Fast Fourier Transform

CS121 Parallel Computing

Spring 2018

Fourier transform

- Translate an input from the space to time domain.
- FFT used for signal processing, operations on polynomials and matrices, convolutions and filtering, differential equations, machine learning, etc.
- One of IEEE's top 10 most important algorithms of the 20th century.
  - Popularized by Cooley and Tukey in 1965.
  - But variants known to Gauss in 1805!

In putting together this issue of *Computing in Science & Engineering*, we knew three things: it would be difficult to list just 10 algorithms; it would be fun to assemble the authors and read their papers; and, whatever we came up with in the end, it would be controversial. We tried to assemble the 10 algorithms with the greatest influence on the development and practice of science and engineering in the 20th century. Following is our list (here, the list is in chronological order; however, the articles appear in no particular order):

- Metropolis Algorithm for Monte Carlo
- Simplex Method for Linear Programming
- Krylov Subspace Iteration Methods
- The Decompositional Approach to Matrix Computations
- The Fortran Optimizing Compiler
- QR Algorithm for Computing Eigenvalues
- Quicksort Algorithm for Sorting
- Fast Fourier Transform
- Integer Relation Detection
- Fast Multipole Method

With each of these algorithms or approaches, there is a person or group receiving credit for inventing or discovering the method. Of course, the reality is that there is generally a culmination of ideas that leads to a method. In some cases, we chose authors who had a hand in developing the algorithm, and in other cases, the author is a leading authority.

1521-9615/00/$10.00 © 2000 IEEE

JACK DONGARRA
University of Tennessee and Oak Ridge National Laboratory
FRANCIS SULLIVAN
IDA Center for Computing Sciences

In this issue

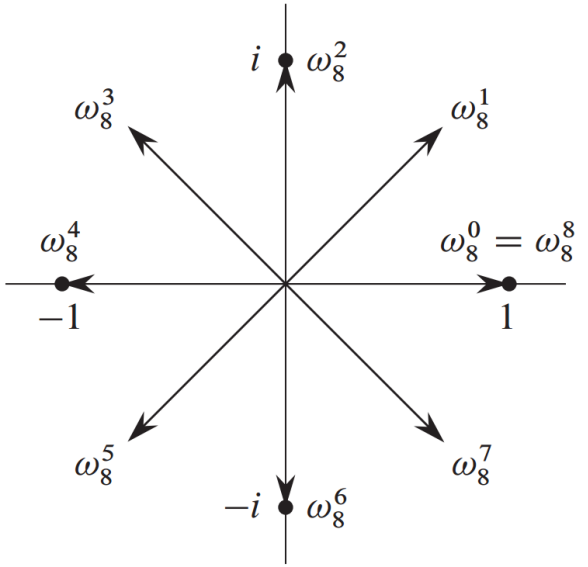
Monte Carlo methods are powerful tools for evaluating the properties of complex, many-body systems, as well as nondeterministic processes. Isabel Beichl and Francis Sullivan describe the Metropolis Algorithm. We are often confronted with problems that have an enormous number of dimensions or a process that involves a path with many possible branch points, each of which is governed by some fundamental probability of occurence. The solutions are not exact in a rigorous way, because we randomly sample the problem. However, it is possible to achieve nearly exact results using a relatively small number of samples compared to the problem's dimensions. Indeed, Monte Carlo methods are the only practical choice for evaluating problems of high dimensions.

John Nash describes the Simplex method for solving linear programming problems. (The use of the word *programming* here really refers to scheduling or planning—and not in the way that we tell a computer what must be done.) The Simplex method relies on noticing that the objective function's maximum must occur on a corner of the space bounded by the constraints of the "feasible region."

Large-scale problems in engineering and science often require solution of sparse linear algebra problems, such as systems of equations. The importance of iterative algorithms in linear algebra stems from the simple fact that a direct approach will require $O(N^3)$ work. The Krylov subspace iteration methods have led to a major change in how users deal with large, sparse, nonsymmetric matrix problems. In this article, Henk van der Vorst describes the state of the art in terms of

Roots of unity

- An n'th root of unity is an ω s.t. $\omega^n = 1$.
- There are n roots n'th roots of unity, and they have the form $e^{\frac{2\pi ik}{n}}$, for $0 \le k \le n-1$.
  - Write $\omega_n = e^{\frac{2\pi i}{n}}$, so that the n'th roots of unity are $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$.
- Fact 1 $\omega_n^n = 1$.
- Fact 2 $\omega_n^{k+\frac{n}{2}} = -\omega_n^k$.
- Fact 3 $\left(\omega_n^k\right)^2 = \omega_n^{2k} = \omega_{n/2}^k$.



