ASPECTS OF PARALLEL TOPOLOGIES APPLIED TO DIGITAL TRANSFORMS OF DISCRETE SIGNALS

A Thesis submitted for the degree of Master of Science

by

R.J. White

1994

Department of Computer and Mathematical Sciences
Faculty of Science
Victoria University of Technology

This thesis contains no material that has been accepted for the award of any other degree or diploma in any University or Tertiary Institute. To the best of my knowledge and belief it contains no material published or written by another person, except where due reference is made in the text of the thesis.

(Signature)..[redacted]..

(Name)........R.J. WHITE........

(Date)........24/6/94........

ABSTRACT

Discrete transformations are widely used in the fields of signal and image processing. Applications in the areas of data compression, template matching, signal filtering pattern recognition all utilise various discrete transforms. The calculation of transformations is a computationally intensive task which in most practical applications requires considerable computing resources. This characteristic has restricted the use of many transformations to applications with smaller datasets or where real-time performance is not essential.

This restriction can be removed by the application of parallel processing techniques to the calculation of discrete transformations. The aim of this thesis is to determine efficient parallel algorithms and processor topologies for the implementation of the discrete Walsh, cosine, Haar and D4 Daubauchies transforms, and to compare the operation of the parallel algorithms running on T800 Transputers with the equivalent serial von Neumann type algorithm. This thesis also examines the transformations of a number of test functions in order to determine their ability to represent various common global and locally defined functions.

It was found that the parallel algorithms developed during the course of this thesis for the discrete Walsh, cosine, Haar and D4 Daubauchies transforms could all be efficiently implemented on a hypercube processor topology.

Development of a number of parallel algorithms also led to the discovery of a new parallel algorithm for the calculation of any transformation which can be expressed as a Kronecker or tensor product/sum. A hypercube based algorithm was devised which converts the Kronecker product to a Hadamard product on a hypercube structure. This provides a simple algorithm for parallel implementations.

Examination of the four sets of transform coefficients for the test functions revealed that all the transforms examined were not suitable for representing functions with large numbers of discontinuity's such as the chirp function. Also, transforms with local basis functions such as the Haar and D4 Daubauchies transforms provided better representations of localised functions than transforms consisting of global basis function sets such as the discrete Walsh and cosine transformations.

CONTENTS

ACKNOWLEDGEMENTS

I would like to thank my supervisors, Dr. Charles Osborne for his support, encouragement and guidance throughout the course of my research, and Dr. Nalin Sharda for his advice and suggestions.

I would also like to thank the Monash University Computer Imaging Group for their assistance.Thanks are also due to Mr. T.Peachey, Dr.P.Cerone and Mr A.McAndrew for their time and assistance.

I would especially like to thank my wife Anne for her patience, encouragement and support, without which this work would not have been possible.

CHAPTER 1

REVIEW OF DISCRETE TRANSFORMATIONS EMPLOYED IN DIGITAL IMAGE AND SIGNAL PROCESSING

1.1 INTRODUCTION

A transformation can be defined as a rule or mapping which assigns to each element of one set a unique element of another set. A more specific definition is given by Bracewell[6a] who defines a transform as an operation which is performed on a function. The transforms reviewed in this thesis are integral transforms where the operation performed consists of multiplying the function by another function known as the kernel function and integrating.

The motivation behind performing transformations is that in many situations such as electric potential distribution or heat diffusion (Kevorkian[36]) the solution to the problem in the function domain is difficult. Performing a transformation can result in a simpler problem in the transform domain. When a solution is found in the transform domain the inverse transformation back to the function domain provides a solution to the original problem, for example using Laplace transforms to solve linear differential equations (Kreyzig[41]).

Coates[17] shows that with a few exceptions functions can be expressed as the sum of a series of simpler basis functions. Integral transforms can be used to determine the series coefficients or weighting's required by the basis functions in order to represent the input function. For example the Fourier transform provides a set of coefficients which can be used to represent a function as the sum of a series of weighted trigonometric functions. Other integral transforms such as the Walsh and Haar transforms provide coefficients for different families of basis functions.

Transforms can therefore be used as a tool for converting information into more useful or amenable forms. This has led to their widespread use in the fields of feature identification (Schutte[62]) and other image processing techniques, filtering (Shynk[63]), speech recognition (Beauchamp[4a]), and data compression (Rao[57b]).

Performing transformations on an image or a long data sequence is a computationally intensive task. A large amount of work has been done to reduce transform computation times. Transforms which are widely used such as the discrete Fourier and discrete cosine transforms have been implemented as application specific integrated circuits (Richards[59],Sun[70]). Software developments have seen the evolution of a number of fast transform algorithms (Kou[39], Gertner[24], Bracewell[6b],Gupta[28a] and others) which reduce the number of calculations required to perform the discrete transform.

A recent development has been the application of parallel processing techniques to speed up transform calculations. Hardware implementations while being fast generally operate on small data sets and can only perform a specific transform. Software implementations of fast transforms on computer are more general, allowing any transform to be performed but are limited by the speed of the processor. Performing transformations on parallel multi-processor computers is an alternative offering the versatility provided by software implementations with a performance exceeding that of a conventional single processor computer. The aim of this thesis is to investigate efficient parallel processing topologies for the calculation of transformations widely used in the areas of signal and image processing.

1.2 THE FOURIER TRANSFORM AND ITS APPLICATION IN ONE DIMENSIONAL SIGNAL PROCESSING.

Orthonormal sets of functions can be used to synthesise any time function, enabling a waveform to be represented by the superposition of members of a set of basis functions (Beauchamp[4a]). The continuous Fourier transform decomposes a waveform into a series of weighted sinusoids. One of the most familiar forms of the Fourier transform is that which transforms a time function x(t) into the frequency function X(f) and is given by the relationship

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi f t}\,dt. \qquad (1.1)$$

If the transformation is to be performed by digital computation the input signal will be represented by a data sequence. The data may present itself naturally in discrete form, or if the data is continuous it can be discretized by sampling the continuous waveform at or above the Nyquist sampling rate (Coates[17]) in order to preserve an accurate representation of the continuous waveform. When data is in a discrete form machine computation of the Fourier transform can be performed by using the discrete Fourier transform, which is defined as

$$X(f) = \frac{1}{N}\sum_{k=0}^{N-1} x(k)e^{-j2\pi f k} \qquad (1.2)$$

where N is the total number of points in the data sequence.

A number of algorithms have been developed to compute the discrete Fourier transform. The Cooley-Tukey fast Fourier transform algorithm (Brigham[8]) was the catalyst for a large number of FFT algorithms proposing computational improvements based on tailoring algorithms to specific data types or processors. For example Sorenson et al[67] provide a review of the effectiveness of a number of FFT algorithms developed to process real data. Richards[59] and Pei[52] have developed a split radix FFT for efficient computation of complex data and efficient VLSI implementation. The Winograd Fourier transform is shown by Silverman[65] to provide computational efficiencies given specific computer hardware requirements are met. More recently fast Fourier transforms have been implemented via other transforms (Gupta[28b]) and neural networks (Culhane[18]).

Most of these algorithms are modifications of the Cooley-Tukey algorithm. This algorithm expresses the transform in terms of a series of sparse matrix multiplications. How this is done is illustrated by considering a specific example.

The one dimensional discrete Fourier transform can be expressed as

$$X(n) = \sum_{k=0}^{N-1} x(k)W^{nk} \tag{1.3}$$

where $W = e^{-j\frac{2\pi}{N}}$ and $n = 0, 1, \ldots\ldots, N-1$.

For values in the range 0 to 3 the variables k and n can be represented in binary form as $k = (k_1, k_0) = 2k_1 + k_0$ and $n = (n_1, n_0) = 2n_1 + n_0$

For the case N = 4 the binary representation of the discrete Fourier transform is

$$X(n_1, n_0) = \sum_{k_0=0}^{1}\sum_{k_1=0}^{1} x(k_1, k_0)W^{(2n_1+n_0)(2k_1+k_0)} \tag{1.4}$$

where n_i, k_i are the ith bit of the binary representations of n and k.

From equation(1.4) it follows that

$$X(n_1,n_0) = \sum_{k_0=0}^{1}\left[\sum_{k_1=0}^{1} x(k_1,k_0)W^{2n_0k_1}\right]W^{(2n_1+n_0)k_0} \qquad (1.5)$$

The summation in the [] brackets can be expressed as

$$x_1(n_0,k_0) = \sum_{k_1=0}^{1} x(k_1,k_0)W^{2n_0k_1} \qquad (1.6)$$

which can be expressed in matrix notation as

$$\begin{bmatrix} x_1(0,0) \\ x_1(0,1) \\ x_1(1,0) \\ x_1(1,1) \end{bmatrix} = \begin{bmatrix} 1 & 0 & W^0 & 0 \\ 0 & 1 & 0 & W^0 \\ 1 & 0 & W^2 & 0 \\ 0 & 1 & 0 & W^2 \end{bmatrix} \begin{bmatrix} x(0,0) \\ x(0,1) \\ x(1,0) \\ x(1,1) \end{bmatrix}. \qquad (1.7)$$

Similarly, the outer summation in (1.5) can be written as

$$X(n_1,n_0) = \sum_{k_0=0}^{1} x_1(n_0,k_0)W^{(2n_1+n_0)k_0} \qquad (1.8)$$

which is expressible as

$$\begin{bmatrix} X(0,0) \\ X(0,1) \\ X(1,0) \\ X(1,1) \end{bmatrix} = \begin{bmatrix} 1 & W^0 & 0 & 0 \\ 1 & W^2 & 0 & 0 \\ 0 & 0 & 1 & W \\ 0 & 0 & 1 & W^3 \end{bmatrix} \begin{bmatrix} x_1(0,0) \\ x_1(0,1) \\ x_1(1,0) \\ x_1(1,1) \end{bmatrix}. \qquad (1.9)$$

The transformation has been reduced to a sequence of matrix multiplications. The various stages of the transformation may be equivalently represented as a signal flow graph, as shown in figure 1.1.

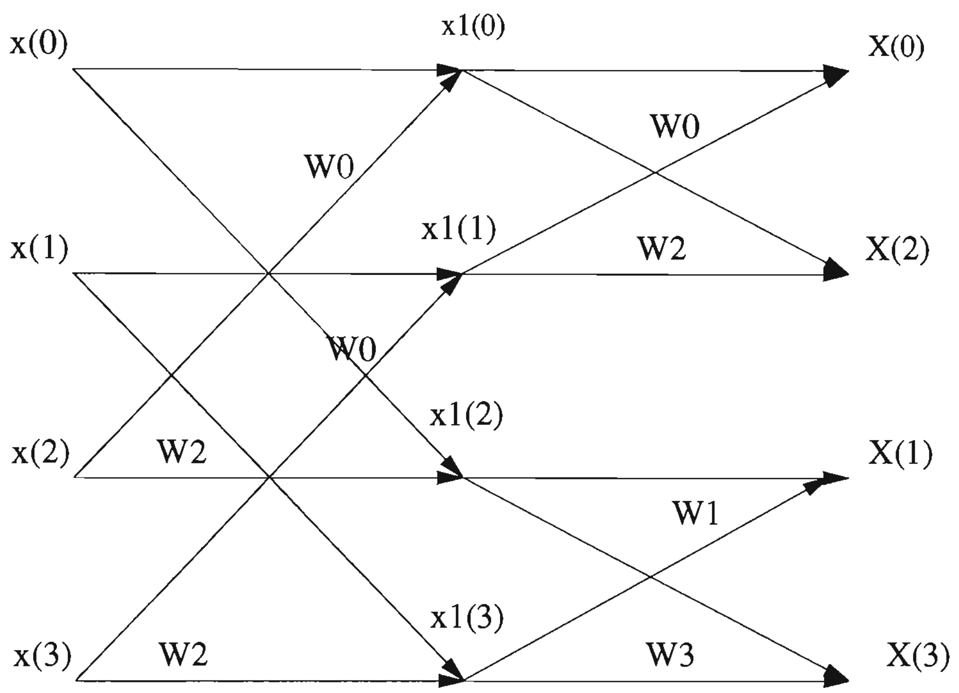


Figure 1.1 A signal flow graph of a fast Fourier transform for four data points.

This is the well known butterfly diagram and represents decimation in time of the original signal.

Many applications use Fourier transforms. Compression of speech and image data uses FFT conversion of the temporal or image data to frequency or spatial frequency domains in order to emphasise important frequencies and filter others (Lookbaugh[45]). The discrete Fourier transform is commonly used in filtering noise and signal detection (Shynk[63],Satt[61],Quirk[55]) and in the reconstruction of a signal from partial information (Dembo[22]). The restoration of blurred images using a wiener filter (Guan[27]), image zoom algorithms (Smit[66]) and cepstrum analysis (Wang[74]) are also accomplished using the FFT.

Many applications use Fourier transforms to calculate the correlation (also known as the covariance, see Otnes & Enochson[50]) and convolution functions (Beauchamp[4a]). The correlation function determines the degree of similarity between two functions (Gonzales & Wintz[25]). The correlation of two continuous functions f(x) and g(x) is defined by the relation

$$f(x)\circ g(x)=\int_{-\infty}^{\infty}\bar{f}(\alpha)g(x+\alpha)\,d\alpha \qquad (1.10)$$

where $\bar{f}$ signifies the complex conjugate. The discrete equivalent is defined as

$$f(x)\circ g(x)=\sum_{i=0}^{N-1}\bar{f}(i)g(x+i). \qquad (1.11)$$

It can be shown, for both the discrete and continuous cases, that the correlation theorem holds (Kraniauskas[40]). This is given as

$$\begin{aligned} f(t)\circ g(t) &\Leftrightarrow \bar{F}(f)G(f) \\ \bar{f}(t)g(t) &\Leftrightarrow F(f)\circ G(f) \end{aligned} \qquad (1.12)$$

where G(f) is the Fourier transformation of g(t). From this it can be seen that the correlation of two functions can be easily determined by calculating the product of the Fourier transformations of the functions.

Similarly the fast Fourier transform can be used in determining the convolution of two functions. The convolution of two functions f(t) and g(t) is defined as

$$f(t)*g(t)=\int_{-\infty}^{\infty}f(\alpha)g(t-\alpha)\,d\alpha \qquad (1.13)$$

with the discrete convolution being given as

$$f(t)*g(t) = \sum_{m=0}^{M-1} f(m)g(t-m). \qquad (1.14)$$

Similar to the correlation theorem (1.12), the convolution theorem states

$$\begin{aligned} f(t)*g(t) &\Leftrightarrow F(f)G(f) \\ f(t)g(t) &\Leftrightarrow F(f)*G(f) \end{aligned} \qquad (1.15)$$

An application of correlation is template or prototype matching. Template matching attempts to identify a signal by computing a correlation between a known signal and an unknown signal. If the correlation of the two functions yields a high value the unknown waveform closely matches the known waveform. A common use of template matching is in determining the location of a sub-image or feature within an image (Chou[15],Chakrabarti[11]).

The term correlation is also commonly referred to as cross-correlation, that is the correlation of two independent functions (Kraniauskas[40]). This term is used to distinguish between cross-correlation and auto-correlation which is the correlation of a function with itself. As shown in Beauchamp[4a] both cross-correlation and auto-correlation are widely applied in the fields of image processing (Whitebread[76]), signal processing applications such as radar pulse compression (Cenzo[10]) and transfer function identification (Fransaer[23]).

As with correlation, convolution also has a wide application base. Many physical systems in the course of their operation perform convolutions of sinusoidal functions. Such systems benefit from the application of Fourier transforms. For example, in the field of scanning spectrometry deconvolution using FFT's is used when determining the true absorption spectrum from the detected absorption spectrum (Marshall[46]).

When performing systems analysis and simulation a system output is determined by convolving a system input signal with the systems impulse response. Fourier transforms can be used to determine the convolution also Fourier transforms in a Galois field play a role in the study of error correcting codes see Blahut[5].

1.3 THE WALSH, HAAR AND DISCRETE COSINE TRANSFORMS

The widespread use of the Fourier transform in signal processing has resulted in the development of a number of other transforms which are either computationally faster or more appropriate for a particular application. Examples of such orthonormal transforms are the Walsh, cosine and Haar transforms. The Walsh transform is an orthogonal transform that is easy to implement digitally and computationally faster than the more widely used Fourier transform. However, it has the disadvantage that it gives a larger mean square error for low resolution calculations than other commonly employed transforms (Beauchamp[4a]). This is a limitation when signal reconstruction using the Walsh transform is based on a small dataset.

