Chapter 11 Image Processing

Low-level Image Processing

Operates directly on a stored image to improve or enhance it. Stored image consists of a two-dimensional array of pixels (picture elements):

Many low-level image-processing operations assume monochrome images and refer to pixels as having gray level values or intensities.
Computational Requirements

Suppose a pixmap has $1024 \times 1024$ pixels and 8-bit pixels.

Storage requirement is $2^{20}$ bytes (1 Mbytes)

Suppose each pixel must be operated upon just once.

Then $2^{20}$ operations are needed in the time of one frame.

At $10^{-8}$ second/operation (10ns/operation), this would take 10 ms.

In real-time applications, the speed of computation must be at the frame rate (typically 60–85 frames/second).

All pixels in the image must be processed in the time of one frame; that is, in 12–16 ms.

Typically, many high-complexity operations must be performed, not just one operation.

Point Processing

Operations that produce an output based upon the value of a single pixel.

Thresholding

Pixels with values above a predetermined threshold value are kept and others below the threshold are reduced to 0. Given a pixel, $x_i$, the operation on each pixel is

$$\text{if } (x_i < \text{threshold}) \quad x_i = 0; \quad \text{else } x_i = 1;$$

Contrast Stretching

Range of gray level values extended to make details more visible. Given pixel of value $x_i$ within range $x_l$ and $x_h$, the contrast stretched to the range $x_H$ to $x_L$ by multiplying $x_i$ by

$$x_i = (x_i - x_l) \left( \frac{x_H - x_L}{x_h - x_l} \right) + x_L$$

Gray Level Reduction

Number of bits used to represent the gray level reduced. Simple method would be to truncate the lesser significant bits.
Histogram

Shows the number of pixels in the image at each gray level:

 Sequential code

for(i = 0; i < height_max; x++)
for(j = 0; j < width_max; y++)
    hist[p[i][j]] = hist[p[i][j]] + 1;

where the pixels are contained in the array p[] and hist[k] will hold the number of pixels having the kth gray level.

Similar to adding numbers to an accumulating sum and similar parallel solutions can be used for computing histograms.
Smoothing, Sharpening, and Noise Reduction

*Smoothing* suppresses large fluctuations in intensity over the image area and can be achieved by reducing the high-frequency content.

*Sharpening* accentuates the transitions, enhancing the detail, and can be achieved by two ways.

*Noise reduction* suppresses a noise signal present in the image.

Often requires a local operation with access to a group of pixels around the pixel to be updated. A common group size is $3 \times 3$:

$$
\begin{array}{ccc}
  x_0 & x_1 & x_2 \\
  x_3 & x_4 & x_5 \\
  x_6 & x_7 & x_8 \\
\end{array}
$$

Figure 11.3  Pixel values for a $3 \times 3$ group.
**Mean**

A simple smoothing technique is to take the *mean* or *average* of a group of pixels as the new value of the central pixel.

Given a $3 \times 3$ group, the computation is

$$x_4' = \frac{x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8}{9}$$

where $x_4'$ is the new value for $x_4$.

**Sequential Code**

Nine steps to compute the average for each pixel, or $9n$ for $n$ pixels.

A sequential time complexity of $O(n)$.

**Parallel Code**

Number of steps can be reduced by separating the computation into four data transfer steps in lock-step fashion.

![Parallel Code Diagram](image)

**Figure 11.4** Four-step data transfer for the computation of mean.
Figure 11.5 Parallel mean data accumulation.
Median

Sequential Code

Median can be found by ordering the pixel values from smallest to largest and choosing the center pixel value (assuming an odd number of pixels).

With a $3 \times 3$ group, suppose the values in ascending order are $y_0$, $y_1$, $y_2$, $y_3$, $y_4$, $y_5$, $y_6$, $y_7$, and $y_8$. The median is $y_4$.

Suggests that all the values in the group must first be sorted, and then the fifth element taken to replace the original value of the pixel.