Source: Introduction to Algorithms, Cormen et al.

Discrete Fourier Transform

- Given a degree n polynomial $A = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$, DFT computes $A(\omega_n^0), A(\omega_n^1), \ldots, A(\omega_n^{n-1})$.
- Since $A = a_0 + x(a_1 + x(a_2 + \cdots) \ldots)$, $A(x)$ can be evaluated in O(n) time and O(1) memory.
- DFT can be computed trivially in $O(n^2)$ time.

Fast Fourier Transform

- FFT computes the DFT in O(n log n) time using divide and conquer.
  - Suppose n is a power of 2.
- Let $A^{[0]} = a_0 + a_2x^1 + a_4x^2 + \cdots + a_{n-2}x^{\frac{n}{2}-1}$

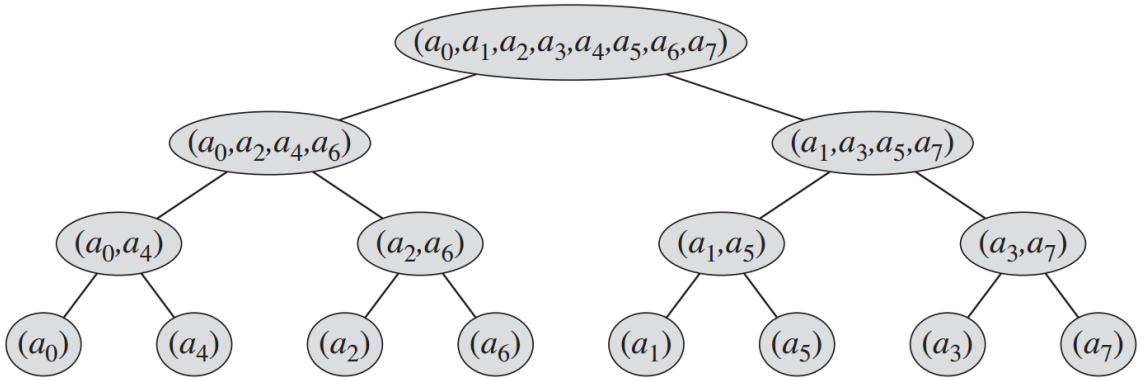
  $A^{[1]} = a_1 + a_3x^1 + a_5x^2 + \cdots + a_{n-1}x^{\frac{n}{2}-1}$.
- Then $A(x) = A^{[0]}\left(x^2\right) + xA^{[1]}\left(x^2\right)$.
- So can compute $A\left(\omega_n^0\right), A(\omega_n^1), \ldots, A(\omega_n^{n-1})$ by computing $A^{[0]}$ and $A^{[1]}$ at $\left(\omega_n^0\right)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2$, and multiplying some terms by $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$.
- But $\left(\omega_n^{k+n/2}\right)^2 = \omega_n^{2k+n} = \omega_n^{2k} = \left(\omega_n^k\right)^2$ by Fact 1.
  - So $\left\{\left(\omega_n^0\right)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2\right\} = \left\{\left(\omega_n^0\right)^2, (\omega_n^1)^2, \ldots, \left(\omega_n^{n/2-1}\right)^2\right\}$, i.e. only need to evaluate $A^{[0]}$ and $A^{[1]}$ at n/2 points instead of n.
- Also, $\left(\omega_n^k\right)^2 = \omega_n^{2k} = \omega_{n/2}^k$.
  - So $\left\{\left(\omega_n^0\right)^2, (\omega_n^1)^2, \ldots, \left(\omega_n^{n/2-1}\right)^2\right\} = \left\{\omega_{n/2}^0, \omega_{n/2}^1, \ldots, \omega_{n/2}^{n/2-1}\right\}$, i.e. only need to evaluate $A^{[0]}$ and $A^{[1]}$ at the (n/2)'th roots of unity.

