# TEMPUS: Activity 2

## Chapter 4

## SIGNAL AND IMAGE PROCESSING ALGORITHMS

István Rényi, KFKI-MSZKI

August 2, 1997

# Chapter 4 Table of Contents

# Chapter 4

# SIGNAL AND IMAGE PROCESSING ALGORITHMS

## 4.1 Introduction

Before engaging in data parallel special purpose array processor architecture and implementation, the properties and classifications of *algorithms* must be understood. This chapter stresses the basic background for algorithmic understanding.

An *algorithm* is a set of rules for solving a problem in a finite number of steps. There is extensive literature exploring the computational aspects of signal/image processing, and also considerable numbers of software packages are available.

Some basic algorithms used in signal and image processing are listed next:

(a) Matrix Operations
Matrix-vector multiplications, solution of linear systems, singular value decomposition, etc.

(b) Basic DSP Operations
Filtering: FIR, IIR, convolution, resampling, template matching, linear phase filtering, etc.
Transformations: Discrete Fourier, FFT, Walsh-Hadamard, Karhunen-Loeve, etc.

(c) Image Processing Algorithms
Restoration: inverse filtering, pseudo inverse filter by SVD
Reconstruction: Back projection, interpolation, pseudoinverse filter, algebraic reconstruction
Enhancement and smoothing: histogram transforms, noise elimination, median filter, edge- and line preserving smoothing, and so on.
Edge and line detection methods, template matching, Heuckel operator, Hough transform
Texture analysis: cooccurrence matrix, run-lengths matrix, autocorrelation, texture

edge detection, Fourier features, texture edge preserving smoothing, etc.
Geometrical operations: connected components, boundary detection/description, warping, distance transform, projections, cross section, thinning, etc.

(d) Other algorithms
Searching and sorting (merge sort, bitonic sort, etc.)
Graph and geometrical algorithms: transitive closure, connected components, minimum spanning tree
Polynomial operations: multiplication, division, encoding/decoding for error correction

Two important aspects of algorithmic study:
*application domains* and *computation counts*

Examples:

Application domains

| Application | Attractive Problem Formulation | Candidate Solutions |
|---|---|---|
| Hi-res direction finding | Symmetric eigensystem | SVD |
| State estimation | Kalman filter | Recursive least squares |
| Adaptive noise cancellation | Constrained Least sq's | Triangular or orthog. decomposition |

Computation counts

| Order | Name | Examples |
|---|---|---|
| $N$ | Scalar | Inner product, IIR filter |
| $N^2$ | Vector | Lin. transforms, Fourier transform, convolution, correlation, matrix-vector products |
| $N^3$ | Matrix | Matrix-matrix products, matrix decoposition, solution of eigensystems, least square problems |

Digital signal processing is characterized by large amounts of data, and tremendous computation requirements. The demand for speed and performance is continually growing. These requirements call for revolutionary *supercomputing technology*.

The algorithmic background provided by this chapter will be essential for the development of VLSI array processors.

## 4.2 Matrix Algorithms

Matrix operations are very common in digital signal- and image processing operations. An *m* x *n* matrix is a square array of *m* rows and *n* columns of data items (scalar, vector or matrix). An *n*-dimensional vector consists of *n* data items, which could be arranged into a row or column vector.

### *4.2.1 Basic Matrix Operations*

The most common matrix operations prevalent in signal processing are summarized below.

#### 4.2.1.1 Inner product

$$\mathbf{u}^T = [u_1, u_2, \ldots, u_n] \quad \text{and} \quad \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ . \\ . \\ . \\ v_n \end{bmatrix}$$

$$\langle \mathbf{u}, \mathbf{v} \rangle = u_1 v_1 + u_2 v_2 + \cdots + u_n v_n = \sum_{j=1}^{n} u_j v_j$$

#### 4.2.1.2 Outer product

$$\begin{bmatrix} u_1 \\ u_2 \\ . \\ . \\ . \\ u_n \end{bmatrix} \cdot [v_1 \quad v_2 \quad \ldots \quad v_m] = \begin{bmatrix} u_1 v_1 & u_1 v_2 & \cdots & u_1 v_m \\ u_2 v_1 & u_2 v_2 & \cdots & u_2 v_m \\ u_3 v_1 & u_3 v_2 & \cdots & u_3 v_m \\ . & . & & . \\ . & . & & . \\ . & . & & . \\ u_n v_1 & u_n v_2 & \cdots & u_n v_m \end{bmatrix}$$

#### 4.2.1.3 Matrix-Vector Multiplication

The *n* x *m* matrix **A** multiplied by an *m*-element column vector **u** results in an *n*-element column vector **v**.

$$\mathbf{v} = \mathbf{A}\,\mathbf{u}$$

The *i*th element of **v** is

$$v_i = \sum_{j=1}^{m} a_{ij} u_j$$

where $a_{ij}$ is the *i*th element of the *j*th column of **A**.

#### 4.2.1.4 Matrix Multiplication

**A** is of size *m* x *n*, that of **B** is *n* x *p*. The product, denoted by **C** is of size *m* x *p* and is given by **C = A B.** The *ij*th element of **C** has the following form:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

### *4.2.2 Solving Linear Systems*

To solve *n* simultaneous linear equations with *n* unknowns, is to find *n* x 1 vector **x**:

$$\mathbf{A\,x} = \mathbf{y}$$

$$\mathbf{x} = \mathbf{A}^{-1}\,\mathbf{y}$$

The number of computations for $\mathbf{A}^{-1}$ is high, the procedure unstable.
Triangularize **A** to get upper triangular matrix **A'**:

$$\mathbf{A}'\,\mathbf{x} = \mathbf{y}'$$

Where **A** is an upper triangular matrix. Back substitution provides solution **x.**

#### 4.2.2.1 Matrix Triangularization

Frequently used methods for matrix triangularization are:

- Gaussian elimination
- LU decomposition
- QR decomposition

In the following the QR method will be discussed briefly.

One way the QR decomposition is obtained is via an orthonormal transform, called Given's rotation (GR): A matrix **A** can be written as the product of a matrix with orthonormal columns and an invertible upper triangular matrix, i.e. **A = Q R**, where **Q** has orthonormal columns and **R** is an upper triangular matrix. The decomposition of **A** is obtained as a sequence of GRs. The GR operator performs plane rotations of **A**, this way annihilating its subdiagonal elements.