Using bubble sort, in which the lesser values are found first in order, sorting could, in fact, be terminated after the fifth lowest value is obtained.

Number of steps for finding each median is given by $8 + 7 + 6 + 5 + 4 = 30$ steps, or $30n$ for $n$ pixels.

Parallel Code

An Approximate Sorting Algorithm

First, a compare-and-exchange operation performed on each of the rows, requiring three steps. For the $i$th row, we have

$$p_{i,j-1} \leftrightarrow p_{i,j}$$
$$p_{i,j} \leftrightarrow p_{i,j+1}$$
$$p_{i,j-1} \leftrightarrow p_{i,j}$$

where $\leftrightarrow$ means “compare and exchange if left gray level greater than right gray level”.

Then the columns sorted using three compare-and-exchange operations:

$$p_{i-1,j} \leftrightarrow p_{i,j}$$
$$p_{i,j} \leftrightarrow p_{i+1,j}$$
$$p_{i-1,j} \leftrightarrow p_{i,j}$$

The value in $p_{i,j}$ is taken to be the fifth largest pixel value. Does not always select fifth largest value. Reasonable approximation. Only requires six steps.
Weighted Masks

The mean method could be described by a weighted $3 \times 3$ mask.

Suppose the weights are $w_0, w_1, w_2, w_3, w_4, w_5, w_6, w_7,$ and $w_8,$ and pixel values are $x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7,$ and $x_8.$

The new center pixel value, $x_4',$ is given by

$$x_4' = \frac{w_0 x_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + w_5 x_5 + w_6 x_6 + w_7 x_7 + w_8 x_8}{k}$$

The scale factor, $1/k,$ is set to maintain the correct grayscale balance in the image after the operation. Often, $k$ is given by $w_0 + w_1 + w_2 + w_3 + w_4 + w_5 + w_6 + w_7.$

The summation of products, $w_i x_i,$ from two functions $w$ and $x$ is the (discrete) cross-correlation of $f$ with $w$ (written as $f \otimes w$).
Figure 11.7 Using a 3 × 3 weighted mask.

Figure 11.8 Mask to compute mean.
Figure 11.9  A noise reduction mask.

$k = \frac{1}{16}  
\begin{array}{ccc}
1 & 1 & 1 \\
1 & 8 & 1 \\
1 & 1 & 1 \\
\end{array}

Figure 11.10  High-pass sharpening filter mask.

$k = \frac{1}{9}  
\begin{array}{ccc}
-1 & -1 & -1 \\
-1 & 8 & -1 \\
-1 & -1 & -1 \\
\end{array}

The computation done with this mask is

$$x_4' = \frac{8x_4 - x_0 - x_1 - x_2 - x_3 - x_5 - x_6 - x_7 - x_8}{9}$$
Edge Detection

Highlighting edges of object where an edge is a significant change in gray level intensity.

Gradient and Magnitude

Let us first consider a one-dimension gray level function, $f(x)$ (say along a row).

If the function, $f(x)$, were differentiated, the first derivative, $\frac{\partial f}{\partial x}$, measures the gradient of the transition and would be a positive-going or negative-going spike at a transition.

The direction of the transition (either increasing or decreasing) can be identified by the polarity of the first derivative.

Figure 11.11  Edge detection using differentiation.
Image Function

A two-dimensional discretized gray level function, $f(x,y)$.

Gradient (magnitude)

$$\nabla f = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$

Gradient Direction

$$\phi(x, y) = \tan^{-1}\left(\frac{\frac{\partial f}{\partial y}}{\frac{\partial f}{\partial x}}\right)$$

where $\phi$ is the angle with respect to the $y$-axis. The gradient can be approximated to

$$\nabla f \approx \frac{\partial f}{\partial y} + \frac{\partial f}{\partial x}$$

for reduced computational effort.

Figure 11.12 Gray level gradient and direction.
Edge Detection Masks

For discrete functions, the derivatives are approximated by differences.

The term $\frac{\partial f}{\partial x}$ is the difference in the $x$-direction and $\frac{\partial f}{\partial y}$ is the difference in the $y$-direction.