Fast Fourier Transform

- Thus, computing $A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2)$ for $x \in \{\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}\}$ requires
  - Computing for $A^{[0]}(x)$ and $A^{[1]}(x)$ for $x \in \left\{\omega_{n/2}^0, \omega_{n/2}^1, \ldots, \omega_{n/2}^{n/2-1}\right\}$.
    - These are also DFT's, so can be done recursively using two n/2-point FFT's.
  - For $0 \le k \le \frac{n}{2} - 1$
    - $A(\omega_n^k) = A^{[0]}\left(\omega_{n/2}^k\right) + \omega_n^k \, A^{[1]}\left(\omega_{n/2}^k\right)$
    - $A\left(\omega_n^{k+n/2}\right) = A^{[0]}\left(\omega_{n/2}^{k+n/2}\right) + \omega_n^{k+n/2} \, A^{[1]}\left(\omega_{n/2}^{k+n/2}\right)$
      $= A^{[0]}\left(\omega_{n/2}^k\right) - \omega_n^k \, A^{[1]}\left(\omega_{n/2}^k\right)$
- Let T(n) be the time to compute a size n FFT. Then $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$, so $T(n) = \Theta(n \log n)$.

Recursive FFT

```
RECURSIVE-FFT(a)
 1  n = a.length
 2  if n == 1
 3       return a
 4  ω_n = e^{2πi/n}
 5  ω = 1
 6  a^[0] = (a_0, a_2, ..., a_{n-2})
 7  a^[1] = (a_1, a_3, ..., a_{n-1})
 8  y^[0] = RECURSIVE-FFT(a^[0])
 9  y^[1] = RECURSIVE-FFT(a^[1])
10  for k = 0 to n/2 - 1
11       y_k = y_k^[0] + ω y_k^[1]
12       y_{k+(n/2)} = y_k^[0] - ω y_k^[1]
13       ω = ω ω_n
14  return y
```



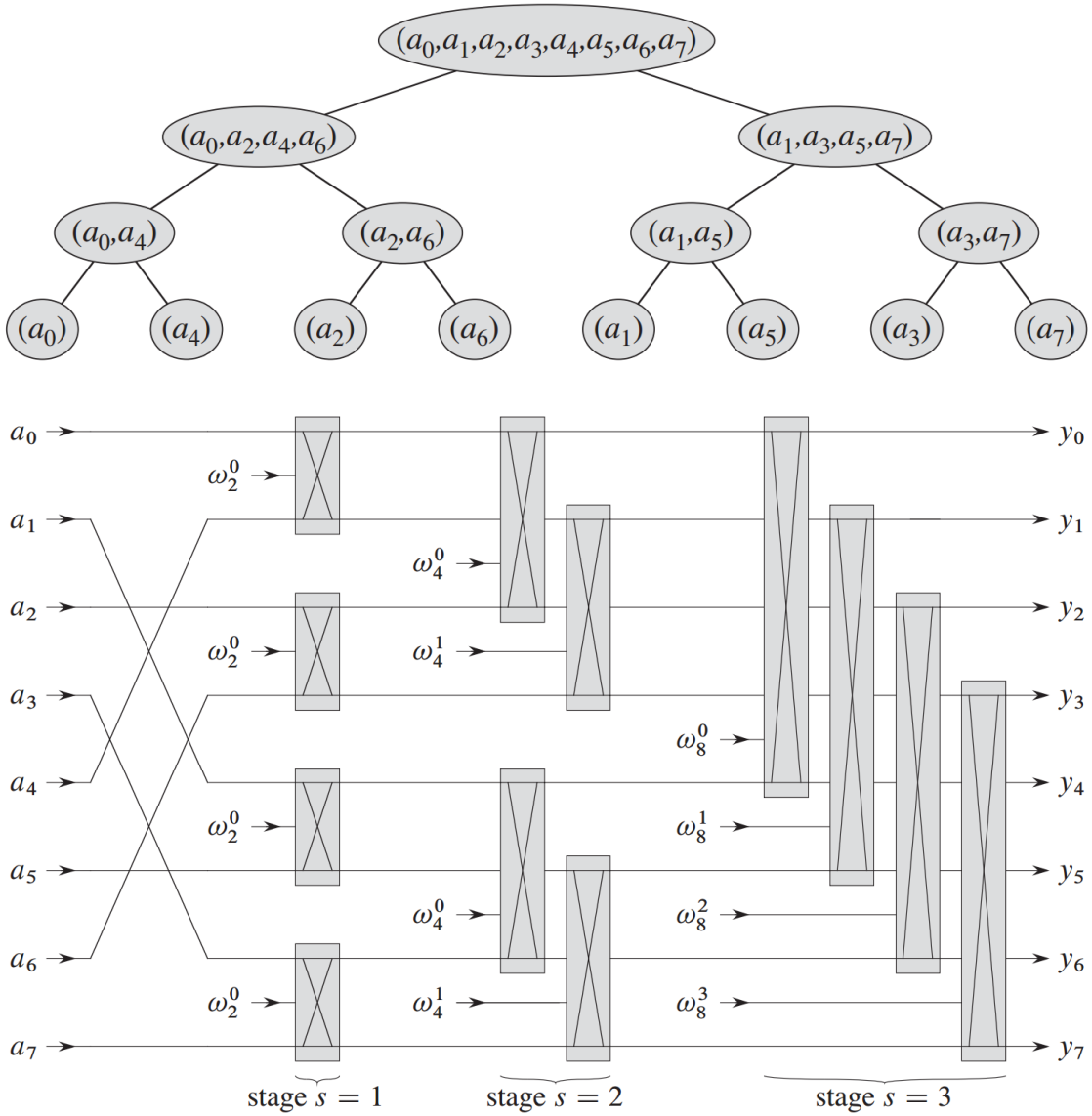
- Note the order that the inputs a_0,..., a_{n-1} appear at the bottom level of recursion, namely $a_0, a_4, a_2, \ldots, a_7$.
- This is a bit reversed order, because if we reverse the bits in the indices (e.g. $a_4 = a_{100}$ becomes $a_{001} = a_1$), then we get the order $a_0, a_1, a_2, \ldots, a_7$.