For an invertible **A**, the upper triangular matrix **R** is obtained as follows:

$$\mathbf{Q}^{\mathbf{T}}\,\mathbf{A} \;=\; \mathbf{R}\,\mathbf{A}^{-1}$$

$$\mathbf{Q}^{\mathbf{T}} = \;\mathbf{Q}_{(N\text{-}1)}\,\mathbf{Q}_{(N\text{-}2)}\;\cdots\mathbf{Q}_{(1)}$$

and

$$\mathbf{Q}_{(p)} = \;\mathbf{Q}^{(p,p)}\,\mathbf{Q}^{(p+1,p)}\ldots\mathbf{Q}^{(N\text{-}1,p)}$$

where $\mathbf{Q}(pq)$ is the GR operator to nullify matrix element at the ($q$+1)st row, and $p$th column, and has the following form:

$$\mathbf{Q}^{(q,p)} = \begin{array}{cc} \begin{array}{ccccccc} & & & \text{col. } p & \text{col. } p+1 & & \\ & & & \downarrow & \downarrow & & \end{array} & \\ \begin{bmatrix} 1 & 0 & \cdots & & & 0 & 0 \\ 0 & 1 & \cdots & & & 0 & 0 \\ 0 & 0 & \cdots & & & 0 & 0 \\ : & : & & \cos\phi & \sin\phi & & : \\ & & & -\sin\phi & \cos\phi & & \\ 0 & 0 & & & & & : \\ 0 & 0 & \cdots & & & 1 & 0 \\ 0 & 0 & \cdots & & & 0 & 1 \end{bmatrix} & \begin{array}{l} \\ \\ \\ \longleftarrow \text{row } q \\ \longleftarrow \text{row } q+1 \\ \\ \\ \\ \end{array} \end{array}$$

where $\Theta = \tan^{-1}\,[a_{q+1,\mathrm{p}}\ /\ a_{q,p}\,]$ is an abbreviation of $\Theta\,(p,q)$. The above operation of creating $\cos\Theta$ and $\sin\Theta$ is called Given's generation (GG).

The matrix product $\mathbf{A'} = \mathbf{Q}^{(q,p)}$ then becomes:

$$a'_{q,k} = a_{q,k}\cos\phi\,+a_{q+1,k}\sin\phi$$

$$a'_{q+1,k} = -\,a_{q,k}\sin\phi\,+a_{q+1,k}\cos\phi$$

$$a'_{jk} = a_{jk}\quad \text{if}\quad j \neq q,\, q+1$$

for all $k = 1, \ldots, N$.

#### 4.2.2.2 Back substitution

Find an $n$ x 1 vector $\mathbf{x}$ such that

$$\mathbf{A'}\cdot\mathbf{x} = \mathbf{y'}$$

where **A'** is an upper triangular matrix. The example to follow makes the operation easy to understand:

$$\begin{bmatrix} 1 & 1 & -1 \\ 0 & 3 & -2 \\ 0 & 0 & 1 \end{bmatrix}\cdot\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -2 \\ -9 \\ 3 \end{bmatrix} \qquad \begin{array}{rrrcr} x_1 & +x_2 & -x_3 & = & -2 \\ & 3x_2 & -2x_3 & = & -9 \\ & & x_3 & = & 3 \end{array}$$

$x_1$, $x_2$ and $x_3$ can be easily obtained from the equations on the right hand side.

### *4.2.3 Iterative Method*

When large, sparse matrices (e.g. $10^5 \times 10^5$ ) are involved, the iterative method may be more efficient:

$$\mathbf{g} = \mathbf{H}\,\mathbf{f}$$

**g** represents physical measurements, **H** is the sparse matrix.

By splitting the **A** matrix into **S** and **T** such that

$$\mathbf{A} = \mathbf{S} + \mathbf{T}$$

Then starting with an initial guess $\mathbf{x}_0$, the iteration:

$$\mathbf{S}\,\mathbf{x}_{k+1} = -\mathbf{T}\mathbf{x}_k + \mathbf{y}$$

Sequence of vectors $\mathbf{x}_{k+1}$ are expected to converge to **x**.

### *4.2.4. Eigenvalue and Singular Value Decomposition*

Let **A** be of size *n* x *n*. If there exists a vector **e** such that

$$\mathbf{A}\,\mathbf{e} = \lambda\,\mathbf{e}$$

$\lambda$ is called eigenvalue, **e** is eigenvector. $\lambda$ is obtained by solving the characteristic equation

$$|\mathbf{A} - \lambda\,\mathrm{I}\,| = 0$$

For distinct eigenvalues:

$$\mathbf{A}\,\mathbf{E} = \mathbf{E}\,\Lambda \quad = [\mathbf{e_1} \quad \mathbf{e}_2 \quad \cdots \quad \mathbf{e}_n]\cdot\begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}$$

The matrix **E** is invertible, and hence $\mathbf{A} = \mathbf{E}\,\Lambda\,\mathbf{E}^{-1}$ If the matrix **A** is an *n* x *n* normal matrix, (i.e. $\mathbf{A}^{\mathbf{H}}\,\mathbf{A} = \mathbf{A}\,\mathrm{A}^{\mathrm{H}}$), it can be factored

$$\mathbf{A} = \mathbf{U}\,\Lambda\,\mathbf{U}^{\mathrm{T}}$$

where **U** is an *n* x *n unitary* matrix and is a diagonal matrix that has all the eigenvalues of **A** in its diagonal. This decomposition is very common in control and image processing. The Karhunen-Loeve or KL transform is based on the decomposition of the covariance matrix.

Another important decomposition of matrices is the *Singular Value Decomposition* or SVD. It is of considerable interest in digital image processing, e.g. in image coding, enhancement and reconstruction.

It can be shown that any *n* x *m* matrix **A** can be factored into

$$\mathbf{A} = \mathbf{Q_1}\,\Sigma\,\mathbf{Q_2}^{\mathrm{T}}$$

where $\mathbf{Q_1}$ : $m \times m$ unitary matrix

$\mathbf{Q}_2$ : $n \times n$ unitary matrix, and

$\Sigma$ has the special form:

$$\Sigma = \begin{bmatrix} \mathbf{D} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}$$

where $\mathbf{D} = \operatorname{diag}(\sigma_1, \sigma_2, \ldots, \sigma_r)$, $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$, $r$ is the rank of **A**.

The SVD formulation can also be written as:

$$\mathbf{A} = \mathbf{Q}_1 \, \Sigma \, \mathbf{Q}_2^{\mathrm{T}} = \Sigma^{r}_{i=1} \sigma_i \, u_i \, v_i^{\mathrm{T}}$$

where $u_i$ is the column vector of matrix $\mathbf{Q}_1$, and $v_i$ is the column vector of matrix $\mathbf{Q}_2$.

The singular values of **A**: $\sigma_1, \sigma_2, \ldots, \sigma_r$ are square roots of the eigenvalues of

$$\mathbf{A}^{\mathbf{T}} \mathbf{A} \quad (\text{or } \mathbf{A}\,\mathbf{A}^{\mathbf{T}})$$

The column vectors of $\mathbf{Q}_1$, $\mathbf{Q}_2$ are the singular vectors of **A**, and are eigenvectors of

$$\mathbf{A}^{\mathbf{T}} \mathbf{A} \text{ and } \mathbf{A}\,\mathbf{A}^{\mathbf{T}} \text{ respectively.}$$

SVD is also used to

— solve the least squares problem

— determine the rank of a matrix

— find good low-rank approximation to the original matrix.