1.3.1 THE WALSH TRANSFORM

The Walsh functions form a set of rectangular waveforms with two amplitude values, +1 and -1 defined over a limited time interval. The first three basis functions, using sequency (similar to ordering by increasing frequency) or Walsh ordering (see Ahmed & Rao[1]), are shown in figure 1.2.

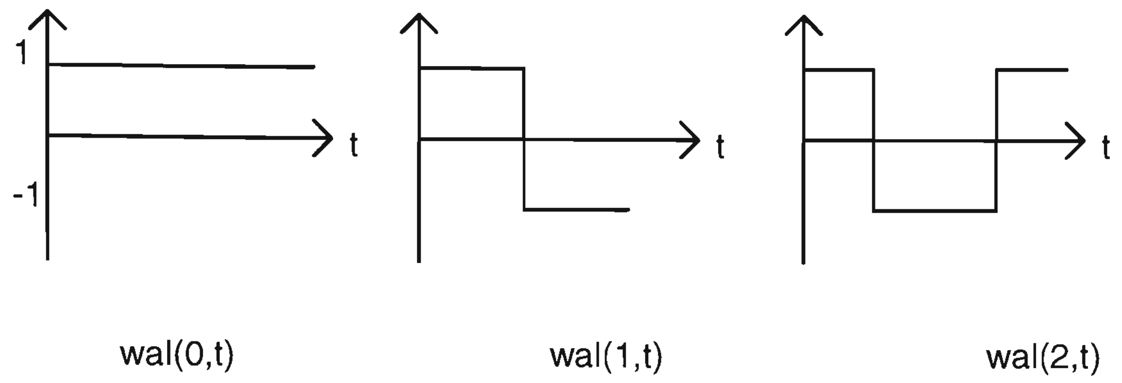


Figure 1.2 The first three sequency ordered Walsh functions.

One representation of the discrete Walsh function is given in (1.16), in which the function g(x,u), having $N = 2^n$ terms, is represented as a continued product (see Gonzales & Wintz[25]). This representation provides a simple derivation of the fast Walsh transform.

$$g(x,u) = \frac{1}{N}\prod_{i=0}^{n-1}(-1)^{b_i(x)\, b_{n-1-i}(u)} \tag{1.16}$$

where x = ordering number

u = time period

$b_i(x)$ = the ith bit of the binary representation of the value of x.

The discrete Walsh transform $W(u)$ is given by

$$W(u) = \sum_{x=0}^{N-1} f(x)\; g(x,u). \tag{1.17}$$

From equation(1.16) it may be noticed that the discrete Walsh transform can be represented as

$$W(u) = \frac{1}{N}\sum_{x=0}^{N-1} f(x)\prod_{i=0}^{n-1}(-1)^{b_i(x)\, b_{n-1-i}(u)}. \tag{1.18}$$

The fast Walsh transform algorithm can be determined using a similar technique to that used in determining the fast Fourier transform. This can be demonstrated by considering the case N = 4. Neglecting scaling

$$W(u) = \sum_{x=0}^{3} f(x) \prod_{i=0}^{1} (-1)^{b_i(x) b_{1-i}(u)} \tag{1.19}$$

where $$\prod_{i=0}^{1} (-1)^{b_i(x) b_{1-i}(u)} = (-1)^{b_0(x) b_1(u)} . (-1)^{b_1(x) b_0(u)}.$$

The binary representation of equation(1.19) for N = 4 is

$$W(u_1, u_0) = \sum_{x_0=0}^{1} \sum_{x_1=0}^{1} f(x_1, x_0)(-1)^{x_0 u_1}(-1)^{x_1 u_0}. \tag{1.20}$$

The inner summation of equation(1.20) can be written as

$$W_1(u_0, x_0) = \sum_{x_1=0}^{1} f(x_1, x_0)(-1)^{x_1 u_0} \tag{1.21}$$

with the outer summation being written as

$$W(u_1, u_0) = \sum_{x_0=0}^{1} W_1(u_0, x_0)(-1)^{x_0 u_1}. \tag{1.22}$$

Both of the summations in (1.21) and (1.22) can be enumerated in matrix form in the same manner as for the fast Fourier transform. When this is done the summations can be combined to give a matrix representation of the fast Walsh transform as given by equation (1.23). This representation reveals the discrete Walsh transform to be a "hard limited" discrete Fourier transform with the sinusoidal functions replaced by square waves.

$$\begin{bmatrix} W(0,0) \\ W(1,0) \\ W(0,1) \\ W(1,1) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} f(0,0) \\ f(0,1) \\ f(1,0) \\ f(1,1) \end{bmatrix}. \qquad (1.23)$$

The various stages of the transformation given in equation(1.23) may also be represented by a signal flow graph, as shown in figure 1.3.

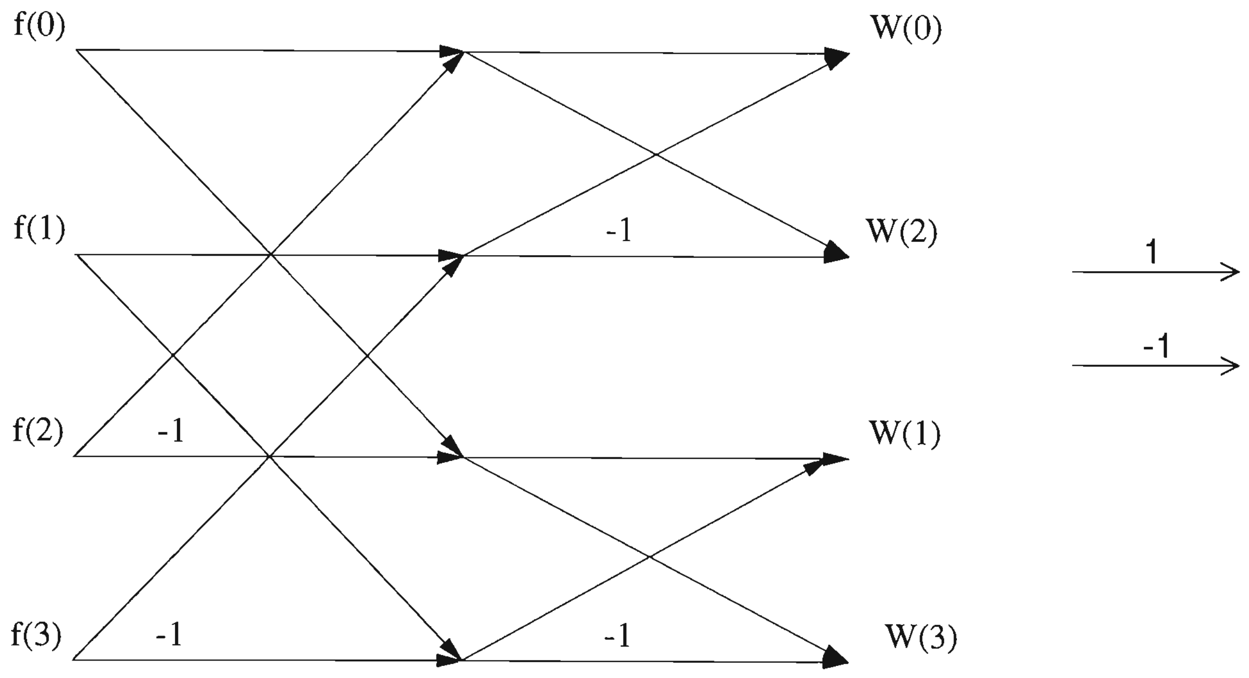


Figure 1.3 A signal flow graph of a fast Walsh transform for four data points.

The Walsh transform can also be found using Hadamard matrices (Beauchamp[4b]). A Hadamard matrix is a square matrix whose elements are only 1 and -1 arranged so that its rows and columns are orthogonal to one another. The lowest order Hadamard matrix is of order two as shown by equation(1.24)

$$\mathbf{H}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \qquad (1.24)$$

Higher order Hadamard matrices can be obtained from the recursive relationship

$$\mathbf{H}_N = \mathbf{H}_{\frac{N}{2}} \otimes \mathbf{H}_2 \qquad (1.25)$$

given $N = 2^n$ and $\otimes$ denotes the Kronecker or tensor product.
A definition of Kronecker products can be found in Brewer[7]. Given a matrix **A** of size $m \times n$ and a matrix **B** of size $p \times q$ the Kronecker or tensor product is a matrix of size $mp \times nq$ as shown by equation(1.26)

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & . & . & . & a_{1n}\mathbf{B} \\ . & & & & . \\ . & & & & . \\ . & & & & . \\ a_{m1}\mathbf{B} & . & . & . & a_{mn}\mathbf{B} \end{bmatrix}. \qquad (1.26)$$

The sparse matrix product used when determining the fast Walsh transform as shown in equation (1.23) can be combined into a single matrix as shown below

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}. \qquad (1.26)$$

The resultant matrix can be seen to be a Hadamard matrix of order four. The Walsh transformation matrices can therefore also be represented as a Hadamard matrix of order N.

1.3.2 THE DISCRETE COSINE TRANSFORM

The cosine transform is based on a sinusoidal kernel function

$$K(\omega,t) = \cos(\omega t). \qquad (1.27)$$

The discrete cosine transform is therefore defined as

$$X(n) = \frac{1}{N}\sum_{i=0}^{N-1} x(i)\cos\left(\frac{n\pi(2i+1)}{2N}\right). \qquad (1.28)$$

The discrete cosine transform is computationally more efficient than the discrete Fourier transform and provides efficient energy compaction similar to that found with the optimal Karhunen-Loeve transform (Beauchamp[4a]) and may be implemented as a fast transform. This has ensured widespread use in image and speech compression (Ngan[49]). A number of discrete cosine transformations have been developed (Kou[39]), one of the best is that given by Chen[14]. Because of its efficiency at image data compression (Cham[12]) the discrete cosine transform is widely used for image coding of video frames (Roese[60]), and is recommended as part of the JPEG colour image data compression algorithm (Rao[57a]). Many fast algorithms for computing the discrete cosine transform have been developed. A review of various discrete cosine transform algorithms is given by Chelemal[13] and Hou[35]. These algorithms can be classified into one of the following categories. Calculation of the discrete cosine transform via another transform, recursive computation, or sparse matrix multiplication. A common implementation is the sparse matrix multiplication. A variety of variations of this have been developed most based on the DCT-II algorithm given by Rao[57a] and shown in figure 1.6.

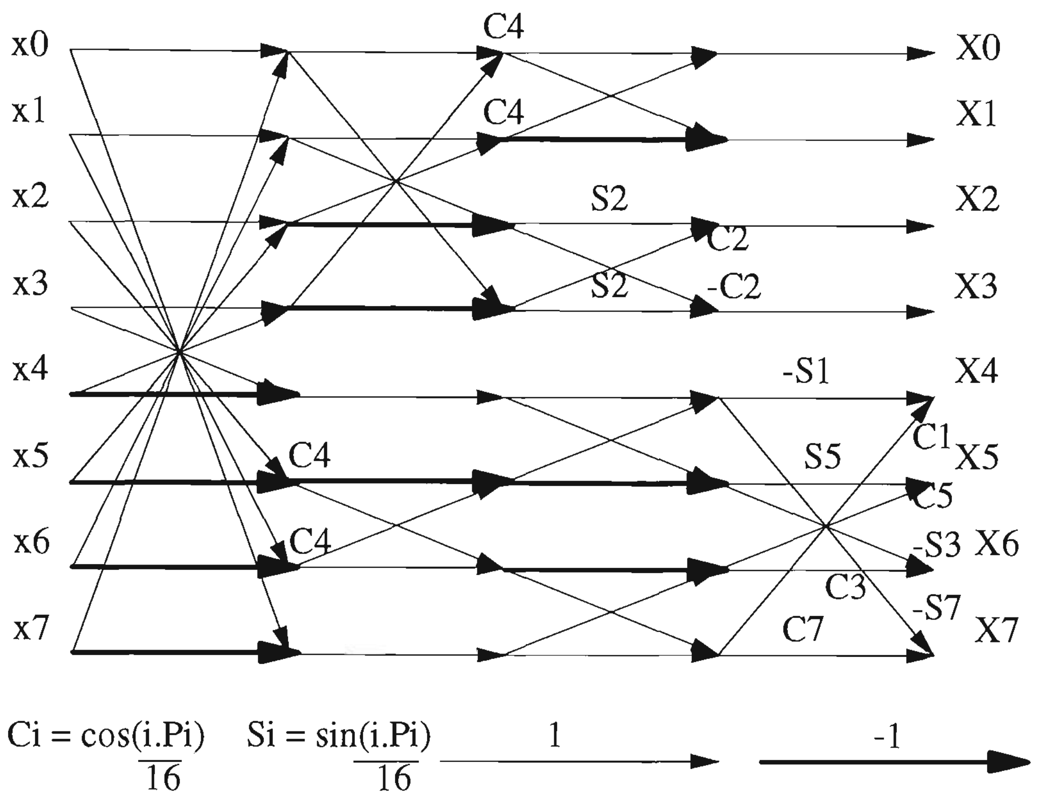


Figure 1.6 A signal flow diagram for the cosine transform for N = 8.

1.3.3 THE HAAR TRANSFORM

The Haar transform is based on a set of periodic rectangular waveforms known as Haar functions. These are defined (Ahmed & Rao[1]) as

$$h(0,0,t) = 1 \qquad t \in [0,1) \qquad (1.29)$$

$$h(r,m,t) = \begin{cases} 2^{\frac{r}{2}} & \dfrac{m-1}{2^r} \le t < \dfrac{m-\frac{1}{2}}{2^r} \\ -2^{\frac{r}{2}} & \dfrac{m-\frac{1}{2}}{2^r} \le t < \dfrac{m}{2^r} \\ 0 & \text{elsewhere } t \in [0,1) \end{cases}$$

where $0 \le r < \log_2 N$ and $1 \le m \le 2^r$.

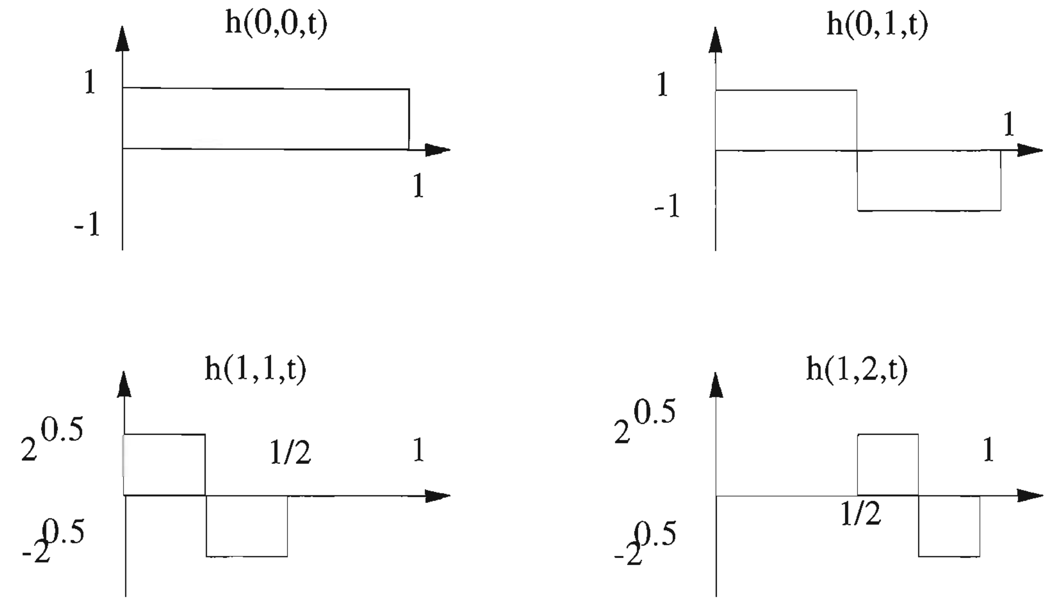


Figure 1.4: The first four continuous Haar functions.

The Walsh, Fourier and cosine transforms all have global basis functions, the Haar transform was the first transformation to have both global and local basis functions. As shown in figure 4 the first two Haar basis functions are global, all other basis functions are local in space. Work by Goupillard, Morlet[29] and others has led to the development of families of basis functions known as wavelets (Strang[69]). Wavelet basis functions consist of translations and dilations of a wavelet function. As most of the Haar basis functions are also translations and dilations of a square wave the Haar functions are now seen to be a member of the family of wavelet functions.

