# Parallel Numerical Algorithms

## Chapter 13 – Fast Fourier Transform

Prof. Michael T. Heath

Department of Computer Science
University of Illinois at Urbana-Champaign

CS 554 / CSE 512





## Outline







## Roots of Unity

- For given integer $n$, we use notation

$$\omega_n = \cos(2\pi/n) - i\sin(2\pi/n) = e^{-2\pi i/n}$$

  for primitive $n$th root of unity, where $i = \sqrt{-1}$
- $n$th roots of unity, sometimes called *twiddle factors* in this context, are then given by $\omega_n^k$ or by $\omega_n^{-k}$, $k = 0, \ldots, n-1$
- For convenience, we will assume that $n$ is power of two, and all logarithms used will be base two
- We will also index sequences (components of vectors) starting from $0$ rather than $1$





## Discrete Fourier Transform

- *Discrete Fourier Transform*, or *DFT*, of sequence $\boldsymbol{x} = [x_0, \ldots, x_{n-1}]^T$ is sequence $\boldsymbol{y} = [y_0, \ldots, y_{n-1}]^T$ given by

$$y_m = \sum_{k=0}^{n-1} x_k\, \omega_n^{mk}, \quad m = 0, 1, \ldots, n-1$$

  or

$$\boldsymbol{y} = \boldsymbol{F}_n \boldsymbol{x}$$

  where entries of Fourier matrix $\boldsymbol{F}_n$ are given by

$$\{\boldsymbol{F}_n\}_{mk} = \omega_n^{mk}$$





## Inverse DFT

- It is easily seen that

$$\boldsymbol{F}_n^{-1} = (1/n)\boldsymbol{F}_n^H$$

- So *inverse DFT* is given by

$$x_k = \frac{1}{n}\sum_{m=0}^{n-1} y_m\, \omega_n^{-mk} \quad k = 0, 1, \ldots, n-1$$





## Example

$$\boldsymbol{F}_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}$$

$$4\,\boldsymbol{F}_4^{-1} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \omega^{-3} \\ 1 & \omega^{-2} & \omega^{-4} & \omega^{-6} \\ 1 & \omega^{-3} & \omega^{-6} & \omega^{-9} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$





## Computing DFT

- To illustrate, consider computing DFT for $n = 4$,

$$y_m = \sum_{k=0}^{3} x_k\, \omega_n^{mk}, \quad m = 0, \ldots, 3$$

- Writing out equations in full,

$$\begin{aligned} y_0 &= x_0\omega_n^0 + x_1\omega_n^0 + x_2\omega_n^0 + x_3\omega_n^0 \\ y_1 &= x_0\omega_n^0 + x_1\omega_n^1 + x_2\omega_n^2 + x_3\omega_n^3 \\ y_2 &= x_0\omega_n^0 + x_1\omega_n^2 + x_2\omega_n^4 + x_3\omega_n^6 \\ y_3 &= x_0\omega_n^0 + x_1\omega_n^3 + x_2\omega_n^6 + x_3\omega_n^9 \end{aligned}$$





## Computing DFT

- Noting that

$$\omega_n^0 = \omega_n^4 = 1, \quad \omega_n^2 = \omega_n^6 = -1, \quad \omega_n^9 = \omega_n^1$$

  and regrouping, we obtain

$$\begin{aligned} y_0 &= (x_0 + \omega_n^0 x_2) + \omega_n^0(x_1 + \omega_n^0 x_3) \\ y_1 &= (x_0 - \omega_n^0 x_2) + \omega_n^1(x_1 - \omega_n^0 x_3) \\ y_2 &= (x_0 + \omega_n^0 x_2) + \omega_n^2(x_1 + \omega_n^0 x_3) \\ y_3 &= (x_0 - \omega_n^0 x_2) + \omega_n^3(x_1 - \omega_n^0 x_3) \end{aligned}$$

- DFT can now be computed with only $8$ additions and $6$ multiplications, instead of expected $(4-1)*4 = 12$ additions and $4^2 = 16$ multiplications

## Computing DFT

- Actually, even fewer multiplications are required for this small case, since $\omega_n^0 = 1$, but we have tried to illustrate how algorithm works in general
- Main point is that computing DFT of original $4$-point sequence has been reduced to computing DFT of its two $2$-point even and odd subsequences
- This property holds in general: DFT of $n$-point sequence can be computed by breaking it into two DFTs of half length, provided $n$ is even





## Computing DFT

- General pattern becomes clearer when viewed in terms of first few Fourier matrices

$$\boldsymbol{F}_1 = 1, \quad \boldsymbol{F}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad \boldsymbol{F}_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}$$