### *4.2.5 Solving the Least Squares Problem*

The well known linear least squares technique is useful in control, communication, signal processing. The technique is applied to

— equalization

— spectral analysis

— adaptive arrays

— digital speech processing

Problem formulation:

Given **A** , an $n \times p$ ( $n > p$, rank = $p$ ) *observation* matrix, and

**y**, an $n$-element desired data vector.

Find **w**, a $p$-element weight vector, which minimizes the Euclidean norm of the residual vector, **e**, defined by

$$\mathbf{e} = \mathbf{y} - \mathbf{A}\,\mathbf{w}$$

**Unconstrained least squares algorithm**

The Euclidean norm of a vector remains unchanged if it is premultiplied by an orthonormal matrix. This way orthonormal matrices can be used to reduce **A** to triangular form, by using a sequence of GR matrices to premultiply the residual vector **e** resulting in the transformed vector:

$$\mathbf{Q}\,\mathbf{e} = \mathbf{Q}\,\mathbf{y} - [\mathbf{Q}\,\mathbf{A}] = \mathbf{y'} - \mathbf{A'}\,\mathbf{w}$$

where Q is an $n$ x $n$ orthonormal matrix, and **A’** is an $n$ x $p$ matrix, i.e. **A’** is reduced to an upper triangular matrix, represented as

$$\mathbf{A}' = \begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix}$$

To minimize the Euclidean norm of **y’ - A’ w**, $\mathbf{w}_{\mathrm{opt}}$ is easily obtained, since **w** has no influence on the lower parts of the difference. Therefore

$$\mathbf{R}\ \mathbf{w}_{\mathrm{opt}} = \mathbf{y}'$$

$\mathbf{w}_{\mathrm{opt}}$ is obtained by back-substitution (**R** is an upper triangular matrix).

## 4.3 Digital Signal Processing Algorithms

In this section the basic signal processing operations, such as convolution, correlation and digital transforms will be introduced.

### *4.3.2 Discrete Time Systems and the Z-transform*

Discrete time signals may arise naturally or are sampled versions of continuous time signals.

Linear Time Invariant (LTI) systems are characterized by *h(n)*, the response to the sampling sequence, $\delta(n)$, where

$$\delta(n) = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{otherwise} \end{cases}$$

Here, *h(n)* is called the unit sample response or impulse response of the system. The output, *y(n)* of an LTI system is given by

$$y(n) = \sum_{k=-\infty}^{\infty} x(k)h(n-k) = x(n) * h(n)$$

This is the *convolution* operation. LTI systems are of two types: finite impulse response (FIR) and infinite impulse response (IIR), depending on whether *h(n)* has finite or infinite duration.

In addition to the time domain representation, systems can also be represented in the transform domain. Analogous to the Laplace transform for continuous signals, the *Z*-transform is used in connection with discrete time signals.

The definition of the Z-transform is the following:

$$X(z) = Z[x(n)] = \sum_{n=-\infty}^{\infty} x(n) z^{-n}$$

where $z$ is a complex number in a region of the $z$-plane.

**Example: *Z*-transform**

Let

$$x(n) = (1/2)^n \cdot s(n),$$

where *s(n)* is the unit sample function defined as

$$s(n) = \{1 \text{ if } n \geq 0, \text{ and } 0 \text{ otherwise}$$

Then

$$\begin{aligned} X(z) &= Z[x(n)] \\ &= \sum_{n=0}^{\infty} (1/2)^n z^{-n} \\ &= 1 / (1 - 0.5z^{-1}) = z / (z - 0.5), \quad \text{for } |z| > 0.5 \end{aligned}$$

Useful properties of the *Z*-transform that will be used later:

(i) $x(n) * h(n) \leftrightarrow X(z) \cdot H(z)$

(ii) $x(n + n_0) \leftrightarrow z^{n_0} \cdot X(z)$

Note also that the Fourier transform $X(j\omega)$ of the discrete sequence *x(n)* can be obtained by evaluating the Z-transform on the unit circle

$$X(j\omega) = X(z)|_{z = \exp(j\omega)}$$

The convolution operation reduces to multiplication in the Z-domain. Taking the Z-transform of y(n), it can be shown that

$$Y(z) = H(z) X(z)$$

where *H(z)* and *X(z)* are *Z*-transforms of *h(n)* and *x(n)* respectively.

### *4.3.2 Convolution*

The convolution of two discrete sequences *u(n)* and *w(n)* of finite lengths *N* is defined as

$$y(n) = \sum_{k=-\infty}^{\infty} u(k) w(n-k) = u(n) * w(n)$$

computation:

$$y(n) = \sum_{k=0}^{N-1} u(k) w(n-k)$$

where $n = 0, 1, 2, \ldots, 2N\text{-}2$

In digital filtering *u(n)* denotes the input sequence and *w(n)* represents the impulse response function of the filter. These two sequences are usually of different lengths. *y(n)* represents the processed (filtered) signal.

An efficient way of computation is first to transform the sequences into an other domain where the operations are simpler, and then to apply the inverse transform.

Using the FFT transform method, the order of computation is reduced from $O(N^2)$ to $O(N \log N)$.

**Recursive equations** – for convolution of two casual sequences:

$$y_j^k = y_j^{k\text{-}1} + u_k \, w_{j-k}$$

$k = 0, 1, ..., j$ when $j = 0, 1, ..., N$-1, and

$k = j - N + 1,\ j - N + 2, ..., N - 1$, when $j = N, N + 1, ...,2N$-2

### 4.3.3 Correlation

Correlation, another useful operation, often arises in connection with random processes and has a similar formula to convolution:

$$y(n) = \sum_{k=-\infty}^{\infty} u(k)w(n+k)$$

computation:

$$y(n) = \sum_{k=0}^{N-1} u(k)w(n+k)$$

where $n = -N + 1, -N + 2, \ldots, -1, 0, 1, \ldots, N - 2, N - 1$.

### 4.3.4 Digital FIR and IIR Filters

Digital filters are LTI systems designed to meet specific design requirements. They are often specified by a transfer function defined on the unit circle:

$$H(e^{j\omega}) = |H(e^{j\omega})| \, e^{j\theta(\omega)}$$

where $|H(e^{j\omega})|$ is the magnitude, $\theta(\omega)$ is the phase response.

Two kinds of digital filters are:

— Finite Impulse Response (FIR)
— Infinite Impulse Response (IIR)

Representation of digital filters, in a linear, time invariant discrete system: $p$th order difference equation:

$$y(n) = \sum_{k=1}^{p} a_k y(n-k) + \sum_{k=0}^{q} b_k x(n-k)$$

$x(n)$ . . . input signal

$y(n)$ . . . output signal

Taking the $Z$-transform and applying the shift property yields:

$$Y(z) = \sum_{k=1}^{p} a_k z^{-k} Y(z) + \sum_{k=0}^{q} b_k z^{-k} X(z)$$

where $Y(z)$ and $X(z)$ are Z - transforms of $\{y(n)\}$ and $\{x(n)\}$ resp.

In simpler form: $Y(z) = X(z) \cdot H(z),$ where the transfer function is given by:

$$H(z) = \frac{B(z)}{A(z)}$$

with $$B(z) = \sum_{k=0}^{q} b_k z^{-k}$$

and $$A(z) = 1 - \sum_{k=1}^{p} a_k z^{-k}$$

There are three classes of basic digital filters:

(1) **Moving average (MA) filter**

A special case of the general filter, where in the definition of *H(z)*, *A(z)* = 1. Here, each output point is a weighted average of the input data. This filter is an FIR filter and the output sequence is the convolution of the input sequence and the impulse response of the filter.