A wavelet function ψ is generated by means of translations and dilations of a scaling function ϕ (Daubauchies[20]) as given below

$$\psi(x) = \sum_k (-1)^k C_{1-k}\phi(2x-k). \qquad (1.30)$$

The scaling function used to create the wavelet function is determined recursively by means of the dilation equation(1.31)

$$\phi_j(x) = \sum_k C_k \, \phi_{j-1}(2x-k). \qquad (1.31)$$

The form of a specific wavelet function is dependant upon the choice of ϕ_0 and the coefficients C_k. For example, the Haar function can be created using the wavelet equations when ϕ_0 is the box function (Strang[69]).

Mapping the Haar functions into the discrete domain by sampling the continuous functions results in a matrix of discrete values. Equation(1.32) is an example of the case N = 8,

$$Ha_8 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ \sqrt{2} & \sqrt{2} & -\sqrt{2} & -\sqrt{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sqrt{2} & \sqrt{2} & -\sqrt{2} & -\sqrt{2} \\ 2 & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & -2 \end{bmatrix}. \qquad (1.32)$$

The Haar transformation **X** can therefore be expressed as a matrix multiplication such as

$$\mathbf{X} = \mathbf{Ha} \, . \, \mathbf{x} \qquad (1.33)$$

where **Ha** is the NxN Haar matrix and **x** is a data vector.

The first two Haar transform coefficients, like the coefficients of the discrete Fourier and Walsh transforms, are a function of all the values in the original data space. The other Haar transform coefficients are a function of a subset of the original data space. This implies that unlike the Fourier and Walsh transformations the Haar transform is both globally sensitive and locally sensitive to the data space.

A number of algorithms to compute the Haar transform have been developed. The algorithm by Andrews[3] is shown as a signal flow diagram in figure 1.5.

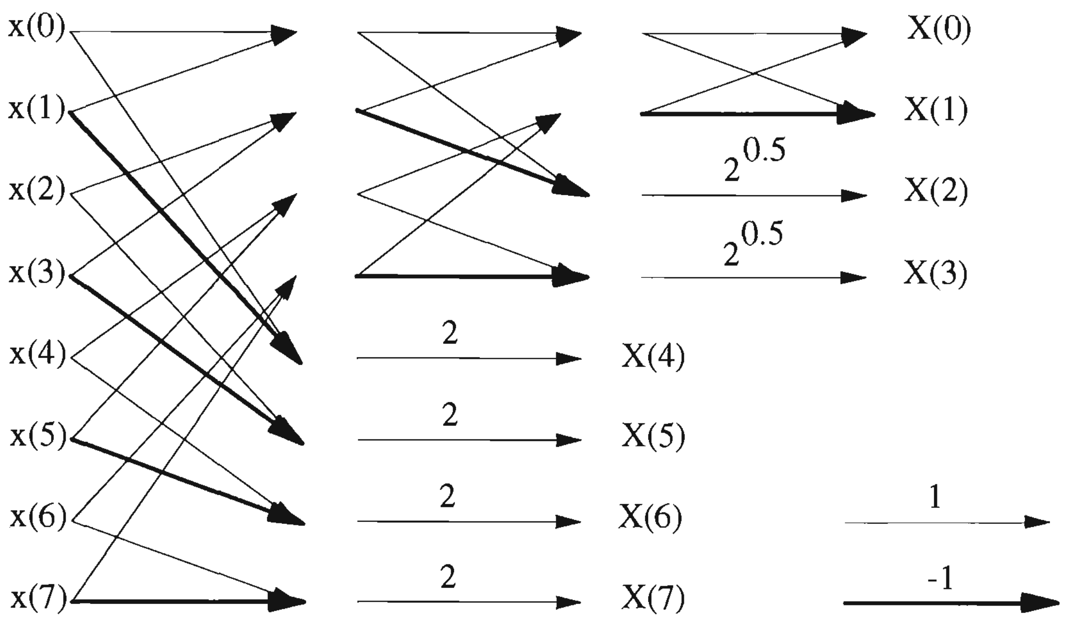


Figure 1.5 A signal flow diagram of the Haar transform for N = 8.

As can be seen from the signal flow diagram the Haar transform performs 2(N-1) additions and subtractions and N multiplications as opposed to the $N\log_2 N$ operations of the fast Walsh and Fourier transforms. The time required to perform the Haar transform is therefore linearly proportional to the size of the dataset N, whereas the transformation time of the transforms outlined earlier is proportional to $N\log_2 N$.

1.4 APPLICATIONS OF TRANSFORMS TO FEATURE EXTRACTION AND DATA COMPRESSION

In the context of signal and image processing the transformation operation can be interpreted as being a process of feature extraction. When corresponding elements of the data vector and the basis function or transformation matrix have similar values and signs, a large positive value of the transform coefficient will result.

A large transform coefficient value implies that the "shape" of the data vector and the basis function are similar. Different transforms can be employed to efficiently detect different features in the data vector. For example, the Walsh transform has been used to detect straight line roads and rectangular buildings in aerial photographs (Beauchamp[4b]).

Another application of the process of transformation is to use it as a mechanism for data compression. When using transform coding techniques for data compression the data is converted into the transform domain, transform coefficients having comparatively small values are discarded and the remaining coefficients representing the compressed data are either transmitted or stored. Reconstruction of the original data is carried out by replacing the transform coefficients not transmitted or stored with zeros and performing the inverse transformation.

The choice of a particular transform is dependant upon the level of reconstruction error that can be tolerated, and the processing resources and requirements. The data compression capabilities of the transforms discussed earlier have been studied extensively (Thomas[72]) and it has been found that given a set percentage retention of transform coefficients the mean square error introduced into a reconstructed image or signal is least when the discrete cosine transform is used. This is followed by the Fourier, Walsh and Haar transforms.

1.5 LIMITATIONS OF GLOBAL TRANSFORMS WHEN FEATURES APPEAR AT A NUMBER OF SCALES

For many different types of signals the important information is carried by singularities and sharp variations in magnitude. For example, in the field of image processing, points of sudden variation provide the locations of contours or edges in satellite and biomedical images (Khanh[37]). Another example is signal processing applications such as ECG, where detection of specific fluctuations in heart rate (Nandagopal[48]) are required. While in processing a time series it may be important to determine both the temporal location of a constituent structure as well as its frequency.

A drawback of global transforms such as the Fourier, cosine and Walsh transforms is that they are unable to describe the spatial locations of singularities or non-stationary structures within a signal. This is because the basis functions of global transforms are expressed over the entire data space and therefore do not possess local sensitivity. Transforms which are well adapted to characterise transient phenomena in signals are transforms whose basis functions are localised in space and frequency. The wavelet transform decomposes a signal on a set of basis functions with compact support and thus can represent a signal as a function of both time and frequency.

The continuous wavelet transform of a function f with respect to the wavelet g may be defined as

$$F_{a,b} = \frac{1}{a}\int g\left(\frac{x-b}{a}\right) f(x)\, dx, \quad a>0, \quad b \in R \qquad (1.34)$$

The position and magnification of the wavelet basis functions can be specified by the values assigned to b and a respectively. Therefore when features appear on a number of scales (Grasseau[30]) they may be detected in the transformation to whatever accuracy is required by appropriate dilation and translation of the wavelet basis functions. Also a particular wavelet function can be chosen to detect a matching feature in the signal or image.

For example Tuteur[73] uses the inverse Fourier transform of the frequency function

$$G(f) = e^{-(af-m)^2} \tag{1.35}$$

as the analysing wavelet function because of its ability to extract Ventricular late potential's from background noise in clinical electrocardiograms, while Grasseau[30] studies the local scaling properties of fractal objects by using a Gaussian type wavelet function

$$g(x) = (1-x^2)e^{\frac{-x^2}{2}} \tag{1.36}$$

because of its fast rate of decay.

1.6 IMPLEMENTATION TECHNIQUES AND PROBLEMS WITH CURRENT TRANSFORM METHODS

There are basically two major methods of implementation of transforms, these are software implementations on general purpose computing machines and specialised hardware/software implementations. The flexibility and availability of the general purpose computer makes software implementation an attractive alternative for carrying out fast transform algorithms. Evidence of this popularity can be seen in the large number of transform algorithms developed for the general purpose computer (Davies[21], Sinha[64]).

A major drawback of this type of implementation has been that the processing time required to perform the transform is too great for many applications; for example, high frequency signal analysis or real time video frame processing. This has been overcome to a large extent by employing specialised hardware/software solutions (Chou[15],Lewis[44],Healey[31]) where the particular application is time critical. Of course such solutions require hardware appropriate to the particular application with software often written in an assembler programming language and optimised for that specific system. When an implementation takes this form, improvement in performance is made at the cost of flexibility and portability of the transform.

If a transform which operates on large datasets is implemented currently on a von Neumann type computer a trade-off has to be made. A general algorithm can be developed which can be run on many different types of computer but which is relatively slow, or a specialised device-specific algorithm can be developed to run optimally on a specific machine, providing fast performance but at the cost of being unsuitable for implementation on any other platform.

1.7 MATRIX REPRESENTATION OF TRANSFORMS AND THEIR RELATIONSHIP TO PARALLEL IMPLEMENTATIONS OF TRANSFORMS

The development of the fast transforms as outlined earlier in this chapter indicates that a discrete transformation can be represented as a matrix multiplication of a data vector (or matrix) with a matrix representing the discrete form of the transform basis functions. There are two possible ways that this matrix multiplication can be represented. One way is to depict the operation as a matrix multiplication between a single densely populated basis function matrix and the data matrix. An example of this is the Walsh transform for N=4 shown in equation(1.37).

$$\begin{bmatrix} W_0 \\ W_2 \\ W_1 \\ W_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \end{bmatrix}. \qquad (1.37)$$

The other method is to represent the transformation as the product of a number of comparatively sparse matrices as given by equation(1.23). These representations of a discrete transform can indicate possible methods of parallel implementation of the transform.

The sparse matrix form of a discrete transform when viewed as a signal flow graph shows clearly any data dependencies in the transform operation and highlights independent and hence parallelizable operations. This form of representation is particularly valuable when attempting to implement a transform in parallel on a MIMD (Multiple Instruction Multiple Data) parallel architecture (Almasi[2]). The MIMD parallel processing paradigm is based on independent processors operating in parallel and communicating by passing messages. Such a processing architecture can be represented schematically by a signal flow graph. Signal flow representation of a transform can therefore provide a direct means of determining parallel MIMD implementation.

On the other hand, the transform may be implemented on a massively parallel SIMD (Single Instruction Multiple Data) architecture (Almasi[2]). A processor topology with all processors executing the same instruction simultaneously on different datasets. A common SIMD processor topology is the mesh. Implementing a transform on this type of architecture can be a straight forward process which operates directly on the dense basis function matrix. From this it can be seen that the various ways of representing a transform matrix can provide information about and, in some cases, determine optimal parallel implementations.

As the parallel implementation of a wide variety of discrete transforms was to be investigated a transputer based MIMD system was chosen because of the large degree of programming flexibility it provides, and because commercially available systems allowed a number of different processor topologies to be investigated.

1.8 SUMMARY

This chapter provides a brief review of the Fourier, cosine, Walsh/Hadamard, Haar and wavelet transforms. Their application to the fields of data compression and feature extraction is outlined. The advantages and disadvantages of a number of discrete transform implementations are reviewed and possible techniques for implementation of transforms on parallel processing computers are discussed.

CHAPTER 2
REVIEW OF THE TRANSPUTER

2.1 INTRODUCTION

Many of the tasks which computers are required to perform such as simulations, computer modelling (Kothe[38], Ransom[56]) and signal processing (Sousa[68]) are computationally intensive . Tasks of this nature can require significant amounts of processing time, even when the fastest microprocessors are used. In an attempt to reduce the processing time required to perform these tasks the concept of parallel processing was introduced.

Parallel processing employs a divide and conquer technique. A particular task is divided into a number of sub-tasks. These sub-tasks are then distributed among a number of processors which execute these tasks concurrently. Many different architectures have been designed employing parallel processing techniques. The variety of implementations of the parallel processing paradigm is so great that a number of parallel processing taxonomies have arisen. The classification system proposed by Flynn (Almasi[1]) and foreshadowed in chapter one is widely used.

The two criteria employed in this system are the number of instruction and data streams. The single processor von Neumann computational model would be viewed as a single stream of instructions working on a single stream of data. The other two classifications of interest are SIMD (single instruction multiple data) and MIMD (multiple instruction multiple data). The SIMD classification covers such architectures as the vector and array processors. The MIMD model includes any architecture that consists of multiple microprocessors operating independently on their own individual data streams. The transputer was designed to achieve improved processing speed-up by utilising the MIMD computational model.

Transputer is a generic name for a family of VLSI components developed by INMOS. Each member of the family possesses its own unique characteristics, but all conform to the same general architecture. This consists of a single microcomputer comprising: processor, system services, RAM, and a number of autonomous serial communications links allowing inter-transputer communication.

First introduced in 1985, the first microprocessor in the transputer series was the T414. Since that time the T212, T800, T222, T425, T805 and T9000 have all been released, each new release providing some improvement in functionality over its predecessor.

2.2 HARDWARE DESIGN OF TRANSPUTERS

The transputer is based on the concept of communicating sequential processes. The particular transputer that is used in this thesis is the T800. The T800 is a 32 bit CMOS microcomputer. It has an on-chip 64 bit floating point unit, 4 Kbytes of on-chip RAM, a configurable memory interface and four bi-directional INMOS communication links.

Processing speed-up is obtained by hardware multi-tasking and by concurrent operation of the CPU and FPU. Fast memory access is available by storing a program or data in the on-chip RAM while, if required, a maximum of 4 Gbytes of memory is available via the memory interface.

What makes the transputer particularly distinctive is its four serial communications links providing communication between transputers. Operating at 2.35 Mbytes per second when running bi-directionally these links allow any number of transputers to be interconnected in the most appropriate configuration for the particular task. Rapid processing of large tasks can be achieved by dividing the task into components and distributing it over the transputer network.

Transputer systems have been found to be particularly suitable when attempting to solve problems in fields such as robotics (Daniel[19]) that benefit from the employment of asynchronously operating sophisticated microprocessors, or process simulation (Ponton[53]) when processing speed-up is best served by coarse grained parallelism.

Coarse grained parallelism is the term given to the division of a task into a small number of relatively large sub-tasks. This technique is easily mapped onto MIMD architectures. The alternative is termed fine grained parallelism, which is the division of a task into a large number of small sub-tasks. This technique is more suitably mapped onto SIMD architectures, such as array processors. An example of the former is the division of fast transform operations onto a small number of processors as shown later in chapter three, an example of fine grained parallelism is a matrix operation where each individual matrix element is distributed to a processor.

Parallel computing applications tend to be naturally easier to implement in one granularity than in the other. As the transputer is a microprocessor that operates autonomously and communicates with other transputers by message passing it is more amenable and efficient in implementing coarse grained parallel applications. Fine grained parallelism would result in a system of great complexity. The complexity arising from attempting to operate a large number of autonomous processors each communicating by asynchronous message passing.

When creating a parallel application a conscious decision must be made as to what is the most profitable configuration for the application. An image processing task for example, may be easier to implement on a SIMD architecture, but most of these architectures exist as specialised stand alone machines. On the other hand, the same task may be harder to design in a coarse grained form but can be mapped onto a multi-transputer plug-in card attached to a personal computer and so be more industrially applicable. Trade-offs such as these must be considered when determining what hardware and methodology to employ to perform a particular task.

2.3 THE TRANSPUTER SYSTEM USED

The hardware configuration consists of two T800-20 transputers, one with 1 Mbyte of DRAM and the other with 32 Kbytes of SRAM and 2 Mbytes of DRAM. These are attached to a B008 plug in board.

The B008 board is a transputer motherboard that plugs into one of the expansion slots of an IBM compatible PC motherboard. It consists of a 16 bit T212 transputer that controls a software configurable C004 32 way crossbar link switch and associated logic. The board can accommodate up to 10 transputer modules which fit into plug-in slots.