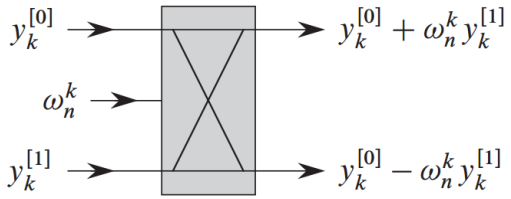
Iterative FFT

```
ITERATIVE-FFT(a)
 1  BIT-REVERSE-COPY(a, A)
 2  n = a.length
 3  for s = 1 to lg n
 4      m = 2^s
 5      ω_m = e^{2πi/m}
 6      for k = 0 to n − 1 by m
 7          ω = 1
 8          for j = 0 to m/2 − 1
 9              t = ω A[k + j + m/2]
10              u = A[k + j]
11              A[k + j] = u + t
12              A[k + j + m/2] = u − t
13              ω = ω ω_m
14  return A
```

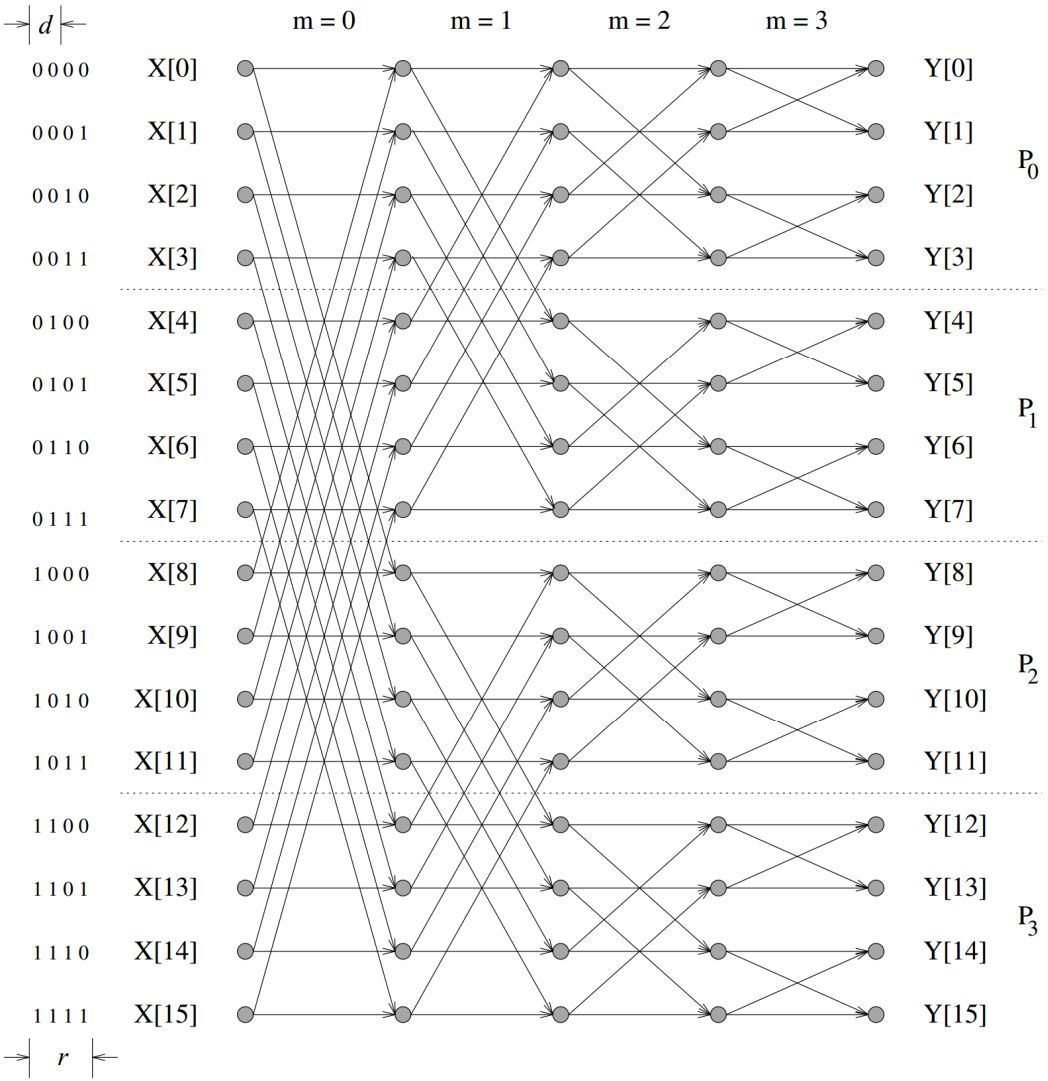
```
RECURSIVE-FFT(a)
 1  n = a.length
 2  if n == 1
 3      return a
 4  ω_n = e^{2πi/n}
 5  ω = 1
 6  a^[0] = (a_0, a_2, ..., a_{n−2})
 7  a^[1] = (a_1, a_3, ..., a_{n−1})
 8  y^[0] = RECURSIVE-FFT(a^[0])
 9  y^[1] = RECURSIVE-FFT(a^[1])
10  for k = 0 to n/2 − 1
11      y_k = y_k^[0] + ω y_k^[1]
12      y_{k+(n/2)} = y_k^[0] − ω y_k^[1]
13      ω = ω ω_n
14  return y
```

FFT circuit


$(a_0,a_1,a_2,a_3,a_4,a_5,a_6,a_7)$
(a_0,a_2,a_4,a_6)
(a_1,a_3,a_5,a_7)
(a_0,a_4)
(a_2,a_6)
(a_1,a_5)
(a_3,a_7)
(a_0)
(a_4)
(a_2)
(a_6)
(a_1)
(a_5)
(a_3)
(a_7)
a_0
a_1
a_2
a_3
a_4
a_5
a_6
a_7
y_0
y_1
y_2
y_3
y_4
y_5
y_6
y_7
ω_2^0
ω_4^0
ω_4^1
ω_8^0
ω_8^1
ω_8^2
ω_8^3
stage $s = 1$
stage $s = 2$
stage $s = 3$


$y_k^{[0]}$
ω_n^k
$y_k^{[1]}$
$y_k^{[0]} + \omega_n^k y_k^{[1]}$
$y_k^{[0]} - \omega_n^k y_k^{[1]}$