(2) **Autoregressive (AR) filter**

A special case of the general filter, where in the definition of *H(z)*, *B(z)* = 1. Here the output *y(n)* is generated as a linear regression of its past values, and hence the AR filter is an IIR filter.

(3) **Autoregressive moving average (ARMA) filter**

A more general filter, which is a composite of MA and AR filters. The ARMA filter is an IIR filter.

### *4.3.5 Linear Phase Filter*

In many applications it is desirable that the filter has a linear phase, i.e. $\Theta(\omega) = \alpha\, \omega.$ This way, signals in the passband are exactly reproduced at the output, except for a time delay, corresponding to the slope of the phase. The impulse response function *h(n)* has the property:

$$h(n) = h(N - 1 - n), \quad n = 0, 1, \ldots, N - 1.$$

Taking advantage of the symmetry, the realization only requires half number of multiplications. If *N* is odd, then

$$H(z) = \sum_{n=0}^{N-1} h(n)z^{-n} = \sum_{n=0}^{(N-1)/2} h(n)z^{-n} + \sum_{n=(N+1)/2}^{N-1} h(n)z^{-1}$$

let $n' = (N-1) - n$

$$H(z) = \sum_{n=0}^{(N-1)/2} h(n)z^{-n} + \sum_{n'=0}^{(N-1)/2} h(n')z^{-(N-1-n')} =$$

$$= \sum_{n=0}^{(N-1)/2} h(n)\left[z^{-n} + z^{-(N-1-n)}\right]$$

A realization of this filter will be shown later.

### 4.3.6 Discrete Fourier Transform (DFT)

The DFT of a finite lengths sequence {*x(n)*} is defined as

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot W_N^{nk}$$

where $k = 0, 1, 2, \ldots, N - 1$ and $W_N = e^{-j2\pi/N}$.

The DFT is efficiently computed using the Fast Fourier Transform (FFT). It has the following two properties:

(1) The DFT is obtained by uniformly sampling the Fourier transform of a sequence at the following $N$ points:

$$\omega = 0, \frac{2\pi}{n}, 2\cdot\frac{2\pi}{n}, \cdots, (N-1)\cdot\frac{2\pi}{n}$$

The inverse DFT is defined as:

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k)W_N^{-nk}, \quad n = 0,1,\ldots, N-1$$

(2) The multiplication of two $N$-point DFTs is equivalent to the circular convolution of the two sequences in the time domain (or space domain). Mathematically, if

$X_1(k)$ = DFT of $[x_1(n)]$

$X_2(k)$ = DFT of $[x_2(n)]$

Then $X_3(k) = X_1(k)X_2(k)$, is the DFT of $[x_3(n)]$, where

$$x_3(n) = \sum_{m=0}^{N-1} \tilde{x}_1(m)\tilde{x}_2(n-m)$$

and $n = 0, 1, \ldots, N - 1$.

### *4.3.7 Fast Fourier Transform (FFT)*

We will first examine the computational complexity of DFT(direct method):

each $x(n)W^{nk}$ requires 1 complex multiplication

$X(k)$ $\{k = 0, 1, ..., N-1\}$ requires $N^2$ complex multiplications + $N(N$-1) additions, roughly $N^2$ operations.

DFT computational complexity using FFT ($N = 2^m$ case):

Utilizing symmetry + periodicity of $W^{nk}$, operation count is reduced from $N^2$ to $N \log_2 N$.

Take the following example. If one complex multiplication takes 0.5 μsec:

| $N$ | $T_{DFT}$ | $T_{FFT}$ |
|---|---|---|
| $2^{12}$ | 8 sec. | 0.013 sec. |
| $2^{16}$ | 0.6 hour | 0.26 sec. |
| $2^{20}$ | 6 days | 5 sec |

There are two kinds of FFT algorithms, the *decimation in time* (DIT) FFT and the *decimation in frequency* (DIF) FFT, both with the same complexity of computation. Only the DIT FFT is discussed here, the DIF FFT is left as a problem.

**DIT FFT**

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk} = \sum_{n=even}^{N-1} x(n) W_N^{nk} + \sum_{n=odd}^{N-1} x(n) W_N^{nk}$$

substituting $n = 2r$ for even, $n = 2r+1$ for odd $n$,

$$X(k) = \sum_{r=0}^{N/2-1} x(2r) W_N^{2rk} + \sum_{r=0}^{N/2-1} x(2r+1) W_N^{(2r+1)k}$$

$$= \sum_{r=0}^{N/2} (W_N^2)^{rk} + W_N^k \sum_{r=0}^{N/2-1} x(2r+1)(W_N^2)^{rk}$$

since

$$W_N^2 = e^{-2j(2\pi/N)} = e^{-2j\pi/(N/2)} = W_{N/2}$$

$$X(k) = \sum_{r=0}^{N/2-1} x(2r) W_{N/2}^{rk} + W_N^k \sum_{r=0}^{N/2-1} x(2r+1) W_{N/2}^{rk} =$$

$$= G(k) + W_N^k H(k)$$

In the last equation $G(k)$ and $H(k)$ can be obtained via an $N/2$-point FFT. Thus an $N$-point FFT is achieved by using two $N/2$-point FFTs and combining the results. By recursively applying this decomposition, we may use a number of 2-point FFTs and combine their results. An 8-point FFT computed this way is shown in Figure 1. below:

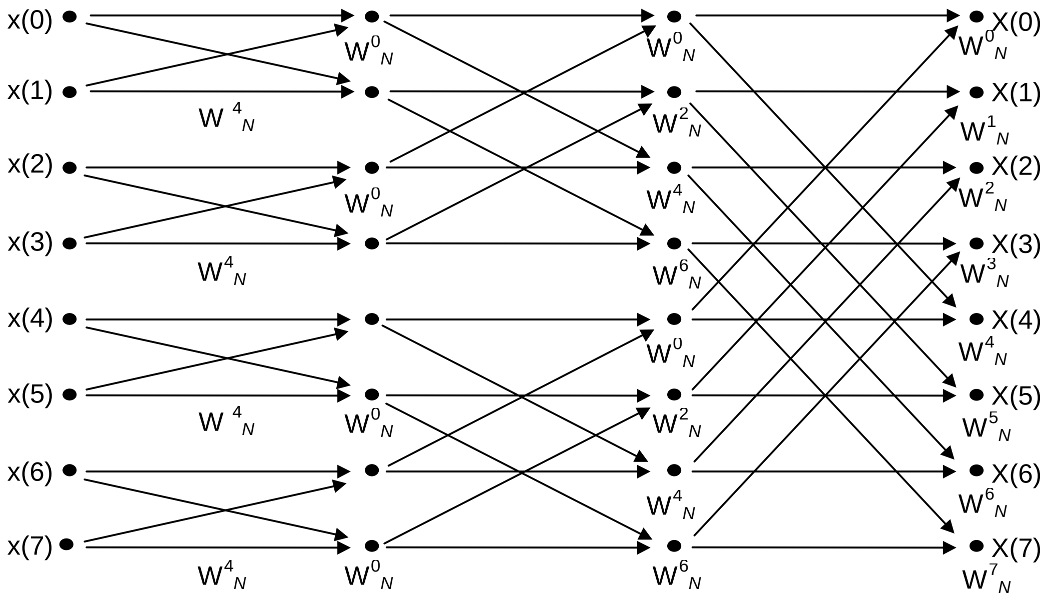


*Figure 1. DIT FFT* ($N$ = 8)

As it is obvious from the above figure, the FFT computation consists of a sequence of "*butterfly*" operations, each consisting of one addition, one subtraction and one multiplication.