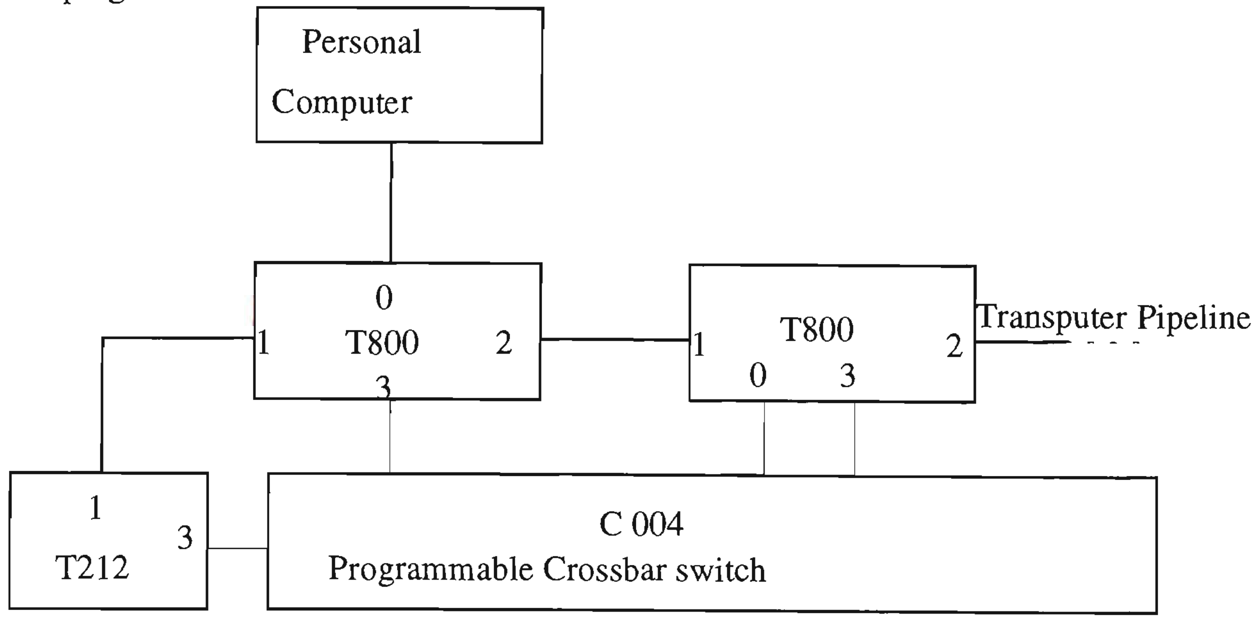


Figure 2.1 Block diagram of the B008 Transputer system.

Transputer links 1 and 2 are hardwired on each of the slots so that when the transputers are plugged in they form a pipeline of processing elements. The remaining transputer links can be connected or "softwired" via the IMS C004 programmable link switch. The softwired links can be configured by utilising the module/motherboard software (mms2) and the hardware description language it provides called HL1.

2.4 THE PARALLEL C PROGRAMMING LANGUAGE

As the transputer architecture is based on the communicating sequential processes model with on-chip serial communications links, parallel processing utilising transputer systems must consist of concurrent sequential processes or tasks which communicate with each other. The programming language used for developing the application software on the transputer system was Parallel C developed by 3L Systems Pty Ltd. Parallel C provides two programming methods for use with transputer systems which conform to this processing model, these are the communicating task technique and the processor farm technique.

An application using the communicating task model would consist of a collection of independent concurrently executing tasks each with its own input and output vectors. The Parallel C programming language includes configuration software for the communicating task model which provides a means of mapping the collection of software tasks onto the physical network of processors. For the programmer this takes the form of a configuration file (see Appendix A) which specifies tasks, their interconnection and their placement on physical processors. Each processor can support any number of software tasks within the restrictions of available memory. Communication between tasks on the same processor being via a designated memory location. Communication between tasks placed on different physical processors is via the transputer serial links, therefore communications between tasks on different processors is limited by the available physical connections.

The processor farm technique consists of a single master task and a number of identical worker tasks. A copy of the master and worker tasks is placed on the transputer which interfaces between the transputer board and the PC, this is known as the root transputer. A copy of the worker task is placed on all other transputers in the network. The master task disseminates data to, and collects data from, the worker tasks. The worker tasks accept data, perform a calculation and return a result. Data is automatically routed to any free worker task. The configuration file required for a processor farm is minimal, consisting of a listing of master and worker task filenames and memory requirements. Also, a processor farm implementation will automatically configure itself to run on any transputer network. Therefore, transputers can be added to or deleted from the network without recompilation or reconfiguration of the application software.

2.5 AN EXAMPLE OF TRANSPUTER PROGRAMMING: THE ONE DIMENSIONAL CONVOLUTION

One of the most efficient means of determining the convolution of two discrete one dimensional functions f(x) and g(x) is by use of the convolution theorem (equation(1.15)). This states that given the Fourier transform of the function f(x) is F(u) and the Fourier transform of the function g(x) is G(u) then the convolution of two functions is given by the inverse transformation of the product of the vectors F(u) and G(u). Gonzales[25] shows that the inverse Fourier transformation may be calculated by utilising the same algorithm that is used to calculate the Fourier transformation. In order to calculate the inverse transform the data vector is run through the same algorithm as the Fourier transform the only variation being that the resulting vector is divided by N, the size of the dataset. Therefore, both the transformation and inverse transformation can be calculated using the same algorithm.

A primary consideration when designing a transputer implementation which is not a processor farm is the number of transputers available and their possible configurations. This can affect both the degree of parallelisation of the algorithm and the mapping of software tasks onto the physical system. The following example is based on the two transputer system outlined in section 2.3.

The core task in a one dimensional convolution is the fast Fourier transform, so parallelisation of this task will be considered first. The Cooley-Tukey algorithm for the fast Fourier transform (Brigham[8]) can be displayed as a signal flow graph as given in figure 1.1. As shown in figure 2.1 this method of depicting the transformation provides a means for determining how the operation could be distributed between two processors.

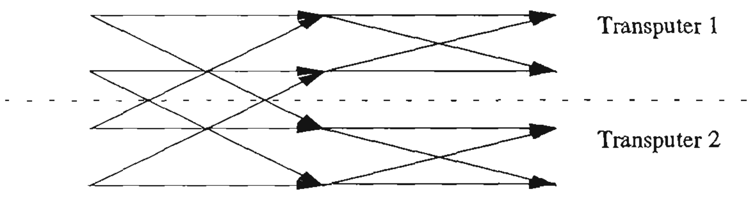


Figure 2.2 The division of butterfly operations between two transputers.

With the operations distributed between two transputers as shown in figure 2.2 the implementation could take the following form. The data vectors representing the functions f(x) and g(x) are sent to the two transputers, each transputer performing the operations outlined above. The product of the elements of the transformation vectors F(u) and G(u), resident on each transputer are found. These results are then redistributed amongst the transputers and the inverse transformation is performed. The results of the inversion are then collected by the root transputer.

Parallel implementations such as this using the communicating sequential processes model can be best exemplified by use of Petri nets (Murata[47]) because they clearly represent message passing and operations or state transitions. Figure 2.3 details a Petri net representation of the one dimensional convolution on a two transputer system.

Once the basic algorithm and its mapping onto a target transputer system is determined the next stage is the development of the application software. The software development on a B008-transputer system consists of a configuration file and the transputer parallel C programs.

The configuration file is an ASCII file comprising five distinct components. The first two components are a description of the physical system. The first part is a list of the physical processors present in the system, allocating each transputer and the host PC identifying labels (User guide[51]).

```
e.g.    PROCESSOR HOST        - the PC
        PROCESSOR TRANS1      - transputers present on the B008 board
        PROCESSOR TRANS2
```

The next component is a description of the physical communication links between all processors in the system. Each link is identified by the specifier WIRE followed by an identifying label for the link, followed by a description of the processors connected and which processor communication ports are connected.

```
e.g.    WIRE  JUMPER1     HOST[0] TRANS1[0]
        WIRE  JUMPER2     TRANS1[2] TRANS2[1]
```

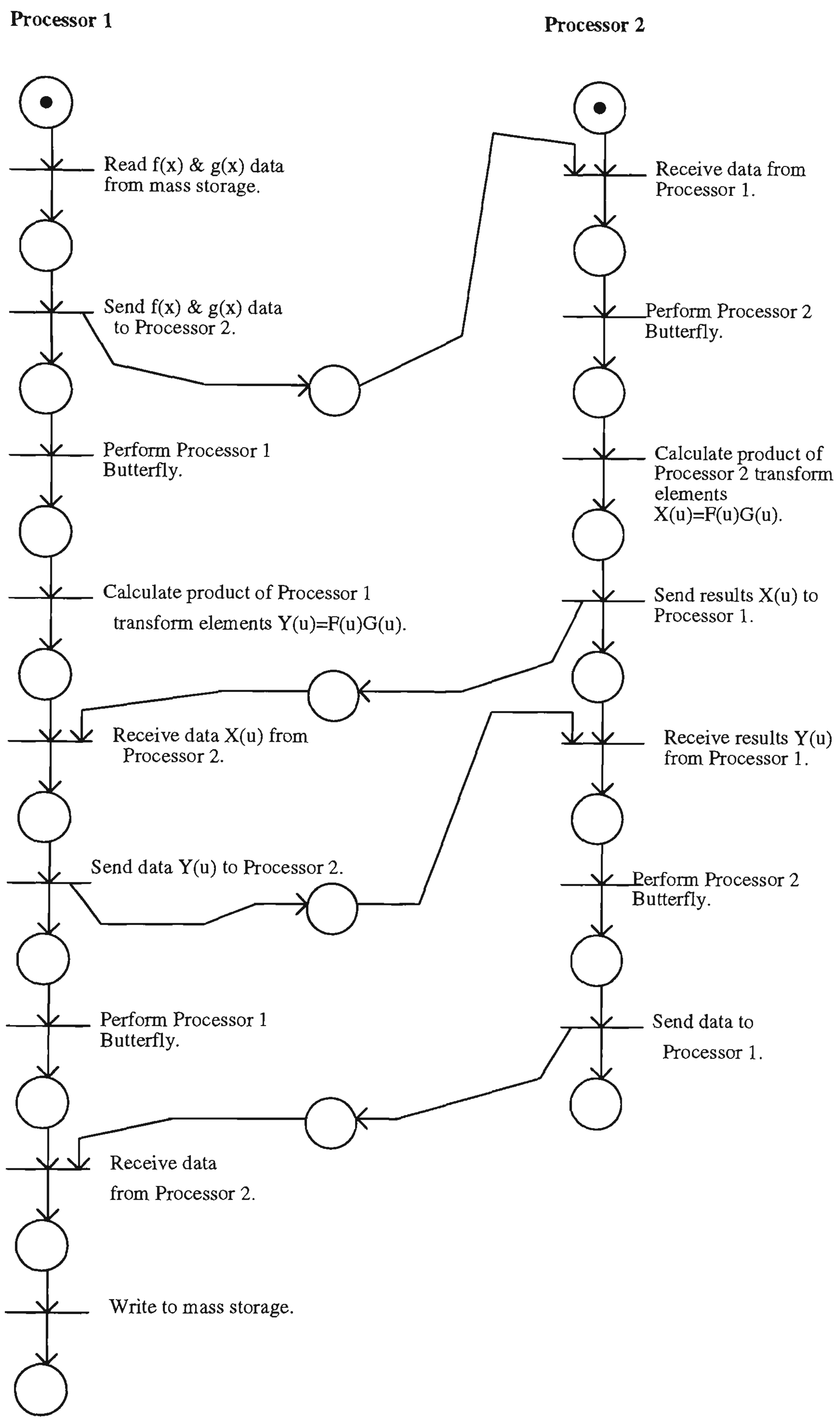

Processor 1
Processor 2
Read f(x) & g(x) data from mass storage.
Send f(x) & g(x) data to Processor 2.
Perform Processor 1 Butterfly.
Calculate product of Processor 1 transform elements Y(u)=F(u)G(u).
Receive data X(u) from Processor 2.
Send data Y(u) to Processor 2.
Perform Processor 1 Butterfly.
Receive data from Processor 2.
Write to mass storage.
Receive data from Processor 1.
Perform Processor 2 Butterfly.
Calculate product of Processor 2 transform elements X(u)=F(u)G(u).
Send results X(u) to Processor 1.
Receive results Y(u) from Processor 1.
Perform Processor 2 Butterfly.
Send data to Processor 1.

Figure 2.3 Petri net representation of two transputer one dimensional convolution.

After this is a listing of all the parallel C programs, known as tasks, and the number of logical communication links available to each software task. The tasks which are incorporated into the system include both the application software which has been developed and various 3L Parallel C library tasks such as FILTER and AFSERVER. These tasks are provided in order to facilitate communications between the transputers, PC and mass storage devices. Detailed information is available from the 3L Parallel C users manual[51].

e.g.
```
TASK AFSERVER   INS=1 OUTS=1
TASK FILTER     INS=2 OUTS=2
TASK FOURIER1   INS=3 OUTS=3
```

The next component of the configuration file is a listing of the placement of all the software tasks onto the appropriate physical processor.

e.g.
```
PLACE  AFSERVER HOST          - afserver runs on PC
PLACE  FILTER TRANS1    - filter & fourier1 both run on transputer 1
PLACE  FOURIER1  TRANS1
```

Finally the last section of the configuration file is a listing of the logical communications links between software tasks. Bi directional links must be specifically stated as shown below.

e.g.
```
CONNECT  ?  AFSERVER[0]      FILTER[0]
CONNECT  ?  FILTER[0]        AFSERVER[0]
```

As the transputer system is based on the model of communicating sequential tasks the parallel C programs are similar to sequential C programs with the addition of library functions to enable transfer of data between tasks. Examples of the parallel C programs are given in appendix A.

2.6 SUMMARY

This chapter introduces the major parallel processing architecture classifications and provides a general overview of the Transputer. A description of the Transputer system used in this thesis and the Parallel C programming language is given. Examination of Transputer systems revealed that the Transputer is most suitable for implementing parallel algorithms employing "coarse-grained" parallelism. An example of such an implementation, the one-dimensional convolution is outlined.

CHAPTER 3
PARALLELISING THE DISCRETE WALSH AND COSINE TRANSFORMS

3.1 INTRODUCTION

The discrete Walsh and cosine transforms were selected for parallel implementation because of their widespread use in implementations employing global transformations. The discrete cosine transform as mentioned in Chapter 1 is widely used in data compression of images being part of the JPEG standard. The discrete Walsh transform although not widely used itself has a close similarity to the fast Fourier transform, enabling any parallel Walsh implementation to be easily applied to the widely used Fourier transform or any similar transform.

3.2 THE WALSH TRANSFORM AND ITS RELATIONSHIP TO THE FOURIER TRANSFORM

The Walsh and the Fourier transforms both belong to a class of transforms that can be expressed in terms of the general relation

$$F(u) = \sum_{x=0}^{N-1} f(x)g(x,u). \qquad (3.1)$$

The function g(x,u) is known as the forward transformation kernel. Also both inverse transforms assume the same form

$$f(x) = \sum_{u=0}^{N-1} F(u)h(x,u) \qquad (3.2)$$

where h(x,u) is the inverse transformation kernel. The nature of the transform is determined by the properties of its transformation kernel. The Walsh transformation kernel consists of a series of global basis functions whose values are +1 or -1 whereas the Fourier transform kernel is based on trigonometric terms.

The Walsh transform can be computed by a fast algorithm identical to the algorithm used to compute the fast Fourier transform. The difference between the two being that the exponential terms in the fast Fourier transform are set at either +1 or -1 for the fast Walsh Transform.

3.3 INITIAL ATTEMPTS TO PROGRAM TRANSPUTERS TO PERFORM THE WALSH TRANSFORM

The fast Walsh transform can be represented as a signal flow graph as depicted in figure 3.1. Expressed in this manner it becomes apparent that the Walsh transform could be viewed as a collection of communicating tasks or processes. This provides an opportunity to implement the transform on a multiple instruction multiple data system such as a transputer network. One possible software implementation would be what is commonly termed a "fine - grained" parallel processing approach.

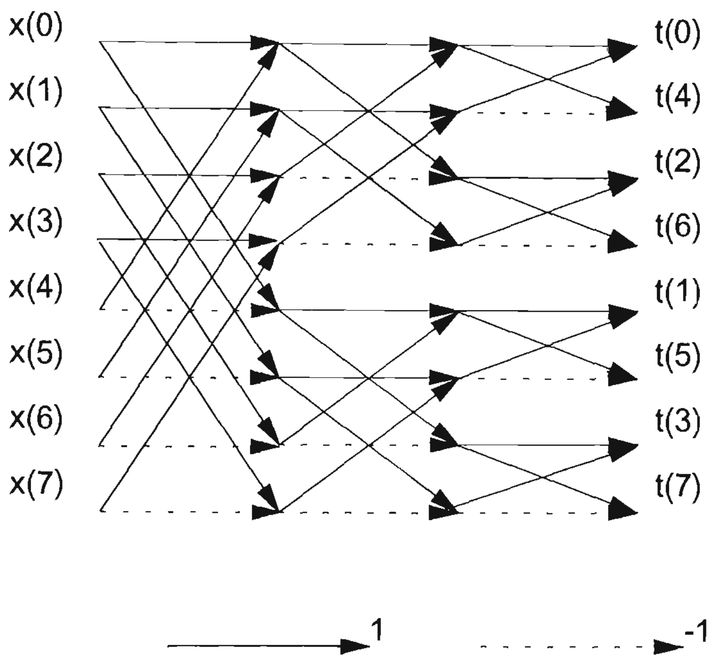


Figure 3.1 A signal flow graph for an eight datapoint Fast Walsh transform.