- Let $\boldsymbol{P}_4$ be permutation matrix

$$\boldsymbol{P}_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$





## Computing DFT

- Let $\boldsymbol{D}_2$ be diagonal matrix

$$\boldsymbol{D}_2 = \mathrm{diag}(1, \omega_4) = \begin{bmatrix} 1 & 0 \\ 0 & -i \end{bmatrix}$$

- Then we have

$$\boldsymbol{F}_4 \boldsymbol{P}_4 = \left[\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & -1 & -i & i \\ \hline 1 & 1 & -1 & -1 \\ 1 & -1 & i & -i \end{array}\right] = \begin{bmatrix} \boldsymbol{F}_2 & \boldsymbol{D}_2\boldsymbol{F}_2 \\ \boldsymbol{F}_2 & -\boldsymbol{D}_2\boldsymbol{F}_2 \end{bmatrix}$$

- Thus, $\boldsymbol{F}_4$ can be rearranged so that each block is diagonally scaled version of $\boldsymbol{F}_2$
- Such hierarchical splitting can be carried out at each level, provided number of points is even





## Computing DFT

- In general, $\boldsymbol{P}_n$ is permutation that groups even-numbered columns of $\boldsymbol{F}_n$ before odd-numbered columns, and

$$\boldsymbol{D}_{n/2} = \mathrm{diag}\left(1, \omega_n, \ldots, \omega_n^{(n/2)-1}\right)$$

- To apply $\boldsymbol{F}_n$ to sequence of length $n$, we need merely apply $\boldsymbol{F}_{n/2}$ to its even and odd subsequences and scale results, where necessary, by $\pm\boldsymbol{D}_{n/2}$
- Resulting recursive divide-and-conquer algorithm for computing DFT is called *Fast Fourier Transform*, or *FFT*
- FFT is particular way of computing DFT efficiently





## FFT Algorithm

**procedure** $\texttt{fft}(x, y, n, \omega)$
**if** $n = 1$ **then**
  $y[0] = x[0]$
**else**
  **for** $k = 0$ **to** $(n/2) - 1$
    $p[k] = x[2k]$
    $s[k] = x[2k+1]$
  **end**
  $\texttt{fft}(p, q, n/2, \omega^2)$
  $\texttt{fft}(s, t, n/2, \omega^2)$
  **for** $k = 0$ **to** $n - 1$
    $y[k] = q[k \mod (n/2)] + \omega^k t[k \mod (n/2)]$
  **end**
**end**





## Complexity of FFT Algorithm

- There are $\log n$ levels of recursion, each of which involves $\Theta(n)$ arithmetic operations, so total cost is $\Theta(n \log n)$
- By contrast, straightforward evaluation of matrix-vector product defining DFT requires $\Theta(n^2)$ arithmetic operations, which is enormously greater for long sequences

| $n$ | $n \log n$ | $n^2$ |
|---|---|---|
| 64 | 384 | 4096 |
| 128 | 896 | 16384 |
| 256 | 2048 | 65536 |
| 512 | 4608 | 262144 |
| 1024 | 10240 | 1048576 |





## FFT Algorithm

- For clarity, separate arrays were used for subsequences, but transform can be computed in place using no additional storage
- Input sequence is assumed complex; if input sequence is real, then additional symmetries in DFT can be exploited to reduce storage and operation count by half
- Output sequence is not produced in natural order, but either input or output sequence can be rearranged at cost of $\Theta(n \log n)$, analogous to sorting
- FFT algorithm can be formulated using iteration rather than recursion, which is often desirable for greater efficiency or when programming language does not support recursion



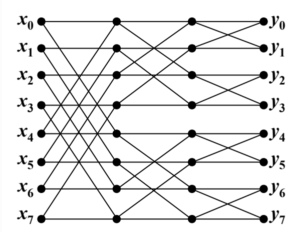


## Computing Inverse DFT

- Because of similar form of DFT and its inverse, FFT algorithm can also be used to compute inverse DFT efficiently
- Ability to transform back and forth quickly between time and frequency domains makes it practical to perform any computations or analysis that may be required in whichever domain is more convenient and efficient

## Binary Exchange Parallel FFT

- To obtain fine-grain decomposition of FFT, we assign input data $x_k$ to task $k$, which also computes result $y_k$




- At stage $m$ of algorithm, tasks $k$ and $j$ exchange data, where $k$ and $j$ differ only in their $m$th bits