We might consider computing the approximate gradient using pixel values $x_5$ and $x_3$ (to get $\frac{\partial f}{\partial x}$) and $x_7$ and $x_1$ (to get $\frac{\partial f}{\partial y}$); i.e.,

$$\frac{\partial f}{\partial x} = x_5 - x_3$$

$$\frac{\partial f}{\partial y} = x_7 - x_1$$

so that

$$\nabla f \approx |x_7 - x_1| + |x_5 - x_3|$$

Two masks are needed, one to obtain $x_7 - x_1$ and one to obtain $x_5 - x_3$.

The absolute values of the results of each mask are added together.

Prewitt Operator

The approximate gradient obtained from

$$\frac{\partial f}{\partial y} \approx (x_6 - x_0) + (x_7 - x_1) + (x_8 - x_2)$$

$$\frac{\partial f}{\partial x} \approx (x_2 - x_0) + (x_5 - x_3) + (x_8 - x_6)$$

Then

$$\nabla f \approx |x_6 - x_0 + x_7 - x_1 + x_8 - x_2| + |x_2 - x_0 + x_5 - x_3 + x_8 - x_6|$$

which requires using the two $3 \times 3$ masks.
Derivatives are approximated to
\[ \frac{\partial f}{\partial y} \approx (x_6 + 2x_7 + x_8) - (x_0 + 2x_1 + x_2) \]
\[ \frac{\partial f}{\partial x} \approx (x_2 + 2x_5 + x_8) - (x_0 + 2x_3 + x_6) \]

Operators implementing first derivatives will tend to enhance noise.

However, the Sobel operator also has a smoothing action.
Figure 11.14 Sobel operator.

<table>
<thead>
<tr>
<th>-1</th>
<th>-2</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>-1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

(a) Original image (Annabel) (b) Effect of Sobel operator

Figure 11.15 Edge detection with Sobel operator.
Laplace Operator

The Laplace second-order derivative is defined as

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

approximated to

$$\nabla^2 f = 4x_4 - (x_1 + x_3 + x_5 + x_7)$$

which can be obtained with the single mask:

<table>
<thead>
<tr>
<th></th>
<th>-1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 11.16 Laplace operator.

Figure 11.17 Pixels used in Laplace operator.
The Hough Transform

Purpose is to find the parameters of equations of lines that most likely fit sets of pixels in an image.

A line is described by the equation

\[ y = ax + b \]

where the parameters, \( a \) and \( b \), uniquely describe the particular line, \( a \) the slope and \( b \) the intercept on the \( y \)-axis.

A search for those lines with the most pixels mapped onto them would be computationally prohibitively expensive [an \( O(n^3) \) algorithm].

Suppose the equation of the line is rearranged as:

\[ b = -xa + y \]

Every point that lies on a specific line in the \( x-y \) space will map into same point in the \( a-b \) space (parameter space).

Therefore, we can find the number of points on each possible line in the \( x-y \) space by simply counting when a mapping maps into a point in the \( a-b \) space.
Finding the Most Likely Lines

A single point \((x_1, y_1)\) can be mapped into the points of a line

\[ b = -x_1a + y_1 \]

in the \(a-b\) space.

In the mapping process, discrete values will be used to a coarse prescribed precision and the computation is rounded to the nearest possible \(a-b\) coordinates.

The mapping process is done for every point in the \(x-y\) space.

A record is kept of those \(a-b\) points that have been obtained by incrementing the corresponding accumulator.

Each accumulator will have the number of pixels that map into a single point in the parameter space.

The points in the parameter space with locally maximum numbers of pixels are chosen as lines.
Method will fail for vertical lines and with lines that approach this extreme.

To avoid the problem, line equation rearranged to polar coordinates:

\[ r = x \cos \theta + y \sin \theta \]

where \( r \) is the perpendicular distance to the origin in the original \((x, y)\) coordinate system and \( \theta \) is the angle between \( r \) and the \( x \)-axis.