This method employs a large number of relatively simple software processes. The process interconnection topology takes the same form as the signal flow graph. Each individual process consists of an operation such as single addition or subtraction and the appropriate message passing instructions.

Once a process topology has been determined the interconnected processes must be mapped onto the available transputer system. A PC-transputer implementation to perform the transformation given in figure 3.1 is shown in figure 3.2.

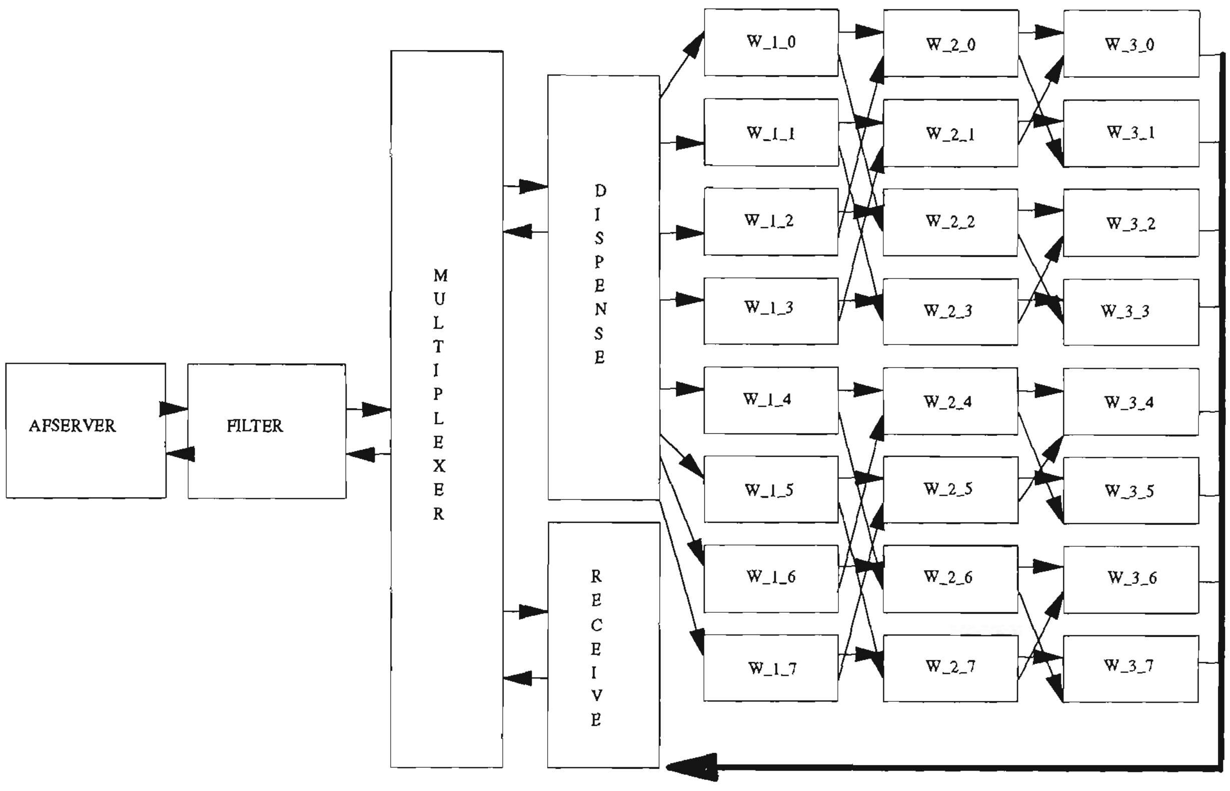


Figure 3.2 Communicating task transputer implementation of the Walsh transform for eight datapoints.

Each of the processes w_p_q receive messages via the channels shown, perform the appropriate addition or subtraction and transmit the result. The 3L Parallel C programming language which was used requires a transputer memory allowance of 5 Kbytes for any process which performs file server I/O. The processes that dispense and receive are interfaced between the library multiplexer process and the butterfly processes in order to obviate a 5 Kbyte memory requirement for each initial and final butterfly process and also to reduce process contention for the PC hard disk when reading and writing data.

This type of implementation can be modified in order to perform on larger datasets and/or larger transputer networks. The transform can be distributed over larger networks by cascading multiplexer tasks as shown in figure 3.3. This allows mapping of the transformation on a range of processor topologies ranging from a few processors to a massively parallel system.

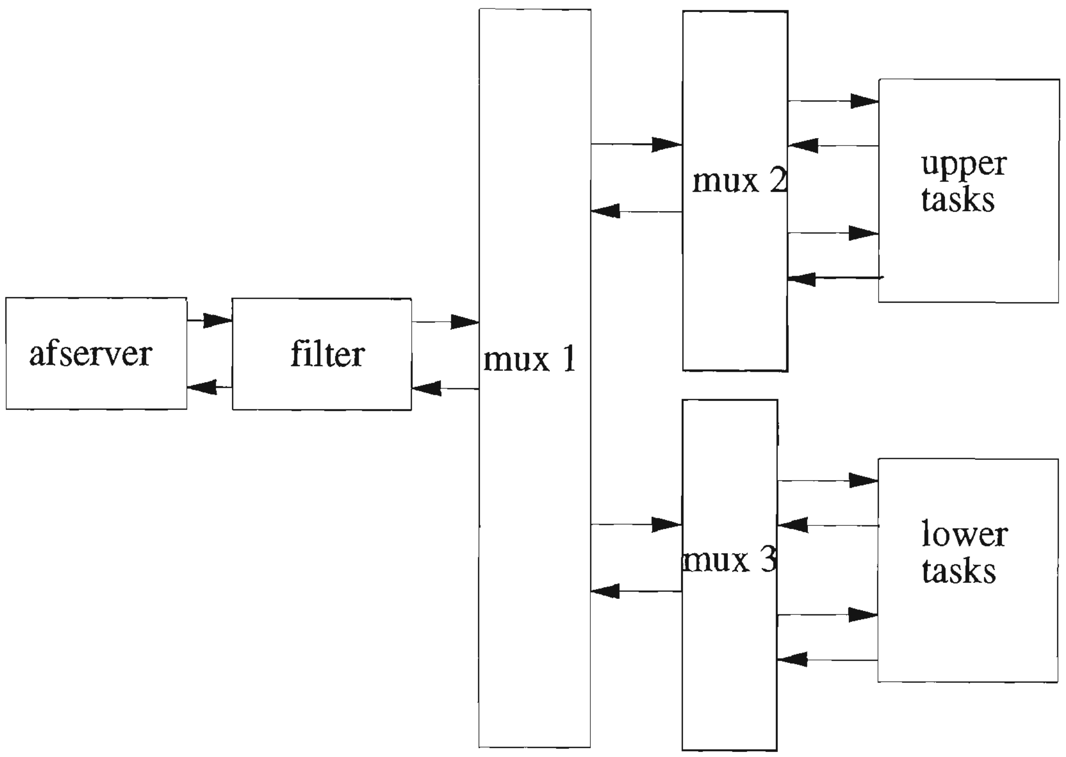


Figure 3.3 Cascading multiplexer tasks.

Butterfly processes can be grouped on transputers with I/O being performed by multiplexer processes. If the transputer network is sufficiently large the network could be configured with the same processor topology as the signal flow graph with each transputer running a single process.

In this implementation the actual butterfly task is generic with the application utilising as many copies as necessary. The connections between the butterfly tasks for datasets of any size can be generated by the following equation.

$$x_n(i) = x_{n-1}(i) + x_{n-1}((j+2^{p-n}) \bmod 2^{p-n+1} + k) \qquad (3.3)$$

where 2^p = datapoints
n = butterfly number
i = datapoint
$j = i \bmod 2^{p-n+1}$
$k = i - j$.

It has been shown earlier in section 1.3.1 that the Walsh transform may also be expressed as a Hadamard matrix of order N. Therefore the Walsh-Hadamard transformation may be performed by means of a matrix multiplication as shown in equation(1.33). The resultant matrix is in bit reversed form. If necessary, it may be ordered by appropriate row transformations of the coefficient matrix. The Walsh-Hadamard form of the transform may be implemented on a transputer network configured as a processor farm.

As discussed in chapter 2 the processor farm implementation consists of a master task placed on the root transputer and a number of identical worker tasks which are placed on all transputers in the network. The master task acquires data containing both the data and basis function matrices from the personal computer mass storage. It then sends the appropriate row and column of the two matrices to a free worker task for processing. The worker task calculates a result which is returned to the master task when the master task is free to receive data. Thus each element in the resultant matrix can be calculated concurrently.

Another possible implementation which was investigated was mapping the fast Walsh transform onto the processor farm. This would require the assignment of the butterfly operation to the worker task and the master task would disseminate the appropriate data pairs and collect the results. It was noted that the worker task butterfly operation would demand little processing time to complete its task while on the other hand the master task would be fully employed determining and transmitting correct data pairs to the worker tasks and receiving and correctly storing results. This would result in the transputer network suffering from a severely unbalanced processing load distribution, this approach was therefore discarded.

3.4 IMPROVED PROGRAMMING TECHNIQUES AND COMPARISON WITH EARLIER METHODS

The two parallel implementations of the Walsh transform discussed above possess both a number of advantages and a number of shortcomings. The advantage of both implementations is, as mentioned earlier, that the transform operation becomes a very easy task to program. However there are a number of shortcomings with both implementations.

The "fine-grained" approach of implementing the butterfly operations as a series of communicating tasks has two drawbacks when mapped on systems consisting of relatively small numbers of processors. First the configuration file discussed in chapter 2 becomes large and cumbersome for datasets of a realistic size. Secondly, for larger datasets a communications bottleneck occurs in the distribution and collection of data by the dispense and receive tasks when each of the independent butterfly tasks compete for access both when transmitting and receiving data. This extends the time spent performing inter-task communications and more than doubles the total processing time.

When implementing the transform as a processor farm it is not viable to perform the fast Walsh transform due to unbalanced processor loads. The other alternative is to perform the transformation as a matrix multiplication, this distributes the processing load more evenly but at the cost of performing N^2 operations rather than the $N\log_2N$ operations of the fast Walsh transform. There is also the problem of communications. The processor farm topology usually consists of a pipeline of processors with the root or PC-interface transputer at one end of the pipeline. It has been found (Webber[75]) that processor farms suffer from a communications bottleneck between the root transputer and more distant transputers attached to the pipeline. This is because distant transputers are required to communicate with the root transputer via all the intermediary transputers.

Any improved transform computation would have to overcome the shortcomings of the implementations given earlier. It would have to be comparatively free of communications bottlenecks and require a compact and straightforward configuration file. Also the processing load on the transputers in the system should be evenly balanced.

The mesh and hypercube processor topologies meet these initial criteria, which of these two is implemented for a particular application is dependant upon a number of other considerations. The mesh topology is often employed in SIMD architectures and is commonly used to perform matrix multiplications (Thinking Machines[71]). It may also be profitably employed using a MIMD architecture.

A study of the process of distributing data over a mesh topology revealed that the number of steps required to distribute or collect data in a MIMD $n \times m$ mesh processor topology from a single data storage source is given by

$$S_{n,m} = \begin{cases} 2\left\lceil \frac{n}{2} \right\rceil + \left\lceil \frac{m-n}{2} \right\rceil & m \geq n,\ n \geq 2 \\ 2\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{m-n}{2} \right\rceil & m \geq n,\ n = 1 \end{cases} \qquad (3.4)$$

where the data storage source is connected to a processor whose communication links are unconstrained by the network topology.

The processor mesh naturally lends itself to matrix operations as the processor topology matches the structure of the data. Performing the transform operation on a mesh as a matrix multiplication therefore becomes an easy task to implement, while suffering from the disadvantage of requiring more operations to perform the matrix multiplication as opposed to the fast transform method. Whilst this in itself is a disadvantage it may be counterbalanced by other application specific considerations. For example, all discrete transforms may be represented as matrix multiplications while methods of performing fast transforms vary widely. Applications which require the calculation of a number of different types of transforms may be better served by a mesh topology matrix multiplication algorithm, its general purpose nature offsetting any loss of performance when calculating a particular transformation.

The hypercube is another processor topology which presents a number of benefits when attempting to implement a transform in parallel. Using a hypercube, it may be shown that access to a mass storage device from the most distant processor in the network can be achieved in D steps where D is the dimension of the hypercube. A comparison of the abilities of the two topologies to distribute data is given in
figure 3.4.

Number of Processors	Mesh size n x m	Maximum required communication steps	
		Mesh	Hypercube
4	2x2	2	2
8	2x4	3	3
16	4x4	4	4
16	2x8	5	4
32	4x8	6	5
32	2x16	9	5
64	8x8	8	6
64	4x16	10	6
64	2x32	17	6

Figure 3.4 Table of inter-processor communication steps required for mesh and hypercube processor topologies.

The results in figure 3.4 illustrate that for processor configurations which contain more than 16 processors the hypercube is more efficient at distributing and collecting data for a message passing MIMD processor network. The reason for this is that for 16 processors or less, the number of communications links per processor utilised by both topologies is the same. For larger numbers of processors the hypercube generates additional communications links while the mesh processors are limited to a maximum of four. The T800 series of transputers is limited to four processor links while the T9000 has six, this limits a straightforward processor network to a six dimension hypercube. If necessary additional transputers could be used to provide communications fan-out given a processor rich environment.

An additional point of interest which may be discerned from figure 3.4 is that a mesh topology data distribution is more efficient if the mesh is maintained in a configuration as square as possible. An illustration of data distribution in mesh and hypercube processor networks can be seen in figure 3.5.

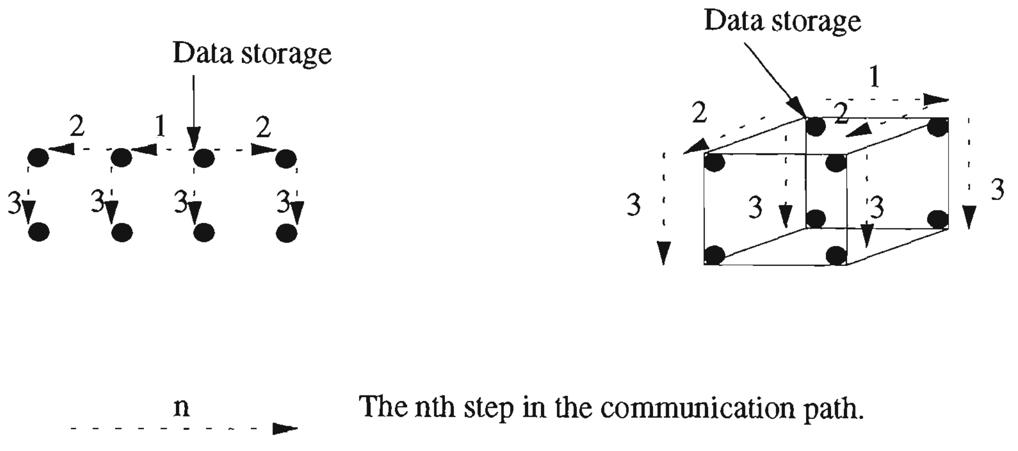


Figure 3.5 An example of two possible data communication paths for eight processor mesh and hypercube processor topologies.