The FFT algorithm has very versatile applications. The procedure of applying the transformation to linear convolution is outlined below

(1) Append zeros to the two sequences of lengths $N$ and $M$, to make them of lengths, an integer power of two, that is larger than or equal to $M+N$-1.

(2) Apply FFT to both zero appended sequences

(3) Multiply the two transformed domain sequences

(4) Apply inverse FFT to the new multiplied sequence

### *4.3.8 Discrete Walsh-Hadamard Transform (WHT)*

The basic functions are binary valued with { +1, -1 }. Therefore the generation and implementation is simple, the operations consisting of only additions and subtractions.

The Hadamard matrix has orthogonal rows and columns, and is a square array of +1s and -1s. It is defined iteratively, as

$$\mathbf{H}_2 = \frac{1}{\sqrt{2}} \cdot \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad \text{and} \quad \mathbf{H}_{2N} = \frac{1}{\sqrt{2}} \cdot \begin{bmatrix} \mathbf{H}_N & \mathbf{H}_N \\ \mathbf{H}_N & -\mathbf{H}_N \end{bmatrix}$$

A Hadamard matrix of size eight is shown below

$$\mathbf{H_8} = \frac{1}{2\sqrt{2}} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix}$$

If we have an input data vector **x** of lengths $N$ ($N$=$2^n$), the transformed data becomes $\mathbf{y} = \mathbf{H}_N\,\mathbf{x}$.

The WHT has been applied to a number of image and speech processing operations, to pattern recognition, spectral analysis of linear systems, correlation and convolution, filtering, data compression, coding, detection, spectroscopy, etc. The WHT, just like other transforms, has a fast algorithm.

### 4.3.9 Solving Toeplitz System (Schur Algorithm)

We attempt to determine **x** when

$$\mathbf{R\,x} = \mathbf{y},$$

and **R** has Toeplitz structure, i.e. $r(i,j) = r(|i-j|)$.

This system of equations can be solved using the procedures on solving general linear systems. However, they don't exploit the Toeplitz structure. The well established serial algorithm to solve Toeplitz system is the Levinson algorithm, which - because it requires inner product operations - is not well suited to parallel implementation. The Schur algorithm avoids inner product computations, and is more suitable for parallel processing.

The major function in solving the Toeplitz system is to perform triangular decomposition of the matrix **R**, i.e.

$$\mathbf{R = U}^{\boldsymbol{T}}\mathbf{\,D\,U = U}^{\boldsymbol{T}}\mathbf{\,U}\ \ \mathbf{(U = D\,U)}$$

where **D** is a diagonal matrix and **U** is an upper triangular matrix. Then the solution **x** becomes:

$$\mathbf{x = R^{-1}\,y = U^{-1}\,(U^{\boldsymbol{T}})^{-1}\,y}$$

which can be separated into a forward substitution step: $\mathbf{x = U^{-1}\,g}$.

## 4.4. Image Processing Algorithms

Most of the one-dimensional signal processing algorithms presented so far can be naturally expanded into two- or multidimensional processing applications. Some of them are discussed next.

### 4.4.1 Two-dimensional Convolution and Correlation

Image data are two-dimensional (2-D), with two space indices. Convolution and correlation will also be 2-D. The formulation of the 2-D convolution is as follows:

$$y(n_1,n_2) = \sum_{k_1=0}^{n_1} \sum_{k_2=0}^{n_2} u(k_1,k_2)\, w(n_1-k_1, n_2-k_2)$$

where $n_1$ , $n_2 \in \{ 0, 1, ..., 2N\text{-}2 \}$

The 2-D correlation formula reads:

$$y(n_1,n_2) = \sum_{k_1=0}^{n_1} \sum_{k_2=0}^{n_2} u(k_1,k_2)\, w(n_1+k_1, n_2+k_2)$$

where $n_1$ , $n_2 \in \{ -N+1, -N+2, ..., -1,0,1, ..., 2N\text{-}2 \}$

The number of computations needed for these operations is usually very large. For this sake transform methods are usually used for efficient computation. The method is similar to the 1-D case.

### 4.4.2 Two-dimensional Filtering

The 2-D filter operation can be represented by any one of two methods

- 2-D difference equation in the space domain
- transfer function in the frequency domain

Computation of a 2-D filter can be done by successively using 1-D filtering. In general, the computation involves

- Fast 2D convolution, via 2-D FFT
- 2D difference equation directly
- Occasionally, successive 1-D filtering

### 4.4.3 2-D DFT, FFT, and Hadamard Transform

The two-dimensional DFT is similar to 1D case, and is defined as follows:

$$X(k_1,k_2) = \sum_{n_1=0}^{N-1} \sum_{n_2=0}^{N-1} x(n_1,n_2) \cdot W_N^{n_1k_1+n_2k_2}$$

$$k_1,k_2 \in \{0,1,2,...,N-1\} \quad \text{and} \quad W_N = e^{-j2\pi/N}$$

Two important properties of the 2-D DFT:

(1) Using 1-D FFT $2N$ times, the 2-D DFT can be calculated in $O(2N^2 \log_2 N)$ time. This is done by first applying 1-D FFT row-wise $N$ times and then 1-D FFT column-wise $N$ times on the transformed sequence.

(2) The 2-D counterpart of the 1-D convolution can be fast computed using transform methods, e.g. by using 2-D FFT, multiplying the transforms and applying inverse 2-D FFT.

The 2-D Hadamard transform is defined in a similar way.

## 4.5 Advanced Algorithms and Applications

The classes of algorithm we are going to discuss next are worthy of further exploration, since they share two desirable features: (1) they belong to an important application domain, and (2) they are potentially suitable for VLSI array processors.

### 4.5.1 Divide-and-Conquer Techniques

The idea of the divide-and-conquer techniques is to decompose the large size problem into smaller ones and combine the solution for the whole. Since the subproblems are formulated like smaller versions of the original one, the *same routine can be used in different partitions* of the problem. This way, the divide-and-conquer approach is a *top down* approach, that can be used *recursively* and yields efficient solutions (see Figure 2.).

