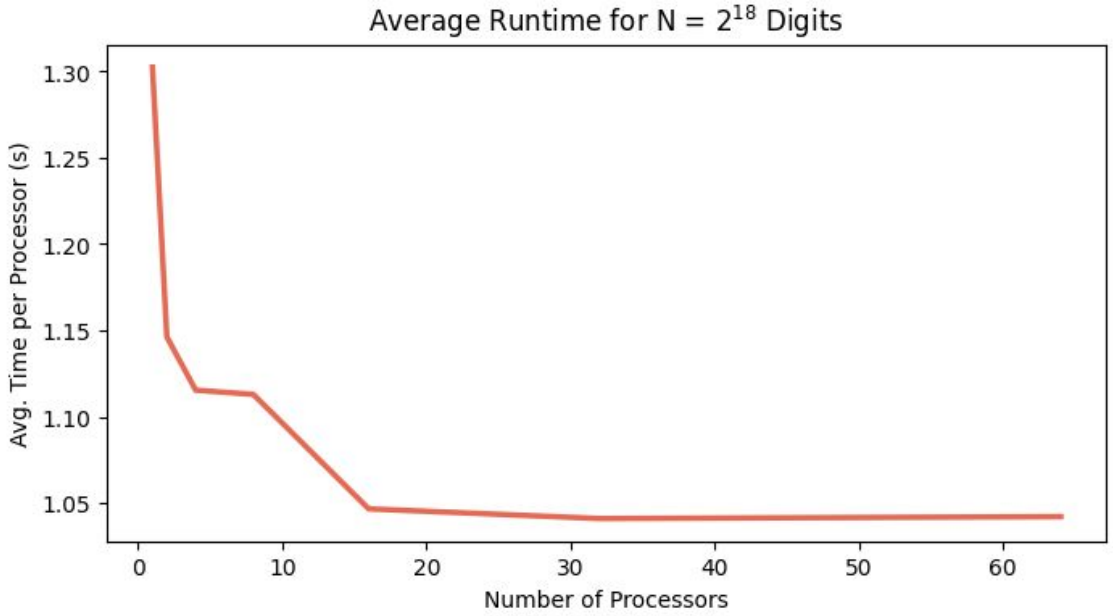
- **Find Type:** If **(x % k) < (k/2)**, $\mathbf{P_x}$ is **i**-type. Else, **j**-type
- **Find Buddy:** If **i**-type, **j = i + (k/2)**. If **j**-type, **i = j − (k/2)**

PARALLEL EXCHANGE

Each processor uses `MPI_Sendrecv` to send and receive adjacent data into the same array.

FFT Results · Scaling with Processors

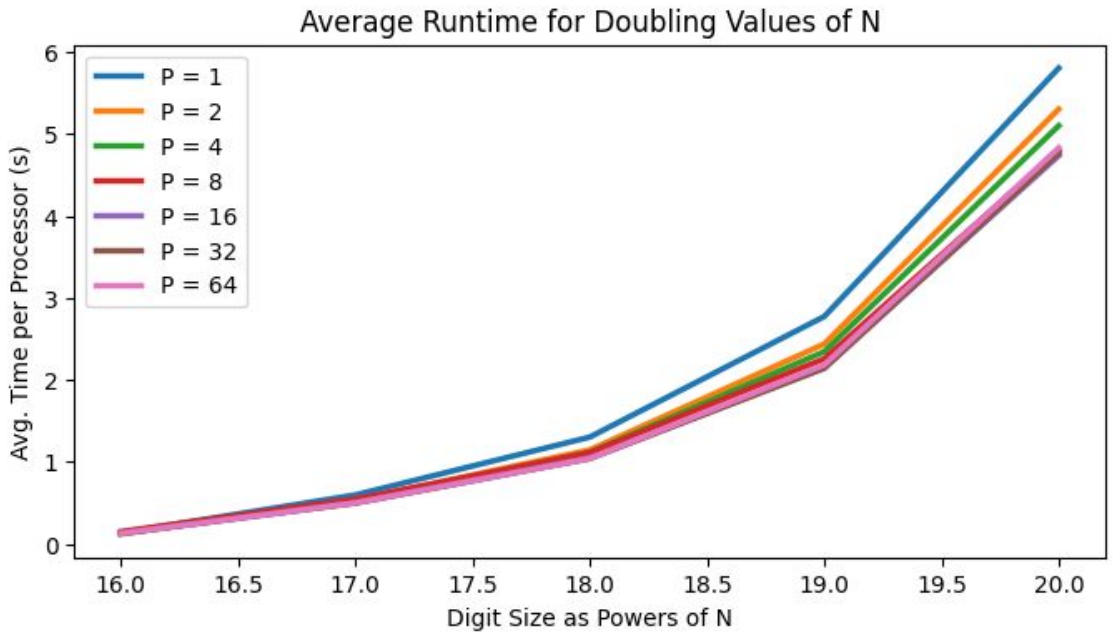
PROCESSORS	AVG. TIME	SPEEDUP
1	1.302s	
2	1.146s	×1.13
4	1.115s	×1.16
8	1.113s	×1.17
16	1.047s	×1.24
32	1.041s	×1.25
64	1.042s	×1.25