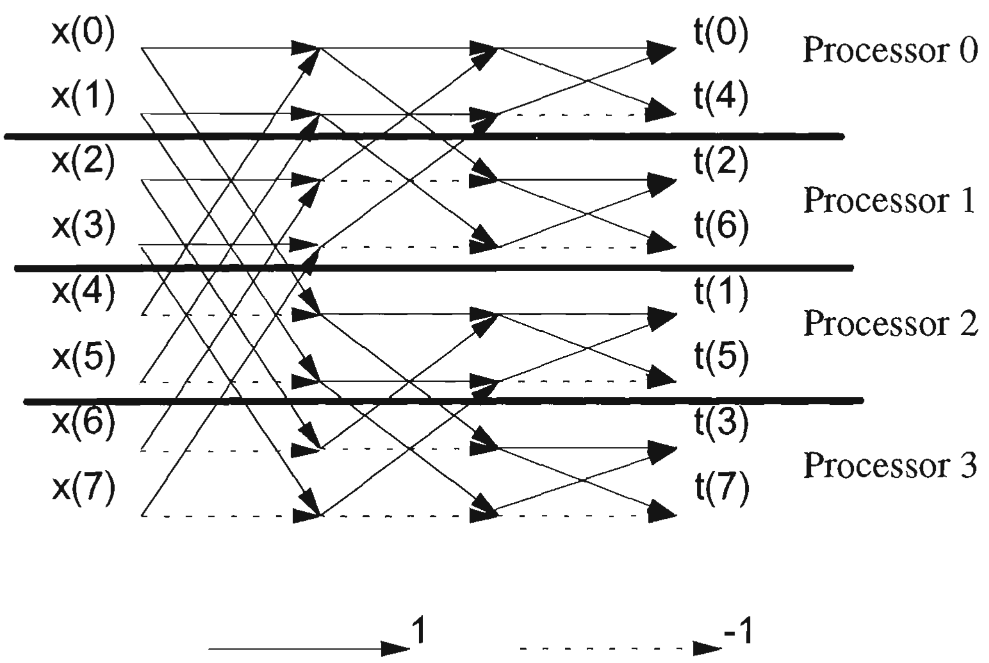


Figure 3.6 The distribution of the fast Walsh transform for eight datapoints on a four processor 2-dimensional hypercube.

The hypercube also possesses the advantage that fast algorithms for global transformations such as the fast Walsh and Fourier transforms can be easily performed in parallel on a hypercube processor topology. Figure 3.6 demonstrates a mapping of the fast Walsh transform onto a two dimensional hypercube processor network.

It can be seen from figure 3.7 that if any inter processor communications are required to perform the fast transform all the communications are between nearest neighbour processors. This may be maintained for higher dimension hypercubes by ensuring that the processor addresses as given in figure 3.6 have a binary address one bit different from all other nearest neighbours (Hillis[33]). Such a distribution of the fast transform on hypercube processors allows for a high degree of parallelism in the computation.

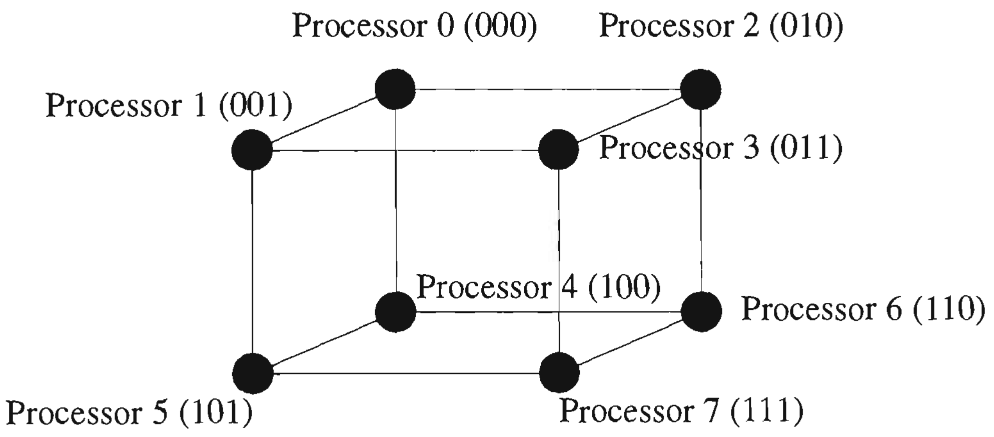


Figure 3.7 Hypercube processor numbering system, decimal and binary representations.

As the major interest of this study was the performance enhancement of transform operations the hypercube was considered the most suitable processor configuration for the implementation of the fast Walsh transform. It avoids the configuration complexities and the communications bottlenecks of the earlier implementations and while the programming is more complex the processor load is balanced and efficient parallelisation of the fast transform is maximised.

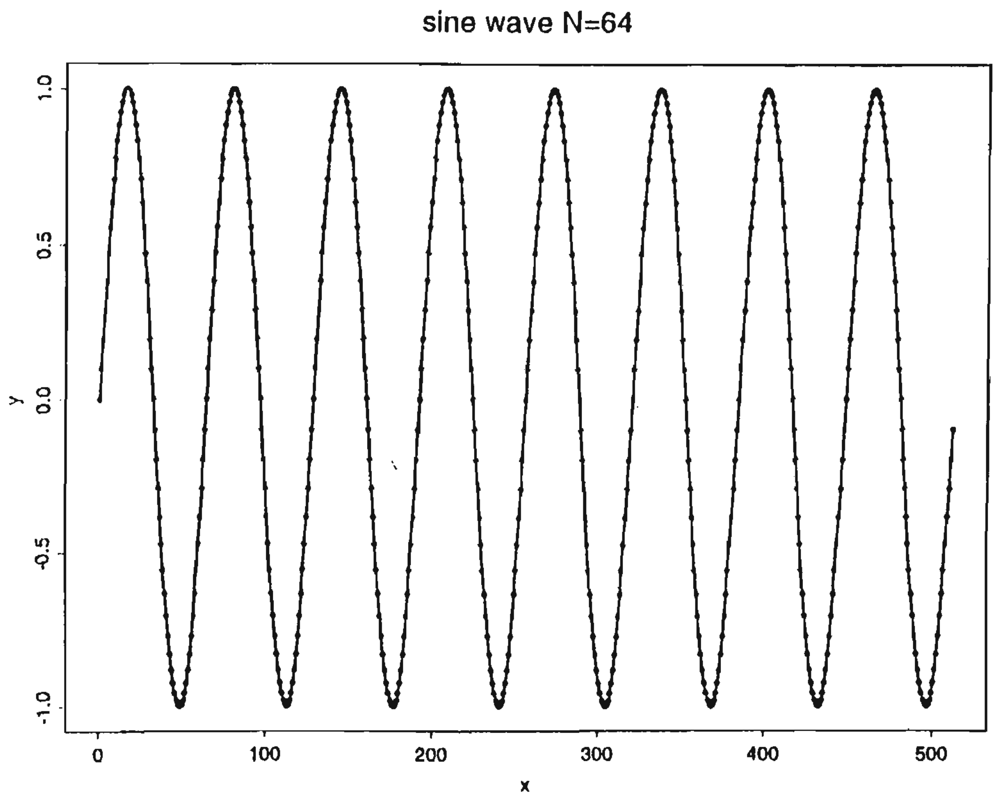
3.5 APPLICATION OF THE WALSH TRANSFORM TO PERIODIC AND NON-PERIODIC FUNCTIONS

The Walsh transform was performed on a number of waveforms in order to determine its ability to distinguish a number of different periodic or localised features in a one dimensional signal. The waveforms selected tested the ability of the Walsh transform to detect smooth and discontinuous periodic functions, smooth non-periodic functions and time localised functions. The test waveforms and their transforms are given in figures 3.8-3.18, more detailed representations can be found in appendix C.

The trigonometric functions were used as examples of smooth periodic waveforms. Their Walsh spectrum is clearly recognisable with a dominant frequency component. The strong harmonics close to the dominant frequency peak are caused by the higher frequency square wave basis functions in the series which are required to smooth the dominant frequency square wave into a more sinusoidal shape. Higher frequency trigonometric functions cause a shift of the spectrum into higher frequency ranges.

The step and chirp functions are representative of periodic discontinuous functions. The step function transforms show that periodic square functions are easily distinguishable due to their close correspondence to the Walsh basis functions. The smaller frequency components in the spectrum are generated by the variation between the initial value of the waveforms and that of the basis functions.

The ability of the Walsh transform to provide distinctive spectra degrades when the periodic function waveform shape departs significantly from the basis function set. An example of this is the chirp function transform. If the discontinuities in the waveform are not present in the basis function set then a large number of spectral components will be generated, representing the basis function series terms needed to approximate the waveform discontinuities. This results in a "noisy" spectrum as evidenced by the chirp function transform.



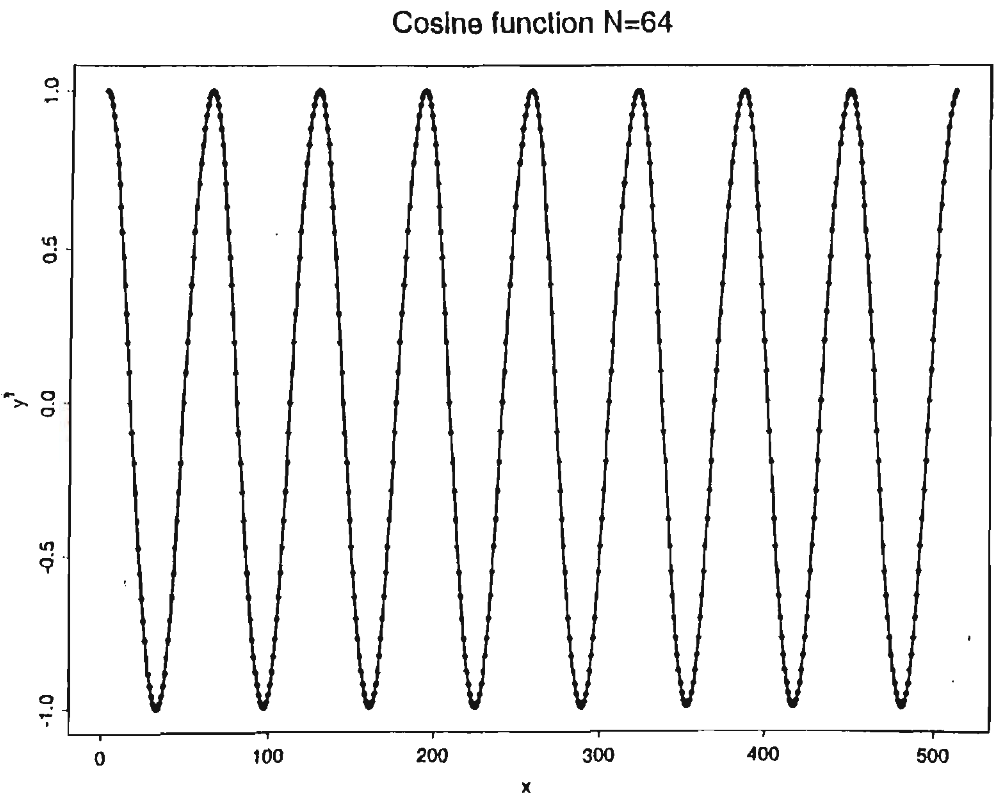
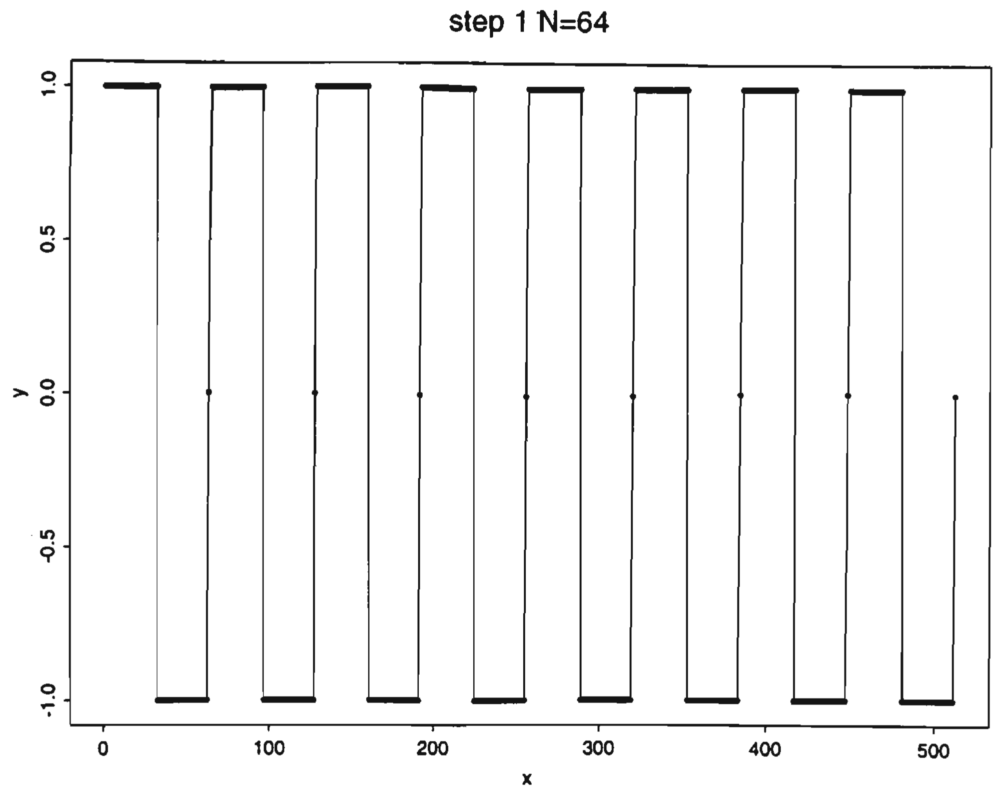


Figure 3.8 Trigonometric test functions.



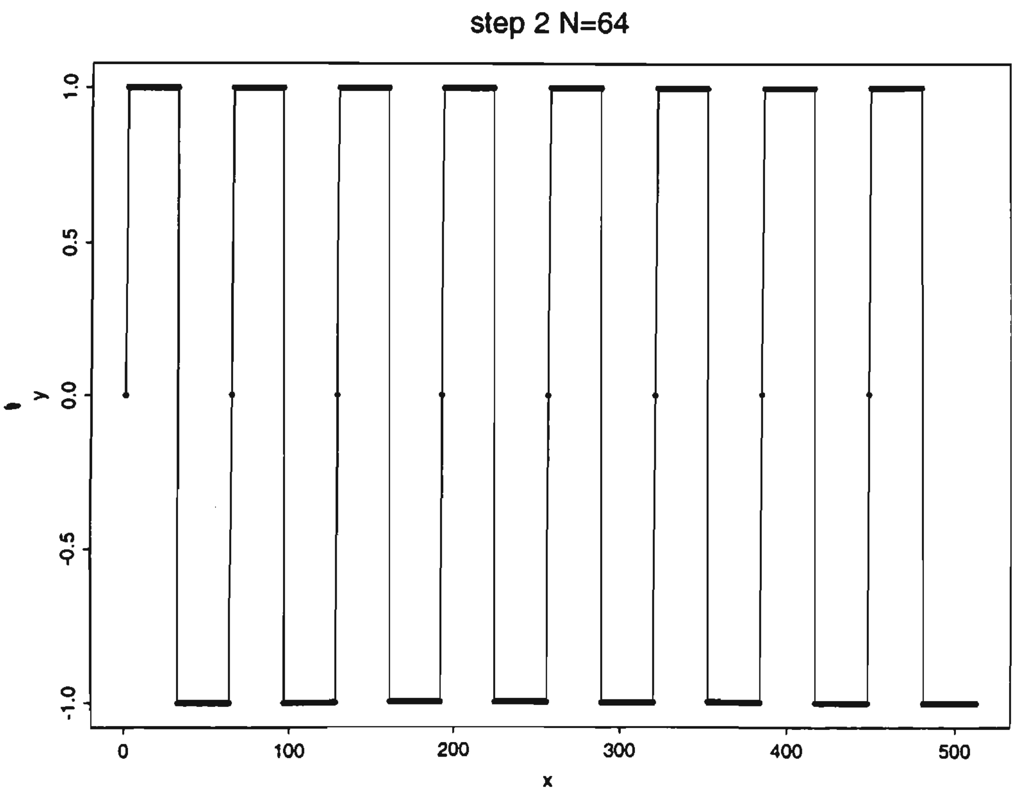
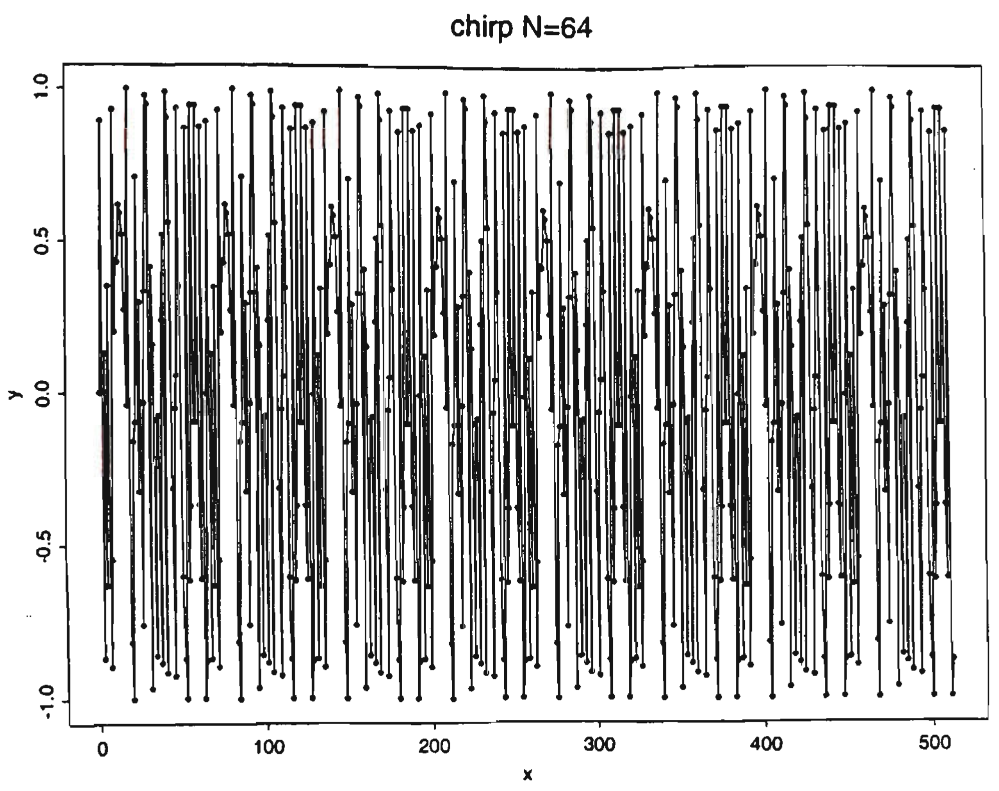


Figure 3.9 Step test functions.