![Diagram](image)

Figure 11.20  Mapping a line into \((r, \theta)\) space.

The \( \theta \) value will be in degrees and very conveniently will also be the gradient angle of the line (with respect to the \( x \)-axis).

Each line will map to a single point in the \((r, \theta)\) space.

A vertical line will simply map into a point where \( \theta = 0^\circ \) if a positive intercept with the \( x \)-axis, or \( \theta = 180^\circ \) if a negative intercept with the \( x \)-axis.

A horizontal line has \( \theta = 90^\circ \).

Each point in the \( x \)-\( y \) space will map into a curve in the \((r, \theta)\) space; i.e., \((x_1, y_1)\) maps into \( r = x_1 \cos \theta + y_1 \sin \theta \).
Implementation

Assume origin at the top left corner.

![Diagram](image1.png)

Figure 11.21 Normal representation using image coordinate system.

The parameter space is divided into small rectangular regions. One accumulator is provided for each region.

Accumulators of those regions that a pixel maps into are incremented. This process must be done for all the pixels in the image.

If all values of $\theta$ were tried (i.e., incrementing $\theta$ through all its values), the computational effort would be given by the number of discrete values of $\theta$, say $k$ intervals. With $n$ pixels the complexity is $O(kn)$.

The computational effort can be reduced significantly by limiting the range of lines for individual pixels using some criteria. A single value of $\theta$ could be selected based upon the gradient of the line.
Sequential Code

```
for (x = 0; x < xmax; x++)  /* for each pixel */
    for (y = 0; y < ymax; y++) {
        sobel(&x, &y, dx, dy);  /* find x and y gradients */
        magnitude = grad_mag(dx, dy); /* find magnitude if needed */
        if (magnitude > threshold) {
            theta = grad_dir(dx, dy); /* atan2() fn */
            theta = theta_quantize(theta);
            r = x * cos(theta) + y * sin(theta);
            r = r_quantize(r);
            acc[r][theta]++;  /* increment accumulator */
            append(r, theta, x, y);  /* append point to line */
        }
    }
```

Finally, when all pixels have been considered, the values held in each accumulator will give the number of pixels that can map onto the corresponding line. The most likely lines are those with the most pixels, and accumulators with the local maxima are chosen. An algorithm for choosing the local maxima must be devised.
Parallel Code

Since the computation for each accumulator is independent of the other accumulations, it could be performed simultaneously, although each requires read access to the whole image. - Left as an exercise.

Transformation into the Frequency Domain

Fourier Transform

Many applications for the Fourier transform in science and engineering.

In the area of image processing, the Fourier transform is used in image enhancement, restoration, and compression.

The image is a two-dimensional discretized function, \( f(x, y) \), but first start with the one-dimensional case.

For completeness, let us first review the results of the Fourier series and Fourier transform concepts from first principles.
Fourier Series

The Fourier series is a summation of sine and cosine terms and can be written as

\[ x(t) = \frac{a_0}{2} + \sum_{j=1}^{\infty} \left( a_j \cos \left( \frac{2\pi ft}{T} \right) + b_j \sin \left( \frac{2\pi ft}{T} \right) \right) \]

where \( T \) is the period (1/T = f, where f is a frequency).

By some mathematical manipulation, we can get the more convenient description of the series:

\[ x(t) = \sum_{j=-\infty}^{\infty} X_j e^{2\pi if t / T} \]

where \( X_j \) is the \( j \)th Fourier coefficient in a complex form and \( i = \sqrt{-1} \).

The Fourier coefficients can also be computed from specific integrals.

Fourier Transform

Continuous Functions

The previous summation developed into an integral:

\[ x(t) = \int_{-\infty}^{\infty} X(f) e^{2\pi if t} df \]

where \( X(f) \) is a continuous function of frequency.

The function \( X(f) \) can be obtained from

\[ X(f) = \int_{-\infty}^{\infty} x(t) e^{-2\pi if t} dt \]

\( X(f) \) is the spectrum of \( x(t) \), or more simply the Fourier transform of \( x(t) \).