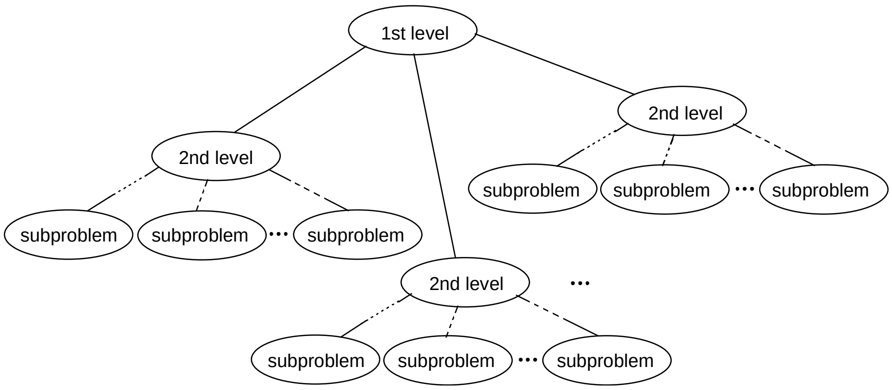


*Figure 2. Schematic, Divide-and-Conquer techniques*

As an example, consider finding the maximum element of a set $S$ containing $n$ elements, where $n$ is a power of two. We first divide $S$ into two subsets $S_1$ and $S_2$, each consisting of $n/2$ elements. The algorithm would find max{ $S_1$ } and max{ $S_2$ }, and the maximum of $S$ could be calculated as the larger of max{ $S_1$ } and max{ $S_2$ }. The efficient solution is realized by recursive application of the algorithm.

An important research topic on array architectures is the design of *interconnection networks* for certain classes of divide-and-conquer algorithms. Prominent examples are sorting and FFT.

### 4.5.2 Dynamic Programming Model

Dynamic programming techniques are extensively used in *optimization* problems to minimize or maximize an objective function, subject to a set of constraints. In essence, it is a *bottom up* procedure, in which the solutions to all subproblems are calculated first and the results used to solve the whole problem. The results of a stage are later used to solve the problems of the stage above. The intermediate results are stored and utilized in later stages.

As opposed to divide-and-conquer techniques, here there is only one subproblem to solve in any one stage. The solutions to these subproblems are linked by the *recurrence relation*.

This recursive formulation is important in *mapping* algorithms to arrays with *local interconnect*. Examples are: Finding the shortest path problem in a graph, and the minimum cost path finding problem., which finds applications in digital speech processing in compensating the variation of speaking rates.

### 4.5.3 Relaxation Techniques

A very promising algorithm class from the array processing perspective is that, based on the relaxation techniques. This in an *iterative approach*, which makes updating at each point in parallel. Each iteration uses data from most recent updating (in most cases neighboring data elements). Computationally more powerful than the one-shot methods, because here the initial choices are successively refined, based on newly available information. The relaxation techniques are very *suitable for array processors*, because they are order independent and can be speeded up by parallel processing. This is because updating at each data point at each iteration is *executed in parallel*.

Examples are: image reconstruction, restoration from blurring, and solving partial differential equations.

### 4.5.4 Simulated Annealing via Stochastic Relaxation

For certain optimization problems The computation complexity may be too large to solve exactly. To these problems *iterative improvement* methods can be successfully applied. One disadvantage of this method is that the solution may only be locally and not globally optimal. This deficiency is common to many optimization problems, where once a state is in local optimum, it may be trapped there forever.

Simulated annealing is a search technique that allows the possibility of getting out of the locally optimal trap by using a trap ‘flattening’ approach, based on stochastic decision of temporarily accepting a *worse* solution.

Suppose that an energy function, $E$ is introduced on a finite set of states $S$. The task is to find a state $s^*$ , that would minimize $E(s)$. For each state $s \in S$, a transition probability $R(s,s')$ is introduced as the probability of making a state transition from s to $s'$. Now a stochastic scheme can be described for constructing a sequence of states $s_0$, $s_1$, $s_2$, . . ., (the initial state is set to be $s_0$). Transition from $s_k$ to $s_k'$ is chosen with probability distribution

$$P[s_k' \mid s_k\ ] = R(s_k, s_k')$$

By setting

$$s_{k+1} = \begin{cases} s_k', & \text{with probability } P_k \\ s_k, & \text{otherwise} \end{cases}$$

where

$$P_k = \exp\left\{ \frac{-\text{Max}\,[\Delta E, 0]}{T_k} \right\}$$

where $\Delta E = E(s_k') - E(s_k)$, and $\{ T_k \}$ is the control parameter (called "temperature" for flattening the trap.

This process of slow cooling is analogous to slow annealing of metal, in order to crystallize it in its lowest energy state. Convergence can be guarantied if the system has certain properties and a suitable temperature sequence is defined.

At the beginning of the process, probability of moving out a local optimum region is high. During the iteration process (when the temperature parameter is lowered) the system will arrive at a  global minimum, and the probability of getting out of it is low.

Examples are: Image restoration and reconstruction, optimization, certain aspects of artificial intelligence, and code design for communication systems.

### 4.5.5 Associative Retrieval

Retrieving data by association is a powerful solution to many high-volume information processing problems. Such a system is very adept at *recognition* and *recall from partial information*. A popular associative processing model is based on the *content addressable memory* (CAM), which represents the simplest property of a simple neuron system. The CAM-type memory is also closely related to how the human brain functions.

Data stored in an associative memory are addressed by their contents and, in this sense associative memory and CAM can be considered to be equivalent. A CAM can perform *parallel search* and *comparison*, as opposed to RAM. These operations are frequently needed in storage and retrieval of rapidly changing database, i.e. radar signal tracking, image processing, computer vision, etc.

**Hopfield networks** The original Hopfield model uses two-state threshold neurons that follow a stochastic algorithm. Each neuron $i$ has two states, characterized by the output $V_i$ of the neuron, with values: $V_i^0$ and $V_i^1$. The input of each neuron comes from two sources: (1) external input $I_i$ , and (2) states of fellow neurons. Hopfield defined an *energy function* for symmetrically connected networks, which, for any state is given by

$$E = -\sum_{i<j} T_{ij} V_i V_j - \sum_i I_i V_i$$

where $T_{ij}$ can be biologically viewed as a description of the synaptic interconnection strength from neuron $i$ to $j$.

The difference energy function between two different levels of neuron $i$, given the state of the other neuron is

$$E_{i-on} - E_{i-off} = \Delta E_i = -\sum_j T_{ij} V_j - I_i$$

If the energy difference $\Delta E_i$ is negative, the unit should turn (stay) on, otherwise it should turn (stay) off.