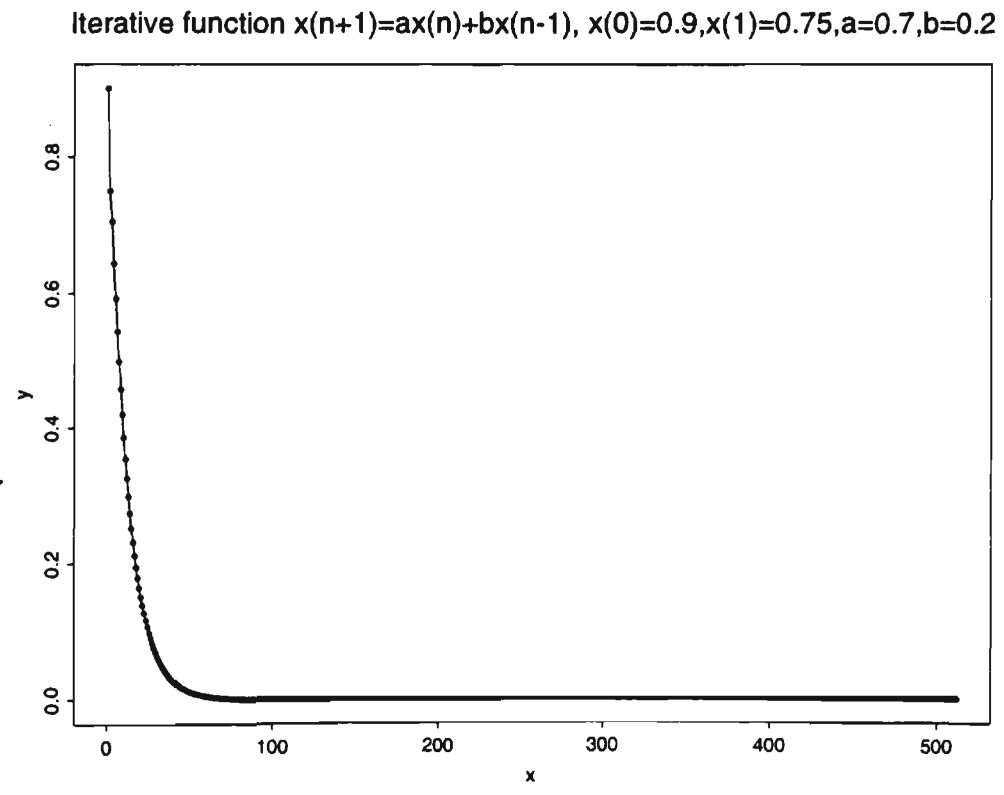
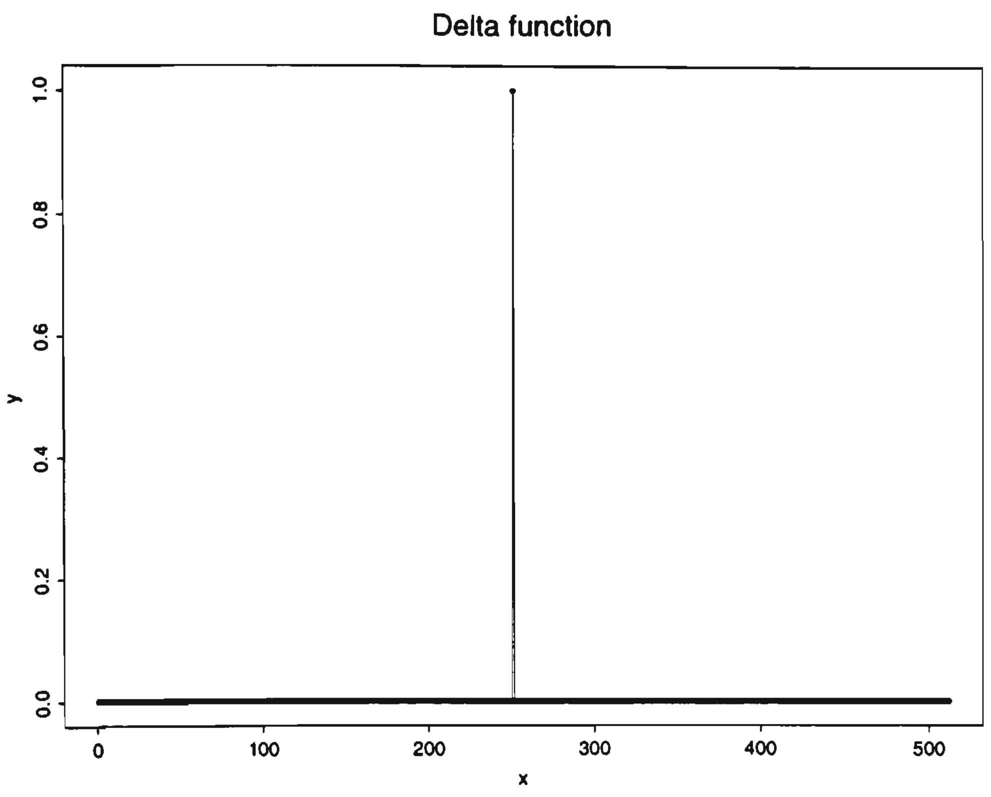


Figure 3.10 Chirp and Iteration test functions.



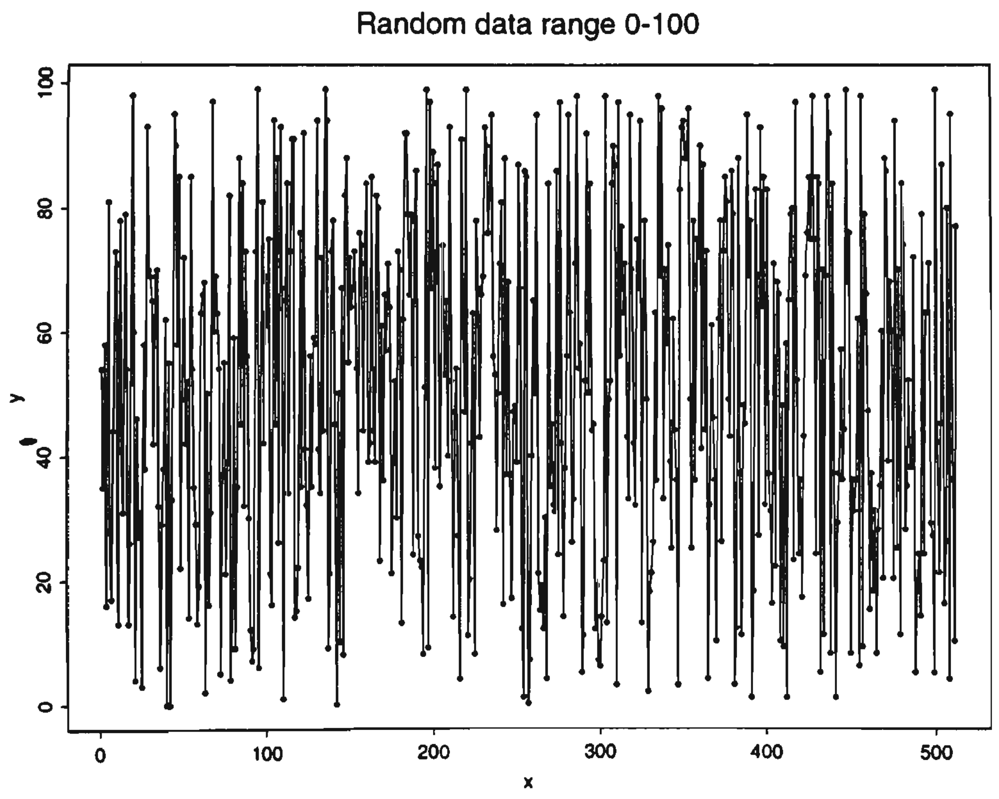
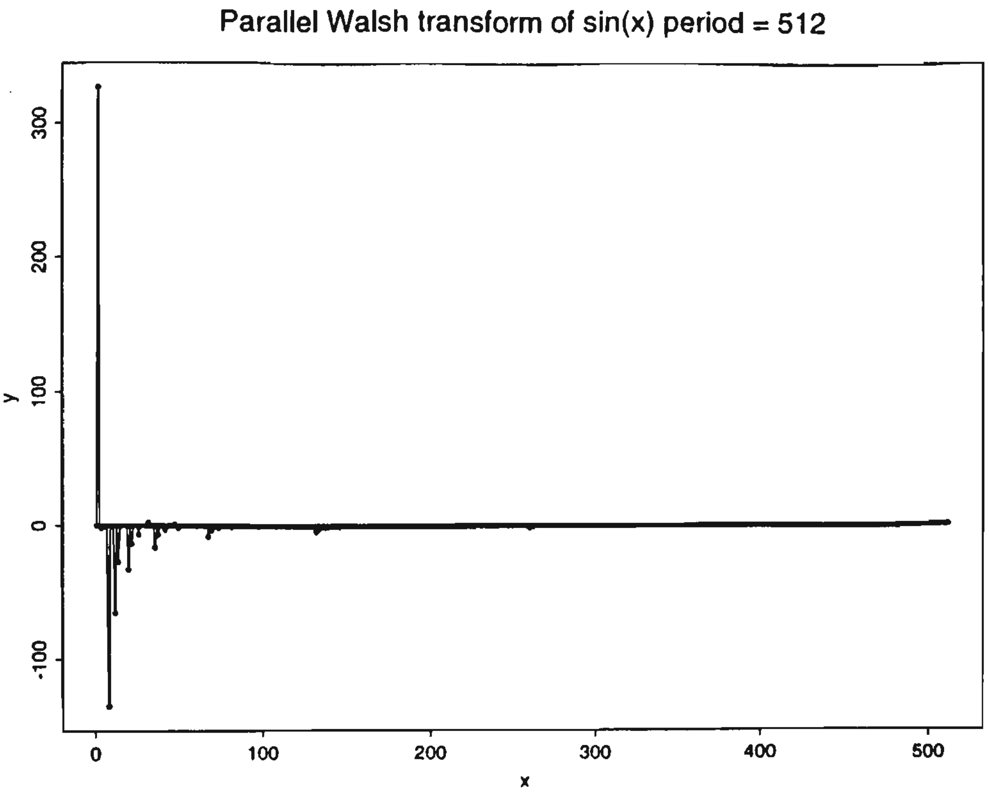


Figure 3.11 Delta and Random test functions.



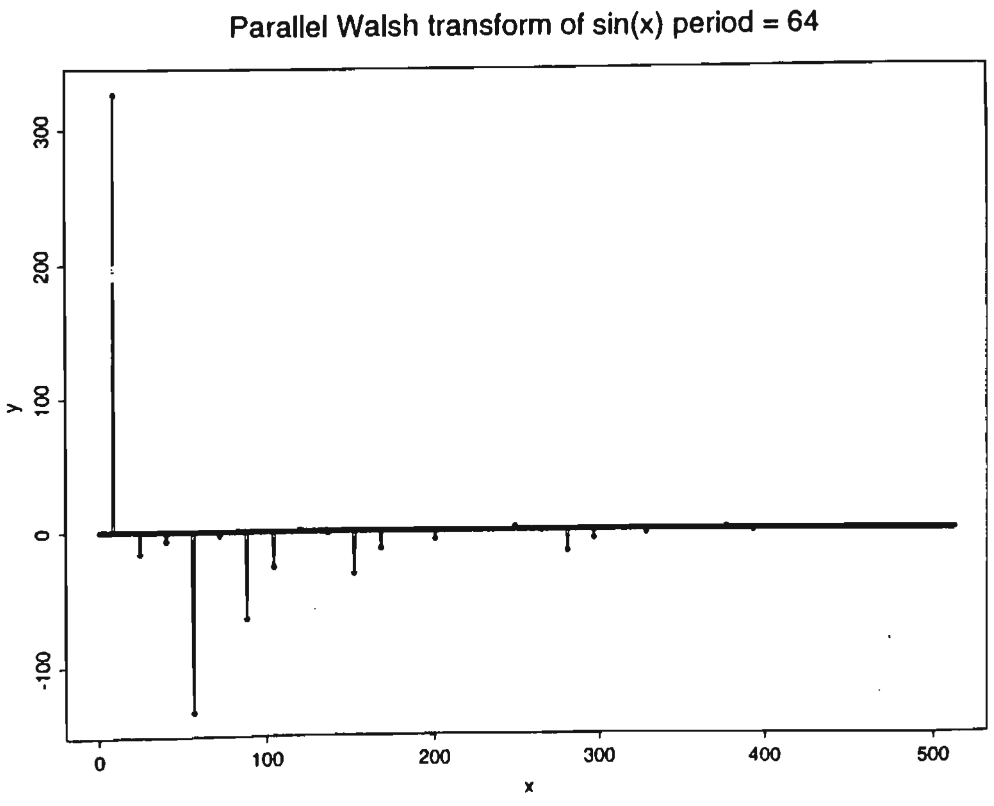
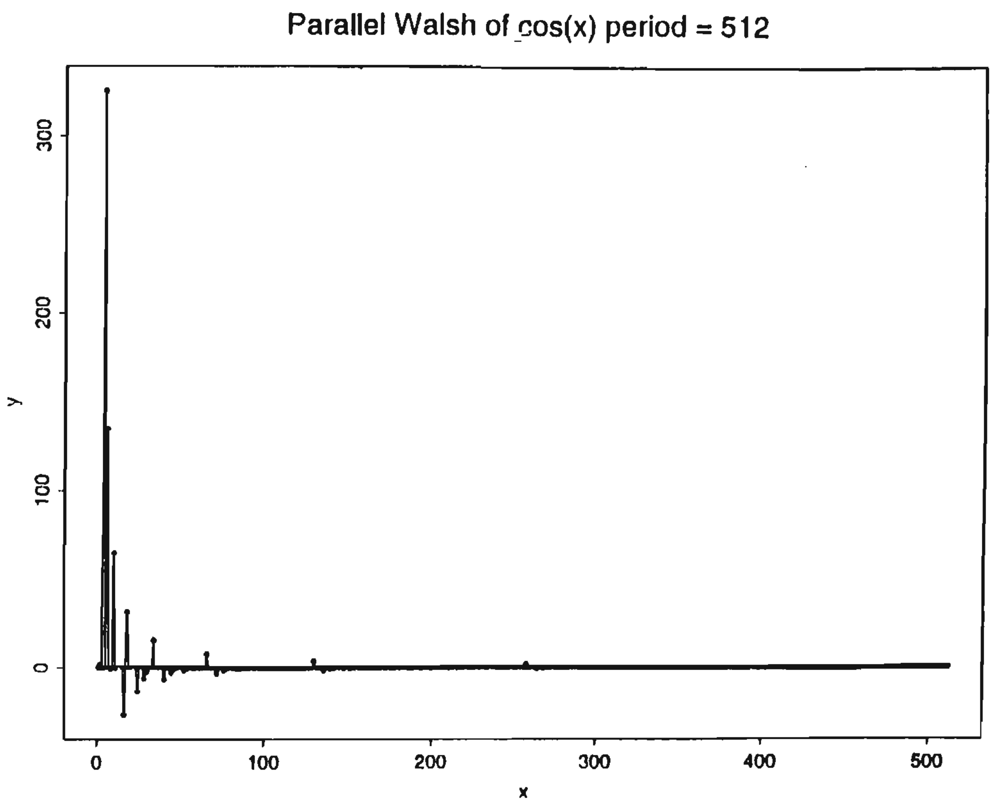


Figure 3.12 Two examples of the discrete Walsh transform of the sine function where x = sequency, zero crossings per unit time (Zps) and y = transform coefficient magnitudes.