10 test cycles, local & parallel computations

FFT Results · Scaling with Data Size

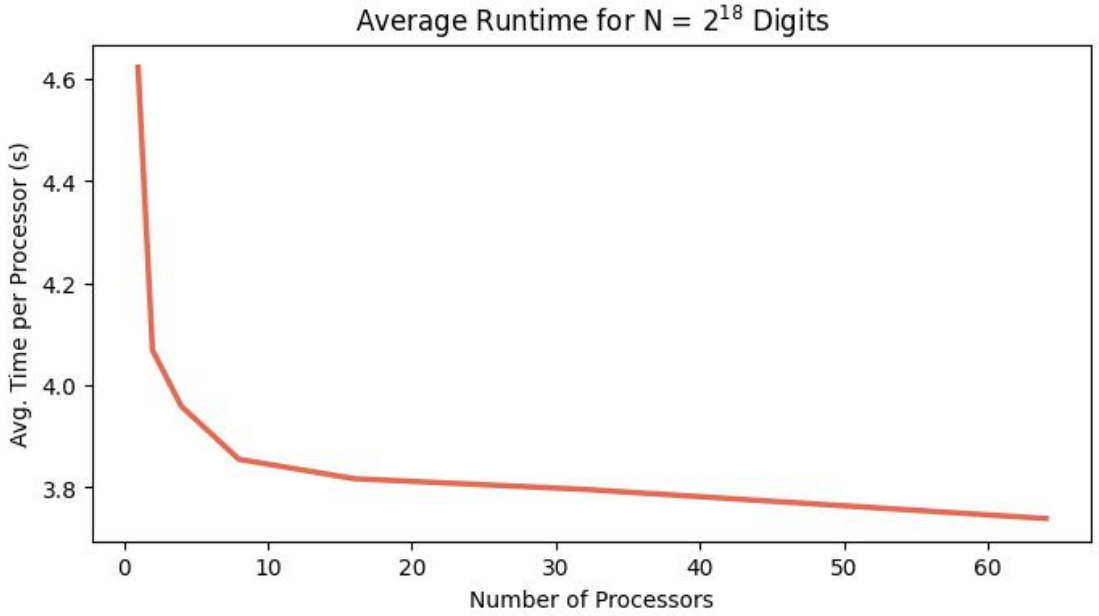
P	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
1	0.133s	0.597s	1.302s	2.773s	5.803s
2	0.119s	0.534s	1.146s	2.438s	5.303s
4	0.126s	0.524s	1.115s	2.344s	5.099s
8	0.149s	0.546s	1.112s	2.257s	4.798s
16	0.127s	0.500s	1.046s	2.151s	4.735s
32	0.127s	0.499s	1.042s	2.140s	4.764s
64	0.126s	0.497s	1.041s	2.183s	4.830s



10 test cycles, local & parallel computations

SSA Results · Scaling with Processors

PROCESSORS	AVG. TIME	SPEEDUP
1	4.623s	
2	4.069s	×1.12
4	3.959s	×1.16
8	3.855s	×1.19
16	3.817s	×1.21
32	3.796s	×1.22
64	3.739s	×1.23



10 test cycles, local & parallel computations

Theoretical Next Steps

1. Algorithmic Optimizations
   - Reduced shuffling
   - FFT speedups
   - SSA speedups
2. Code Optimizations
   - Identify & alleviate bottlenecks
   - Chunk-based MPI calls
   - Non-blocking MPI calls
3. Further Testing
   - Even more processors
   - Even more (& varied) data

Thank You!

Questions?

References

[1] "Joseph Fourier". Wikipedia. https://en.wikipedia.org/wiki/Joseph_Fourier

[2] "Milestones:First Demonstration of the Fast Fourier Transform (FFT), 1964". Engineering and Technology History Wiki. https://ethw.org/Milestones:First_Demonstration_of_the_Fast_Fourier_Transform_(FFT),_1964

[3] "Fast Fourier transform". Wikipedia. https://en.wikipedia.org/wiki/Fast_Fourier_transform

[4] Veritasium. "The Most Important Algorithm Of All Time". Youtube. https://youtu.be/nmgFG7PUHfo?si=0RI5H_9j8ksaKym4

[5] "Fourier Transform Formula". KeySight. https://www.keysight.com/used/us/en/knowledge/formulas/fourier-transform

[6] K. Azad. "An Interactive Guide To The Fourier Transform". BetterExplained. https://betterexplained.com/articles/an-interactive-guide-to-the-fourier-transform/

[7] Ali the Dazzling. "Fourier Transform Explained (for Beginners"). YouTube. https://youtu.be/iOsGkk63NfE?si=J9apfD9EYvYMiDov

[8] "Fourier transform". Wikipedia. https://en.wikipedia.org/wiki/Fourier_transform

[9] "Discrete Fourier transform". Wikipedia. https://en.wikipedia.org/wiki/Discrete_Fourier_transform

[10] S. Xu. "Discrete Fourier Transform - Simple Step by Step". YouTube. https://youtu.be/mkGsMWi_j4Q?si=ESECqm06sRFluxIe

[11] Reducible. "The Fast Fourier Transform (FFT): Most Ingenious Algorithm Ever?". YouTube. https://youtu.be/h7apO7q16V0?si=hEyBRmPjFLx4tPoW

[12] J. W. Cooley, J. W. Tukey. "An algorithm for the machine calculation of complex Fourier series". American Mathematical Society. https://doi.org/10.1090/S0025-5718-1965-0178586-1

[13] "Schönhage-Strassen algorithm". Wikipedia. https://en.wikipedia.org/wiki/Sch%C3%B6nhage%E2%80%93Strassen_algorithm

[14] "Arnold Schönhage". Wikipedia. https://en.wikipedia.org/wiki/Arnold_Sch%C3%B6nhage

[15] "Volker Strassen". Wikipedia. https://en.wikipedia.org/wiki/Volker_Strassen

[16] "Discrete Fourier transform over a ring". Wikipedia. https://en.wikipedia.org/wiki/Discrete_Fourier_transform_over_a_ring