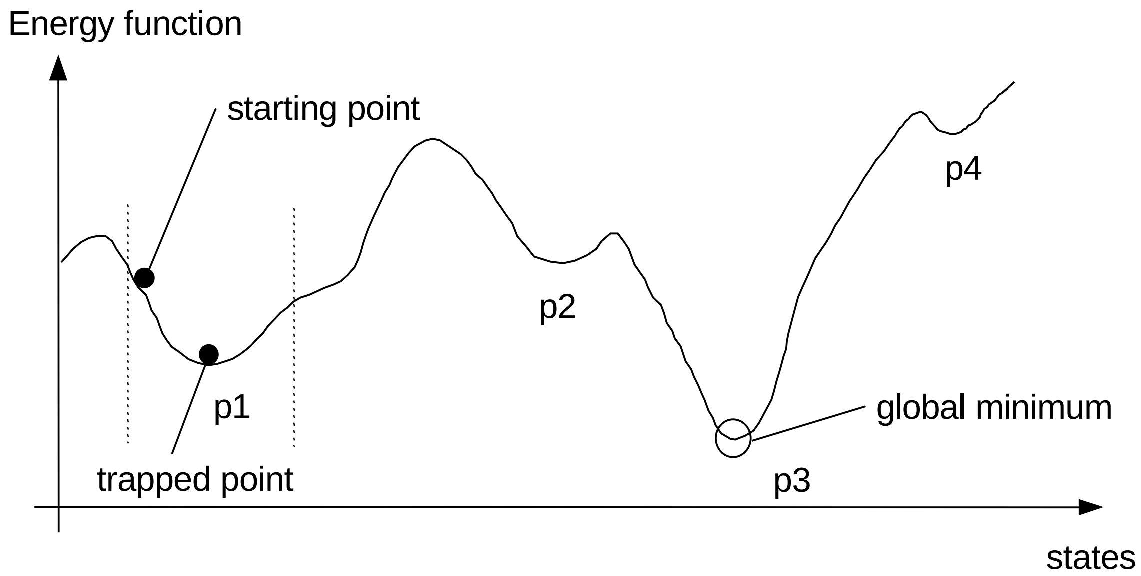


*Figure 3.  The Hopfield model behaves as an associative memory. The local minimum (p1, p2, p3, and p4) correspond to stored target patterns*

The model also behaves like an associative memory. With reference to the Figure 3., each local minimum corresponds to a stored *target pattern*. The memory is content addressable, because if it is started anywhere close to a particular stable state (local energy minimum) and far from all others, it then would converge to that state. In the given state space, if the position of a stable point is considered the information of a specific memory, then the states in close proximity include partial information about that memory. A final stable state containing all information can be found from a consideration of an initial state of partial information concerning a memory. By supplying some subpart of the memory in the initial state, the memory can be reached, as opposed to using an address.

## 4.6 VLSI Array Algorithms

An *array algorithm* is a set of rules for solving a problem in a finite number of steps by a multiple number of *interconnected* processors. Now we will focus on the interconnection issue.

*Concurrency* is important to achieve high throughput using VLSI array technology. In general, concurrency is achieved by decomposing the problem into independent subtasks executable in *parallel*, or into dependent subtasks executable in a *pipelined* fashion.

The most crucial issue regarding efficiency is *communication*, i.e. a scheme of moving data among processing elements (PEs). VLSI technology constrains *recursive* and *locally dependent* algorithms. [In a recursive algorithm all processors perform nearly identical tasks, and each processor repeatedly executes a fixed set of subtasks on sequentially available data.]

An effective *algorithm design* must start with a full understanding of the problem specification, and the mathematical / algorithmic analysis. The dependence graph provides an effective tool, since it exhibits the full dependencies incurred during the execution of the algorithm. New algorithmic design methodologies may exploit potential concurrency available in array processor environment

### 4.6.1 Algorithm Design Criteria for VLSI Array Processors

The effectiveness of mapping an algorithm onto a processor array heavily depends on the way the algorithm is decomposed. Two dissimilar algorithms with equivalent performance in a sequential computer may perform very differently in an array processor environment.

On sequential machines complexity depends on computation count and storage requirement. Because a large number of PEs are involved, the overhead in an array processor environment critically depends on the availability of processors, and is therefore not uniform. Computation count is no longer an effective measure of processor performance.

#### 4.6.1.1 Area-Time Complexity Theory

Complexity theory attempts to provide systematic information about algorithmic complexity to allow an educated choice to be made between two algorithms. For VLSI computation models area-time complexity measures have received special attention. They depend on two factors: on computation time ($T$) and on chip area ($A$). In particular, complexity measure is computed as $AT^2$. This measure will not be emphasized here, because is not a recognized and good design criteria, it offers little practical implication in VLSI system design. Practically any cost effectiveness measure $f(A,T)$, a function of speed and chip area depends strongly on the particular application. In any event, little relationship has been established between the special measure $AT^2$ (used mainly in the academic community) and a practical measure $f(A,T)$, which always can be tailored to special needs.

#### 4.6.1.2 Design Criteria for VLSI Array Algorithms

It is clear that a new criteria are needed to determine algorithm efficiency to include

- stringent *communication problems*, associated with VLSI technology
- *communication costs*
- additional influential factors, like parallelism and *pipelining rate*

Generally speaking, in designing array algorithms for array processing, the design criteria should comprise computation, communication, memory and I/O. Their key aspects are:

(1) **Maximum parallelism.** The favorable algorithm expresses higher degree of parallelism, which is *exploitable* by the computing arrays.

(2) **Maximum pipelineability.** The exploitation of pipelining is essential for regular and locally connected networks. The best among all algorithms and arrays must be selected to maximize throughput rate. Unpredictable data dependency may jeopardize efficiency. Iterative methods, dynamic, data-dependent branching are less well suited to pipelined architectures. Well-structured algorithms with dependable data movements are required.

(3) **Balance among computations, communications and memory.** A good array algorithm should offer a good balance between different bandwidths incurred in different communication hierarchies to avoid bottlenecks and data draining. Balancing of com-

putations and communications is critical to the effectiveness of array computing. Pipelining is suitable for balancing computations and I/O.

(4) **Trade-off between computation and communication.** For a practical and efficient interconnection network regular communication should be encouraged. Key issues affecting communication regularity include: local / global, static / dynamic, data dependent / data independent interconnection modules. The trade-off between interconnection cost and throughput should be maximized. Recently a lot of emphasis has been placed on local and recursive algorithms.

(5) **Numerical performance, quantization effects.** Numerical behavior depends on many factors, including word lengths and algorithms used. To improve precision, additional computation may be necessary. This is a heavily 'problem dependent' issue, no general rule could be given. A prominent example is that of the FFT algorithm, which is computationally cost effective and at the same time numerically well behaved.

### *4.6.2 Locally Versus Globally Recursive Algorithms*

The basic question now is, how do signal- and image processing applications dictate the design of array processors, and how these algorithms can be implemented in VLSI hardware.

Common features of signal / image processing algorithms:

- intensive computation
- matrix operations
- localized or perfect-shuffle operations

These all point to promising systematic design methods for array architectures.

For proper communication, in an interconnected network each PE should know *when, where* and *how* to send / fetch data.

— **where?** In locally recursive algorithms, data movements are confined to nearest neighbor PEs. Therefore locally interconnected network will suffice to execute the algorithm at high performance.

— **when?** The conventional approach to send data is to use a globally synchronous scheme. Here, timing is controlled by a sequence of 'beats' (see systolic array).

— **how?** To the question of how to communicate, it depends very much on the interconnection network adopted in the system.

#### 4.6.2.1 Local and Global Communications in Algorithms

Concurrent processing performance critically depends on the *communication cost* for data transactions. It is therefore necessary to formally characterize the communication requirement and to evaluate the costs incurred.

Each PE will be assigned a *location index*, and their distance will be defined as the difference of their location indices. The communication cost is characterized by the *distance* between PEs involved in the data transaction.