The original function, \( x(t) \), can obtained from \( X(f) \) using the first integral given, which is the inverse Fourier transform.
Discrete Functions

For functions having a set of \( N \) discrete values by replacing the integral with a summation, leading to the *discrete Fourier transform* (DFT) given by

\[
X_k = \frac{1}{N} \sum_{j=0}^{N-1} x_j e^{-2\pi i \frac{jk}{N}}
\]

and *inverse discrete Fourier transform* given by

\[
x_k = \sum_{j=0}^{N-1} X_j e^{2\pi i \frac{jk}{N}}
\]

for \( 0 \leq k \leq N - 1 \). The \( N \) (real) input values, \( x_0, x_1, x_2, \ldots, x_{N-1} \), produce \( N \) (complex) transform values, \( X_0, X_1, X_2, \ldots, X_{N-1} \).

Fourier Transforms in Image Processing

A two-dimensional Fourier transform is

\[
X_{lm} = \sum_{j=0}^{N-1} \sum_{k=0}^{M-1} x_{jk} e^{-2\pi i \left( \frac{jl}{N} + \frac{km}{M} \right)}
\]

where \( j \) and \( k \) are row and column coordinates, \( 0 \leq j \leq N - 1 \) and \( 0 \leq k \leq M - 1 \). Assume image is square, where \( N = M \). Equation can be rearranged into

\[
X_{lm} = \sum_{j=0}^{N-1} \left[ \sum_{k=0}^{N-1} x_{jk} e^{-2\pi i \frac{km}{N}} \right] e^{-2\pi i \frac{jl}{N}}
\]

Inner summation is a one-dimensional DFT operating on \( N \) points of a row to produce a transformed row. Outer summation is a one-dimensional DFT operating on \( N \) points of a column. We might write

\[
X_{lm} = \sum_{j=0}^{N-1} X_{jm} e^{-2\pi i \frac{jl}{N}}
\]

Hence, the two-dimensional DFT can be divided into two sequential phases, one operating on rows of elements and one operating on columns of (transformed) elements.
Applications

Frequency filtering can be described by the convolution operation:

\[ h(j, k) = g(j, k) * f(j, k) \]

the same operation as the cross-correlation operation for symmetrical masks, where \( g(j, k) \) describes the weighted mask (filter) and \( f(j, k) \) describes the image.

It can be shown that the Fourier transform of a product of functions is given by the convolution of the transforms of the individual functions.

Hence, the convolution of two functions can be obtained by taking the Fourier transforms of each function, multiplying the transforms

\[ H(j, k) = G(j, k) \times F(j, k) \]

(element by element multiplication), where \( F(j, k) \) is the Fourier transform of \( f(j, k) \) and \( G(j, k) \) is the Fourier transform of \( g(j, k) \), and then taking the inverse transform to return the result into the original spatial domain.
Parallelizing the Discrete Fourier Transform

Starting from

\[ X_k = \sum_{j=0}^{N-1} x_j e^{-2\pi i \frac{jk}{N}} \]

and using the notation \( w = e^{-2\pi i/N} \),

\[ X_k = \sum_{j=0}^{N-1} x_j w^{jk} \]

The \( w \) terms are called the twiddle factors. Each input value has to be multiplied by a twiddle factor.

The inverse transform can be obtained by replacing \( w \) with \( w^{-1} \).
Sequential Code

```c
for (k = 0; k < N; k++) {  /* for every point */
    X[k] = 0;
    for (j = 0; j < N; j++)  /* compute summation */
        X[k] = X[k] + w^j * k * x[j];
}
```

where \( X[k] \) is the \( k \)th transformed point, \( x[k] \) is the \( k \)th input and \( w = e^{-2\pi/N} \). Summation step requires complex number arithmetic.

The code can be rewritten as

```c
for (k = 0; k < N; k++) {
    X[k] = 0;
    a = 1;
    for (j = 0; j < N; j++) {
        X[k] = X[k] + a * x[j];
        a = a * w^k;
    }
}
```

where \( a \) is a temporary variable.