Binary exchange FFT



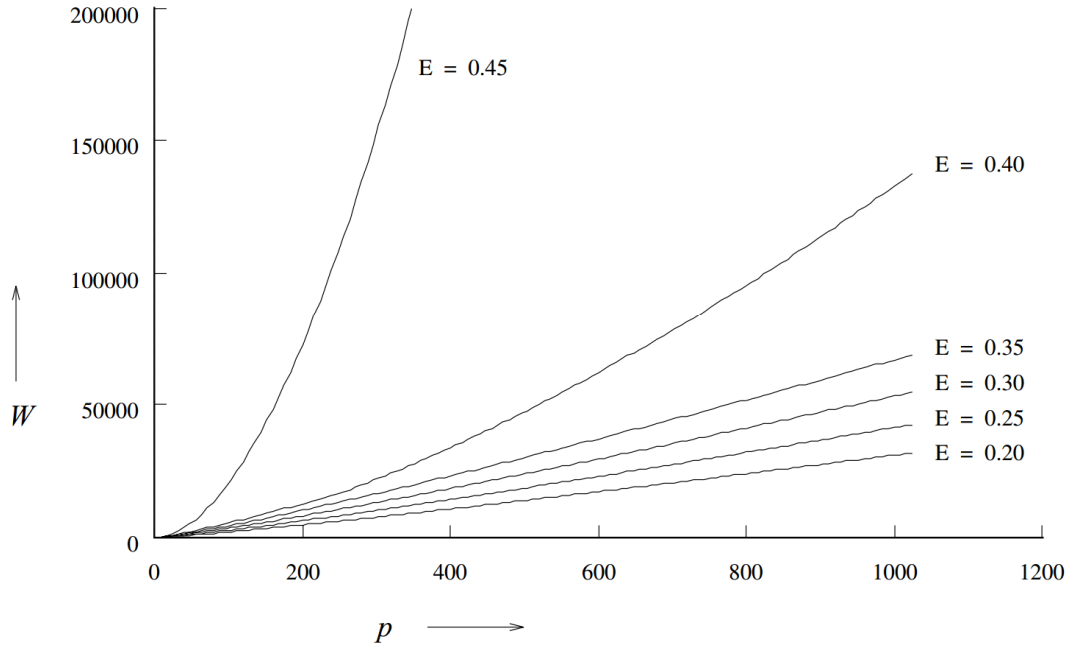
- Consider doing n point FFT using p ≤ n processors.
  - Assume n, p are both powers of 2.
  - Let r = log n, d = log p.
- Map n/p elements of input and output per processor.
- In total r stages of computation.
- In i'th stage, processors whose IDs differ in i'th digit communicate.
  - So communication during first d stages.
  - No communication in last r-d stages.

Efficiency on hypercube

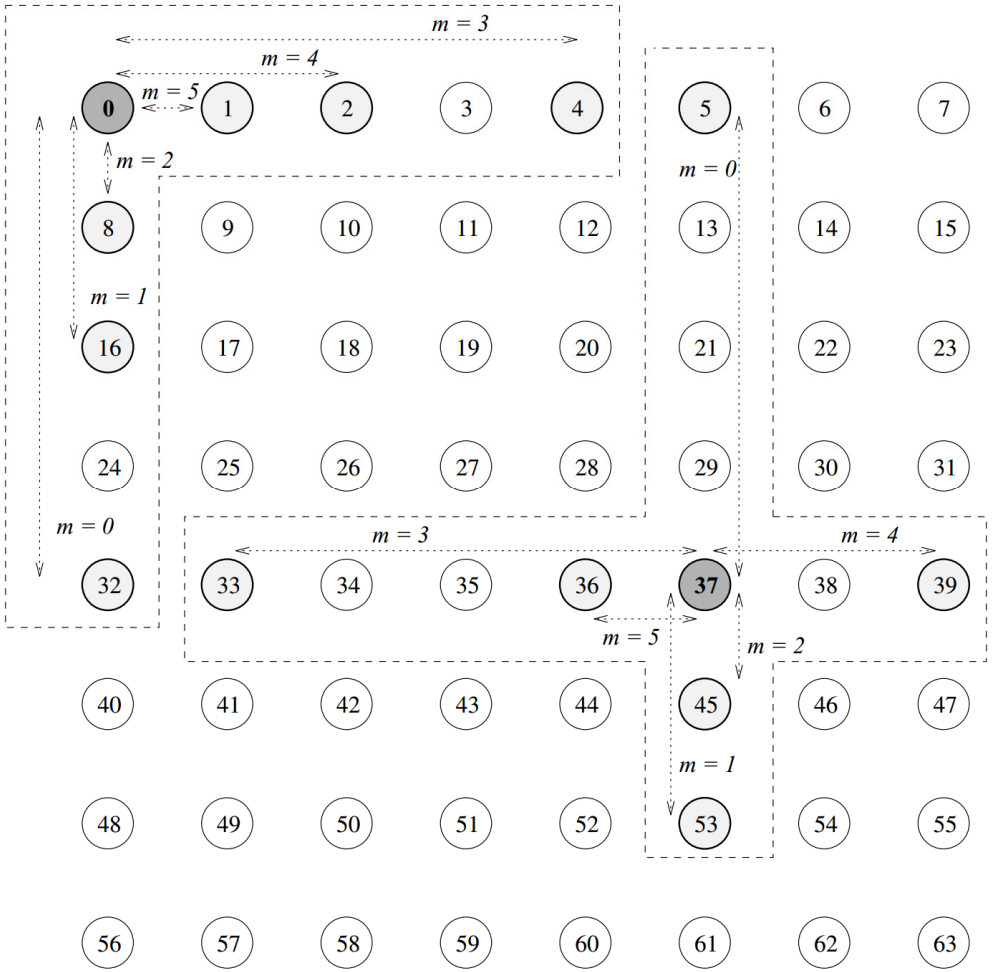
- Consider the binary exchange algorithm on a hypercube architecture.
- Each processor connected to d others, which differ in each digit of ID.
  - Communication only with neighbors, send n/p values each time.
  - Since d = log p rounds of communication, communication time $T_c = t_s \log p + t_w \frac{n}{p} \log p$.
- Each stage does n/p computation.
  - Total computation time $T_p = \frac{t_c n}{p} \log n$.
- Efficiency is $\frac{T_p}{T_p + T_c} = 1/(1 + \frac{t_s p \log p}{t_c n \log n} + \frac{t_w \log p}{t_c \log n})$.
- To maintain isoefficiency, want last two terms in denominator to be constant.
- $\frac{t_s p \log p}{t_c n \log n} = \frac{1}{K}$ implies $n \log n = W = K \frac{t_s}{t_c} p \log p$.
- $\frac{t_w \log p}{t_c \log n} = \frac{1}{K}$ implies $\log n = K \frac{t_w}{t_c} \log p$, so $n = p^{K t_w / t_c}$, so $W = K \frac{t_w}{t_c} p^{K t_w / t_c} \log p$.