To fully describe the recursive activities, *time index, spatial index* are also needed to show when and where computation takes place. The two major classes of recursive algorithms with local or global interconnections are:

— **Local type** recursive algorithm: index separations are within a certain limit. The majority of signal processing algorithms shares this locally data-dependent property. (e.g. matrix multiplication, convolution, etc.)

— **Global type** recursive algorithm: recursion involves separated space indices. Calls for globally interconnected structures (e.g. FFT and sorting)

#### 4.6.2.2 Locally Recursive Algorithms

Majority of algorithms used in signal and image processing share the common traits of localized operations, intensive computation (as compared with I/O) and matrix operations. When these algorithms are mapped onto an array structure only *local communications* are required. This feature can be exploited to simplify architectural and software requirements.

Another subject will be entirely devoted to the locally recursive algorithms. In the meantime a short digression is made to explore the class of globally recursive algorithms.

#### 4.6.2.3 Globally Recursive Algorithms: An FFT example

A typical example for this class of globally recursive algorithms is the FFT. The perfect shuffle in FFT inherently requires global communication to support data shuffling among stages of computation. The basic array configurations for FFT computation are shown in Figure 4. Note, that in the FFT, the total number of butterfly operations is $(N/2)\log_2 N$. For each butterfly four real multiplications and four real additions needed. In the single state configuration a linear array of $N/2$ PEs and $\log_2 N$ time units are needed to complete an FFT of $N$ points. In this example we use the decimation in time FFT algorithm.

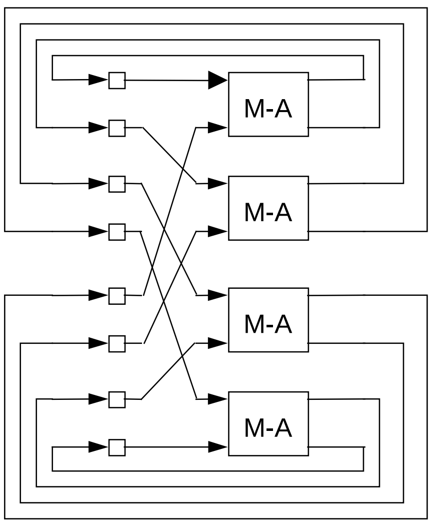


*Figure 4a. Array configuration for FFT computation*
*Single-stage array*

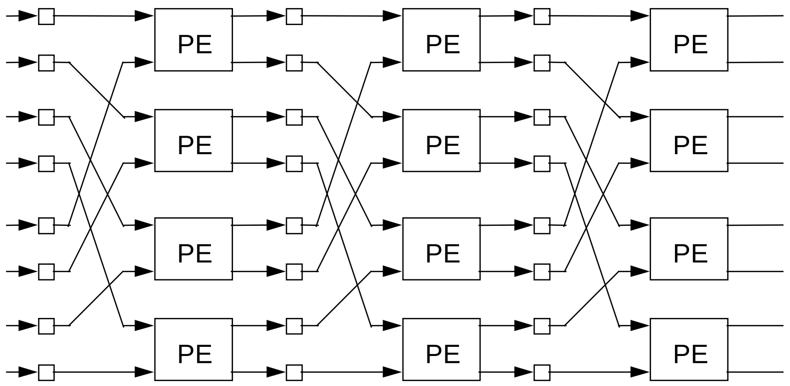


*Figure 4b. Array configuration for FFT computation*
*Multi-stage array*

**Perfect Shuffle Permutation** The permutation here corresponds to a single bit left shift of the binary representation of index *x:*

$$x = \{ b_{n},, b_{n\text{-}1}, ..., b_1 \}$$

$$\sigma(x) = \{b_{n\text{-}1}, b_{n\text{-}2},, ..., b_1, b_n \}$$

**Exchange permutation** The exchange permutation can be defined as follows:

$$\varepsilon_{(k)}(x) = \{ b_n, ..., b_k', ..., b_1 \}$$

where $b_k'$ denotes the complement of the *k*th bit. Figure 4. compares perfect shuffle permutation and exchange permutation networks.

**FFT via Shuffle-Exchange Network** Interconnection network for in-place computation has to provide

- exchange permutation ($\varepsilon_{(k)}$)
- bit-reversal permutation ($\rho$)

For an 8-point DIT FFT the interconnection network can be represented as follows

$$\rho[\varepsilon_{(1)}[\varepsilon_{(2)}[\varepsilon_{(3)}]]]$$

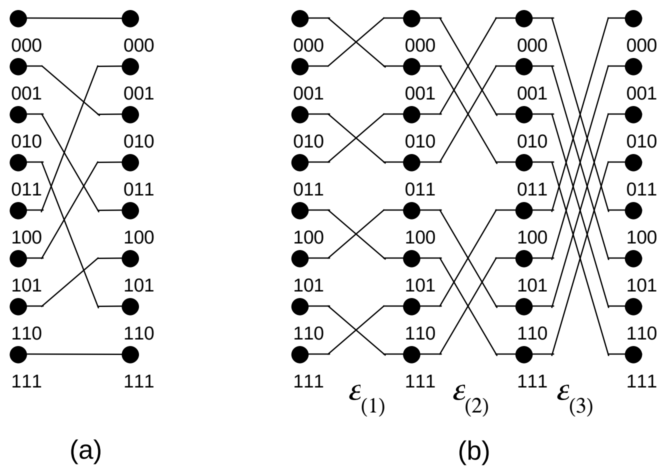


*Figure 4. (a) Perfect shuffle permutation. (b) Exchange permutation*

The sequence of operations is to apply $\varepsilon_{(3)}$ first, $\varepsilon_{(2)}$ next, etc. We can consider computing $X(k)$ by separating $x(k)$ into even and odd $N/2$-point sequences.

$$X(k) = \sum_{n=0}^{7} x(n) W_N^{nk}$$

where k = 0, 1, . . . , 7. Both $n$ and $k$ are represented by 3-bit binary numbers:

$$n = (n_3\, n_2 n_1) = 4n_3 + 2n_2 + n_1$$

$$k = (k_3\, k_2\, k_1) = 4k_3 + 2k_2 + k_1$$

The result of the operation is shown in the following table:

| Original Index | | Bit-reversed Index | |
|---|---|---|---|
| $x(0)$ | 000 | $x(0)$ | 000 |
| $x(1)$ | 001 | $x(4)$ | 100 |
| $x(2)$ | 010 | $x(2)$ | 010 |
| $x(3)$ | 011 | $x(6)$ | 110 |
| $x(4)$ | 100 | $x(1)$ | 001 |
| $x(5)$ | 101 | $x(5)$ | 101 |
| $x(6)$ | 110 | $x(3)$ | 011 |
| $x(7)$ | 111 | $x(7)$ | 111 |

Due to in-place replacement (i.e. input and output data share storage):

$n_1$ is replaced by $k_3$ ,

$n_3$ is replaced by $k_1$ , etc.

$x(n_3\, n_2\, n_1)$ is stored in the array position $X(k_1\, k_2\, k_3)$

In other words, to determine the position of $x(n_3\, n_2\, n_1)$ in the input, the bits of index $n$ have to be reversed.