## Binary Exchange Parallel FFT

- There are $n$ tasks and $\log n$ stages, so parallel time required to compute FFT is

$$T_n = (t_c + t_s + t_w)\,\log n$$

  where $t_c$ is cost of multiply-add, and $t_s + t_w$ is cost of exchanging one number between pair of tasks at each stage

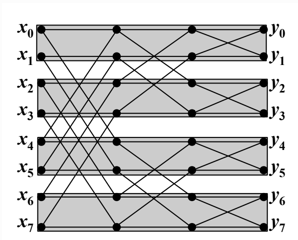
- Hypercube is natural network for FFT algorithm





## Binary Exchange Parallel FFT

- To obtain smaller number of coarse-grain tasks, agglomerate sets of $n/p$ components of input and output vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, where we assume $p$ is also power of two








## Binary Exchange Parallel FFT

- Components having their $\log p$ most significant bits in common are assigned to same task
- Thus, exchanges are required in binary exchange algorithm only for first $\log p$ stages, since data are local for remaining $\log(n/p)$ stages





## Binary Exchange Parallel FFT

- Each stage involves updating of $n/p$ components by each task, and exchange of $n/p$ components for each of first $\log p$ stages
- Thus, total time required using hypercube network is

$$T_p = t_c\, n\,(\log n)/p + t_s\,(\log p) + t_w\, n\,(\log p)/p$$

- To determine isoefficiency function, set

$$t_c\, n\, \log n \approx E\,(t_c\, n\, \log n + t_s\, p\, \log p + t_w\, n\, \log p)$$

  which holds if $n = \Theta(p)$, so isoefficiency function is $\Theta(p\, \log p)$, since $T_1 = \Theta(n\, \log n)$





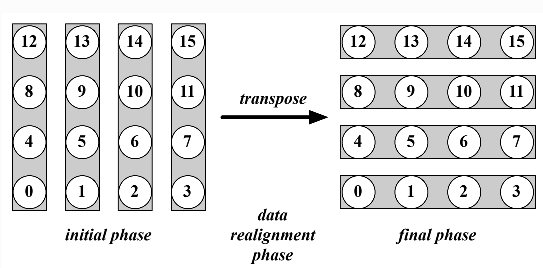
## Transpose Parallel FFT

- Binary exchange algorithm has one phase that is communication free and another phase that requires communication at each stage
- Another approach is to realign data so that both computational phases are communication free, and only communication is for data realignment phase between computational phases
- To accomplish this, data can be organized in $\sqrt{n} \times \sqrt{n}$ array, as illustrated next for $n = 16$





## Transpose Parallel FFT








## Transpose Parallel FFT

- If array is partitioned by columns, which are assigned to $p \le \sqrt{n}$ tasks, then no communication is required for first $\log(\sqrt{n}\,)$ stages
- Data are then transposed using all-to-all personalized collective communication, so that each *row* of data array is now stored in single task
- Thus, final $\log(\sqrt{n}\,)$ stages now require no communication
- Overall performance of transpose algorithm depends on particular implementation of all-to-all personalized collective communication

## Transpose Parallel FFT

- Straightforward approach yields total parallel time

$$T_p = t_c\, n\, (\log n)/p + t_s\, p + t_w\, n/p$$

- Compared with binary exchange algorithm, transpose algorithm has higher cost due to message start-up but lower cost due to per-word transfer time
- Thus, choice of algorithm depends on relative values of $t_s$ and $t_w$ for given parallel system





## References

- A. Averbuch and E. Gabber, Portable parallel FFT for MIMD multiprocessors, *Concurrency: Practice and Experience* 10:583-605, 1998
- C. Calvin, Implementation of parallel FFT algorithms on distributed memory machines with a minimum overhead of communication, *Parallel Computing* 22:1255-1279, 1996
- R. M. Chamberlain, Gray codes, fast Fourier transforms, and hypercubes, *Parallel Computing* 6:225-233, 1988
- E. Chu and A. George, FFT algorithms and their adaptation to parallel processing, *Linear Algebra Appl.* 284:95-124, 1998





## References

- A. Edelman, Optimal matrix transposition and bit reversal on hypercubes: all-to-all personalized communication, *J. Parallel Computing* 11:328-331, 1991
- A. Gupta and V. Kumar, The Scalability of FFT on Parallel Computers, *IEEE Trans. Parallel Distrib. Sys.* 4:922-932, 1993
- R. B. Pelz, Parallel FFTs, D. E. Keyes, A. Sameh, and V. Venkatakrishnan, eds., *Parallel Numerical Algorithms*, pp. 245-266, Kluwer, 1997
- P. N. Swarztrauber, Multiprocessor FFTs, *Parallel Computing* 5:197-210, 1987