Efficiency

- Isoefficiency depends on number of processors and machine parameters t_c, t_s, t_w.
- $W = \max\{p \log p,\ K\frac{t_s}{t_c} p \log p,\ K\ \frac{t_w}{t_c} p^{Kt_w/t_c}\ \log p\ \}$.
  - $K = \frac{E}{1-E}$, where E is efficiency.
  - Once $\frac{Kt_w}{t_c} > 1$, work size needed for isoefficiency grows exponentially.
- Ex Plot of work size for isoeffiency for $t_c = 2, t_w = 4, t_s = 25$.



Binary exchange FFT on mesh



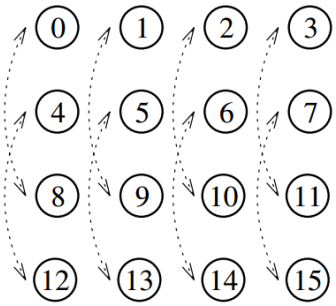
- Consider a $\sqrt{p} \times \sqrt{p}$ mesh.
  - Let d = log p.
- d rounds of communication.
  - In first d/2 rounds, processor communicates along its column, and in last d/2 rounds it communicates along its row.
- For $0 \leq m \leq \frac{d}{2} - 1$, processor communicates distance $2^{d/2-m-1}$ away along column. Similarly for rows.
- Since all processors along row or column do this in the same round, then the congestion on a column link is $2^{d/2-m-1}$ in round m.

Efficiency on mesh

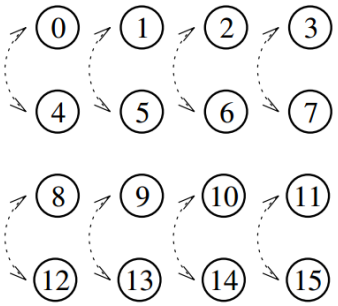
- Total computation time $T_p = t_c \frac{n}{p} \log n$.
- Total communication time $T_c = \sum_{m=0}^{\frac{d}{2}-1} (t_s + t_w \frac{n}{p} 2^m) \approx t_s \log p + 2t_w \frac{n}{\sqrt{p}}$.
- Efficiency $E = 1/(1 + \frac{t_s p \log p}{t_c n \log n} + \frac{2t_w \sqrt{p}}{t_c \log n})$.
- For isoefficiency, need both terms in denominator to be constant.
- Second term = $\frac{1}{K}$ implies $W = 2K \frac{t_w}{t_c} 2^{2K\sqrt{p}t_w/t_c} \sqrt{p}$.
- FFT is not scalable on mesh, due to its poor $O(\sqrt{p})$ bisection bandwidth.