Elementary Master-Slave Implementation

One slave process of \( N \) slave processes could be assigned to produce one of the transformed values; i.e., the \( k \)th slave process produces \( X[k] \). Parallel time complexity with \( N \) (slave) processes is \( O(N) \).

![Master-slave approach for implementing the DFT directly.](image-url)
Pipeline Implementation

Unfolding the inner loop for \( x[k] \), we have

\[
\begin{align*}
x[k] &= 0; \\
a &= 1; \\
x[k] &= x[k] + a \ast x[0]; \\
a &= a \ast w^k; \\
x[k] &= x[k] + a \ast x[1]; \\
a &= a \ast w^k; \\
x[k] &= x[k] + a \ast x[2]; \\
a &= a \ast w^k; \\
x[k] &= x[k] + a \ast x[3]; \\
a &= a \ast w^k; \\
\ldots \\
\end{align*}
\]

Each pair of statements

\[
\begin{align*}
x[k] &= x[k] + a \ast x[0]; \\
a &= a \ast w^k;
\end{align*}
\]

could be performed by a separate pipeline stage.

Figure 11.26 One stage of a pipeline implementation of DFT algorithm.
Figure 11.27  Discrete Fourier transform with a pipeline.
DFT as a Matrix-Vector Product

The $k$th element of the discrete Fourier transform is given by

$$X_k = x_0w^0 + x_1w^1 + x_2w^2 + x_3w^3 + \ldots + x_{N-1}w^{N-1}$$

and the whole transform can be described by a matrix-vector product:

$$\begin{bmatrix}
X_0 \\
X_1 \\
X_2 \\
\vdots \\
X_k \\
\vdots \\
X_{N-1}
\end{bmatrix} = \frac{1}{N} \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\
1 & w & w^2 & w^3 & \ldots & w^{N-1} \\
1 & w^2 & w^4 & w^6 & \ldots & w^{2(N-1)} \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
1 & w^k & w^{2k} & w^{3k} & \ldots & w^{(N-1)k} \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
1 & w^{N-1} & w^{2(N-1)} & w^{3(N-1)} & \ldots & w^{(N-1)(N-1)}
\end{bmatrix} \begin{bmatrix} x_0 \\
x_1 \\
x_2 \\
\vdots \\
x_k \\
\vdots \\
x_{N-1}
\end{bmatrix}$$

(Note $w^0 = 1$.) Hence, the parallel methods of producing a matrix-vector product as described in Chapter 10 can be used for the discrete Fourier transform.

Fast Fourier Transform

Method of obtaining discrete Fourier transform with a time complexity of $O(N \log N)$ instead of $O(N^2)$.

Let us start with the discrete Fourier transform equation:

$$X_k = \frac{1}{N} \sum_{j=0}^{N-1} x_j w^{jk}$$

where $w = e^{-2\pi i/N}$. 

---

Slides for Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen © Prentice Hall, 1999. All rights reserved.
Many formulations. The version that starts by dividing the summation into the following two parts:

\[ X_k = \frac{1}{N} \left[ \sum_{j=0}^{(N/2)-1} x_{2j} w^{2jk} + \sum_{j=0}^{(N/2)-1} x_{2j+1} w^{(2j+1)k} \right] \]

where the first summation processes the \( x \)'s with even indices and the second summation processes the \( x \)'s with odd indices.

Rearranging, we get

\[ X_k = \frac{1}{2} \left[ \frac{1}{(N/2)} \sum_{j=0}^{(N/2)-1} x_{2j} w^{2jk} + w^k \frac{1}{(N/2)} \sum_{j=0}^{(N/2)-1} x_{2j+1} w^{2jk} \right] \]

or

\[ X_k = \frac{1}{2} \left[ \frac{1}{(N/2)} \sum_{j=0}^{(N/2)-1} x_{2j} e^{-2\pi i \left( \frac{jk}{N/2} \right)} + w^k \frac{1}{(N/2)} \sum_{j=0}^{(N/2)-1} x_{2j+1} e^{-2\pi i \left( \frac{jk}{N/2} \right)} \right] \]

Each summation can now be recognized as an \( N/2 \) discrete Fourier transform operating on the \( N/2 \) even points and the \( N/2 \) odd points, respectively.

\[ X_k = \frac{1}{2} [X_{\text{even}} + w^k X_{\text{odd}}] \]

for \( k = 0, 1, \ldots N-1 \), where \( X_{\text{even}} \) is the \( N/2 \)-point DFT of the numbers with even indices, \( x_0, x_2, x_4, \ldots \), and \( X_{\text{odd}} \) is the \( N/2 \)-point DFT of the numbers with odd indices, \( x_1, x_3, x_5, \ldots \).

Now, suppose \( k \) is limited to \( 0, 1, \ldots N/2 - 1 \), the first \( N/2 \) values of the total \( N \) values. The complete sequence can be divided into two parts:

\[ X_k = \frac{1}{2} [X_{\text{even}} + w^k X_{\text{odd}}] \]

and

\[ X_{k+N/2} = \frac{1}{2} [X_{\text{even}} + w^{k+N/2} X_{\text{odd}}] = \frac{1}{2} [X_{\text{even}} - w^k X_{\text{odd}}] \]

since \( w^{k+N/2} = -w^k \), where \( 0 \leq k < N/2 \). Hence, we could compute \( X_k \) and \( X_{k+N/2} \) using two \( N/2 \)-point transforms.

---

Slides for Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen © Prentice Hall, 1999. All rights reserved.
Each of the $N/2$-point DFTs can be decomposed into two $N/4$-point DFTs and the decomposition could be continued until single points are to be transformed. A 1-point DFT is simply the value of the point.

Computation often depicted in the form:

![Four-point discrete Fourier transform diagram](image)

Figure 11.29  Four-point discrete Fourier transform.
\[ x_k = \sum_{i=0}^{14} w^i \left( \sum_{j=0}^{7} a_{ij} \right) \]

**Sequential Code**

The sequential time complexity is essentially \( O(N \log N) \) since there are \( \log N \) steps and each step requires a computation proportional to \( N \), where there are \( N \) numbers.

The algorithm can be implemented recursively or iteratively.

Figure 11.30 Sixteen-point DFT decomposition.
Parallelizing the FFT Algorithm

Binary Exchange Algorithm

Figure 11.31 Sixteen-point FFT computational flow.

Figure 11.32 Mapping processors onto 16-point FFT computation.
Analysis
Computation

Given $p$ processors and $N$ data points, at each step each processor will compute $N/p$ points, and each point requires one multiplication and addition. With $\log N$ steps, the parallel time complexity is given by
\[ t_{\text{comp}} = O(N \log N). \]

Communication

If $p = N$, communication occurs at each step and there is one data exchange between pairs of processors at each of the $\log N$ steps.

Suppose a hypercube or another interconnection network that allows simultaneous exchanges is used. In that case, the communication time complexity is given by
\[ t_{\text{comm}} = O(\log N). \]

If $p < N$, the communication time complexity is simply given by
\[ t_{\text{comm}} = O(\log p). \]

Transpose Algorithm

If processors organized as a two-dimensional array, communications would first take place between processors in each column, and then in each row:

![Figure 11.33 FFT using transpose algorithm — first two steps.](image)
During the first two steps, all communication would be within a processor, as shown. During the last two steps, the communication would be between processors. In the **transpose algorithm**, between the first two steps and the last two steps, the array elements are transpose.

![Diagram of transpose algorithm](image1)

**Figure 11.34** Transposing array for transpose algorithm.

After the transpose, the last two steps proceed but now involve communication only within the processors. The only communication between processors is to transpose the array.

![Diagram of FFT using transpose algorithm](image2)

**Figure 11.35** FFT using transpose algorithm — last two steps.