2D transpose FFT

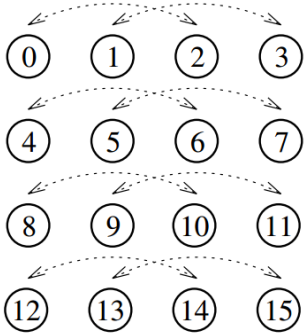
- Binary exchange FFT has isoefficiency function $O(p \log p)$ if $K \frac{t_w}{t_c} \leq 1$, but $O(p^{K t_w / t_c} \log p)$ isoefficiency for $K \frac{t_w}{t_c} > 1$.
  - So efficiency degrades when t_w / t_c large.
- 2D transpose FFT algorithm has $O(p^2 \log p)$ isoefficiency regardless of machine parameters.
- Uses a hypercube, or other architecture with bisection width $\Theta(p)$ for p processors.
- Assume $\sqrt{n}$ is a power of 2, and arrange n input values into a $\sqrt{n} \times \sqrt{n}$ grid.



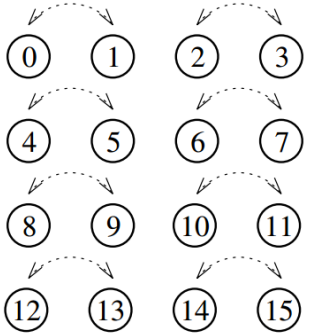
(a) Iteration m = 0



(b) Iteration m = 1



(c) Iteration m = 2

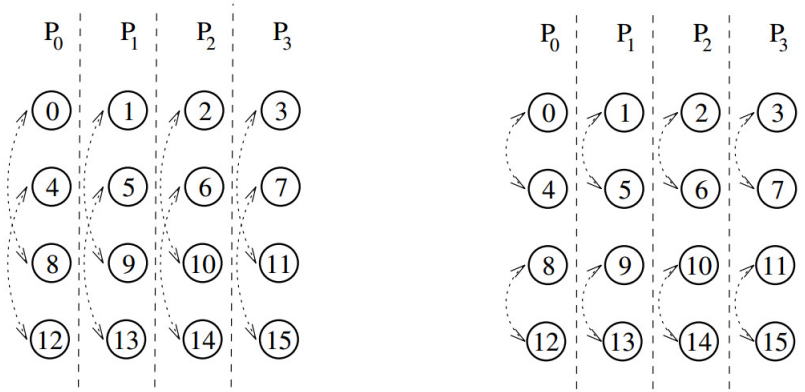


(d) Iteration m = 3

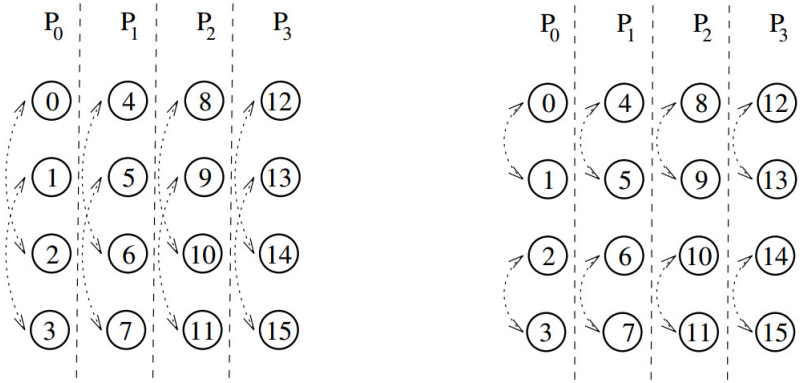
2D transpose FFT

- Assign each processor a column of values.
- For first (log n) / 2 rounds, no communication.
- After round (log n) / 2, transpose the values on all the processors (step b).
- Continue for (log n) / 2 more rounds, with no communication.
- The only communication is for the transpose.
- With p processors, give each $\sqrt{n}/p$ columns of values in striped format.

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64



(a) Steps in phase 1 of the transpose algorithm (before transpose)



(b) Steps in phase 3 of the transpose algorithm (after transpose)

Efficiency

- In each phase of computation, do $\sqrt{n}/p$ FFT's each of size $\sqrt{n}$, which takes time $t_c \frac{\sqrt{n}}{p} \sqrt{n} \log n$.
- The only communication step is for transpose, which takes $t_s(p-1) + t_w n/p$ time using all to all personalized communication, assuming $\Theta(p)$ bisection bandwidth.
- Efficiency is $1/(1 + \frac{t_s p^2}{t_c n \log n} + \frac{t_w}{t_c \log n})$.
- For isoefficiency, second term requires $n \log n = \Omega(p^2)$. Also, we have $\sqrt{n} = \Omega(p)$, since at most $\sqrt{n}$ processors can be used in the grid.
  - So we have $n = \Omega(p^2)$, and $W = n \log n = \Omega(p^2 \log p)$.