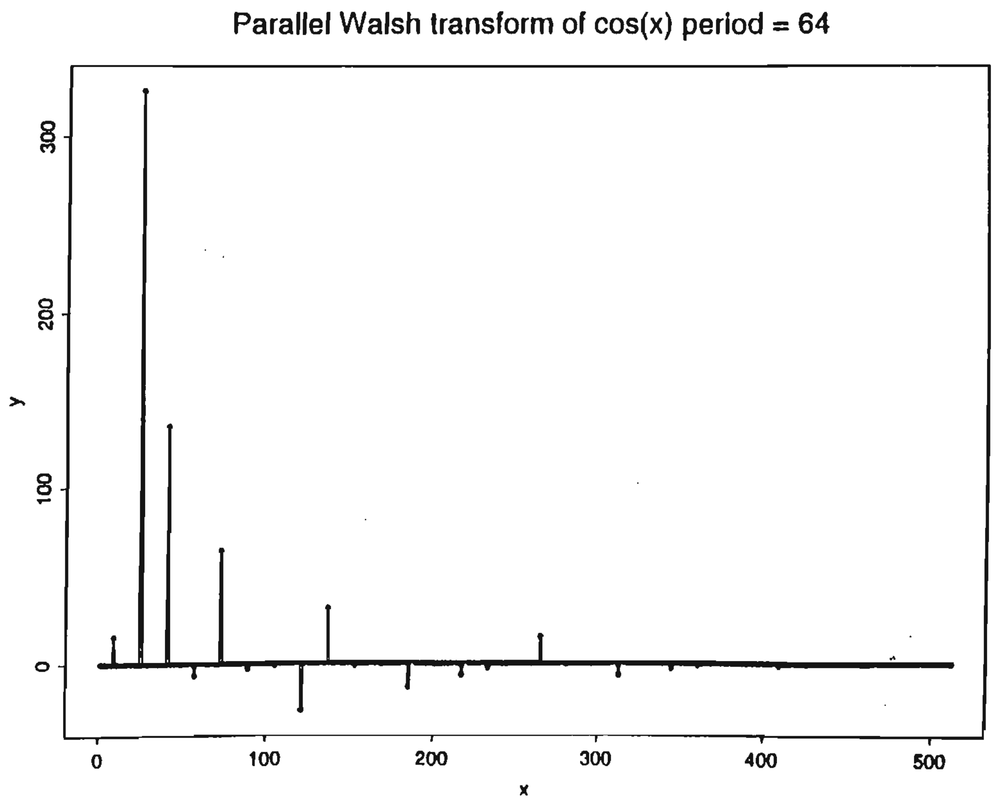
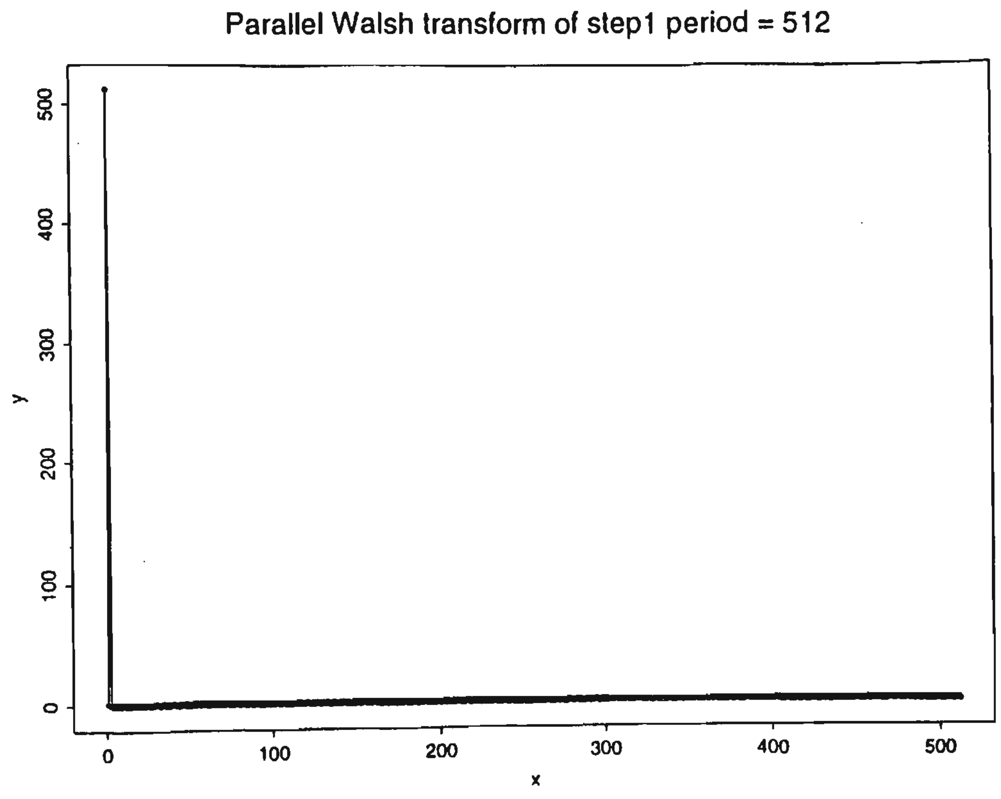


Figure 3.13 Two examples of the discrete Walsh transform of the cosine function where x = sequency, zero crossings per unit time (Zps) and y = transform coefficient magnitudes.



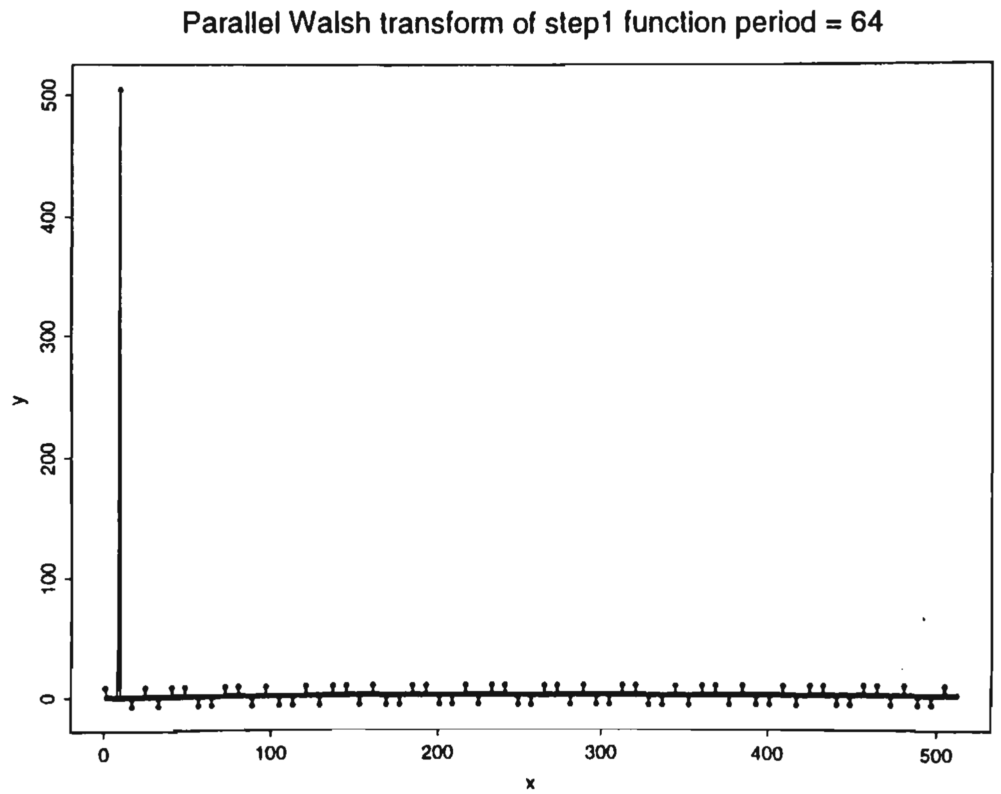
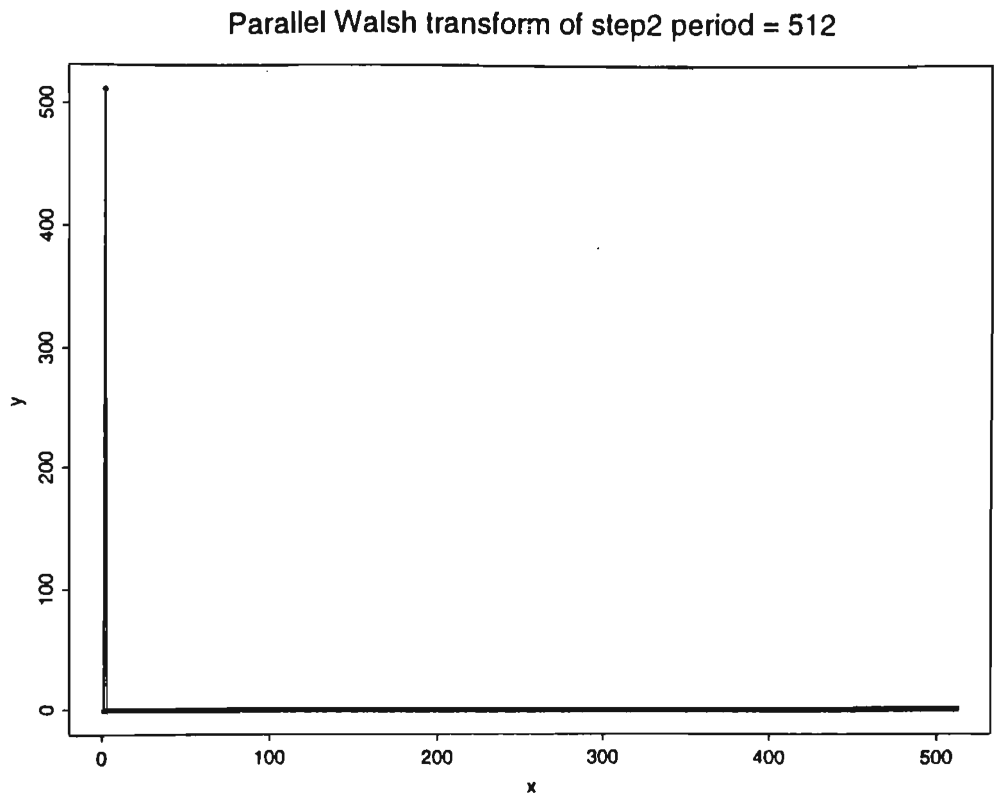


Figure 3.14 Two examples of the discrete Walsh transform of the Step 1 function where x = sequency, zero crossings per unit time (Zps) and y = transform coefficient magnitudes.



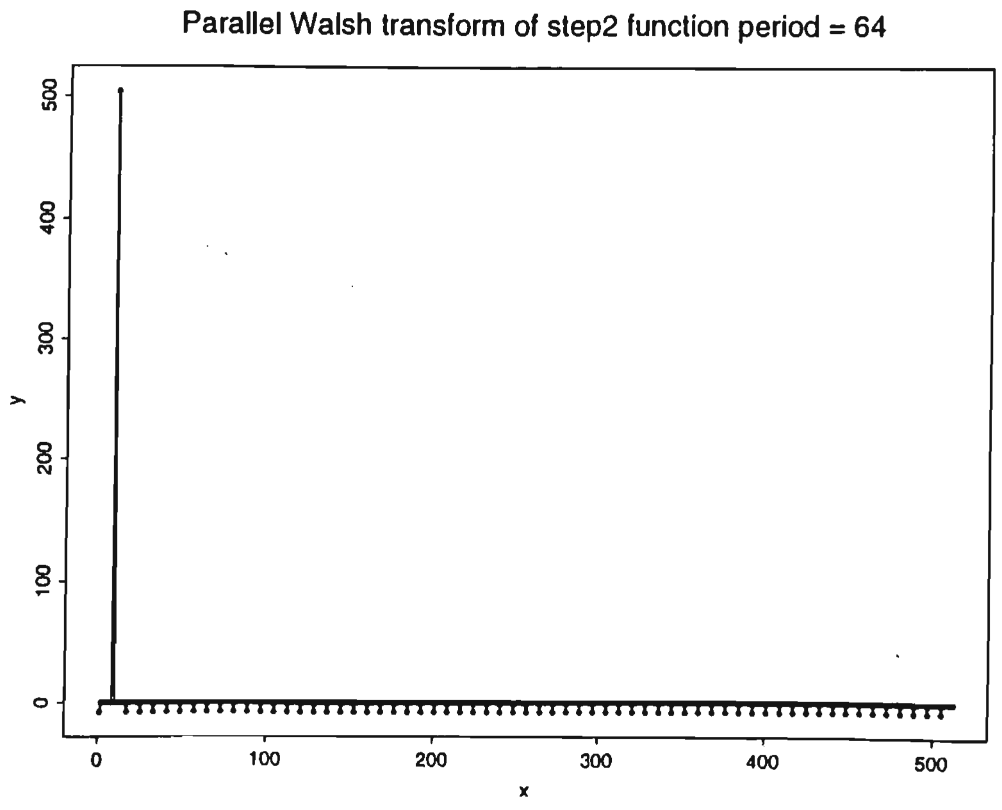
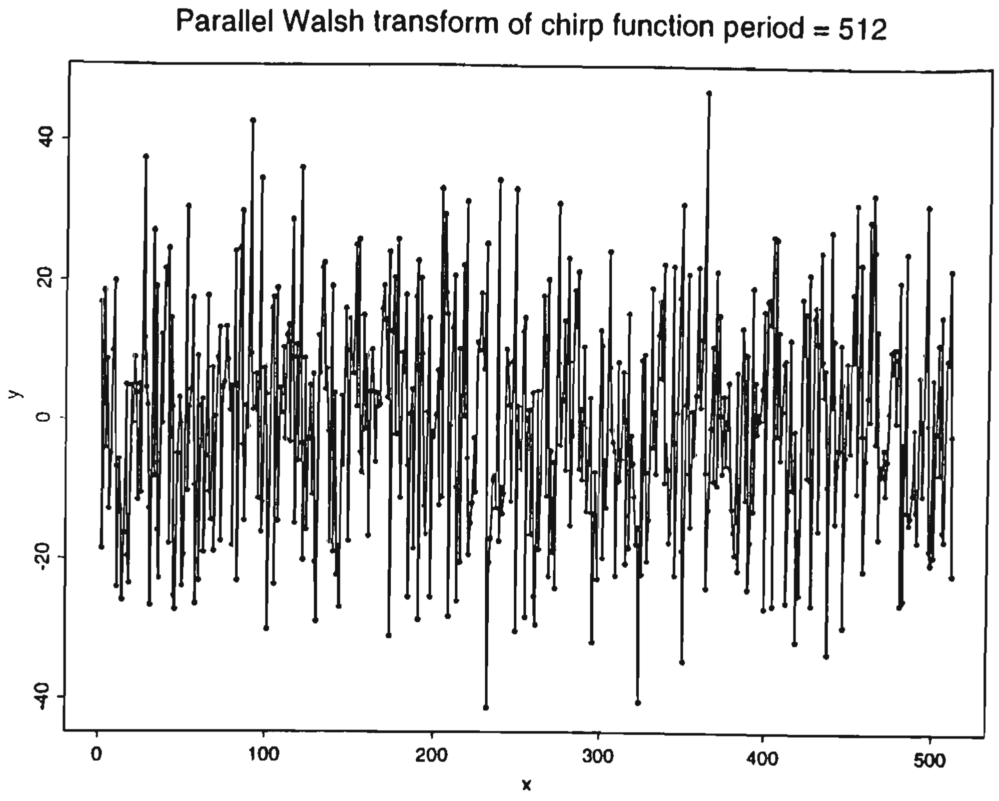


Figure 3.15 Two examples of the discrete Walsh transform of the Step 2 function where x = sequency, zero crossings per unit time (Zps) and y = transform coefficient magnitudes.



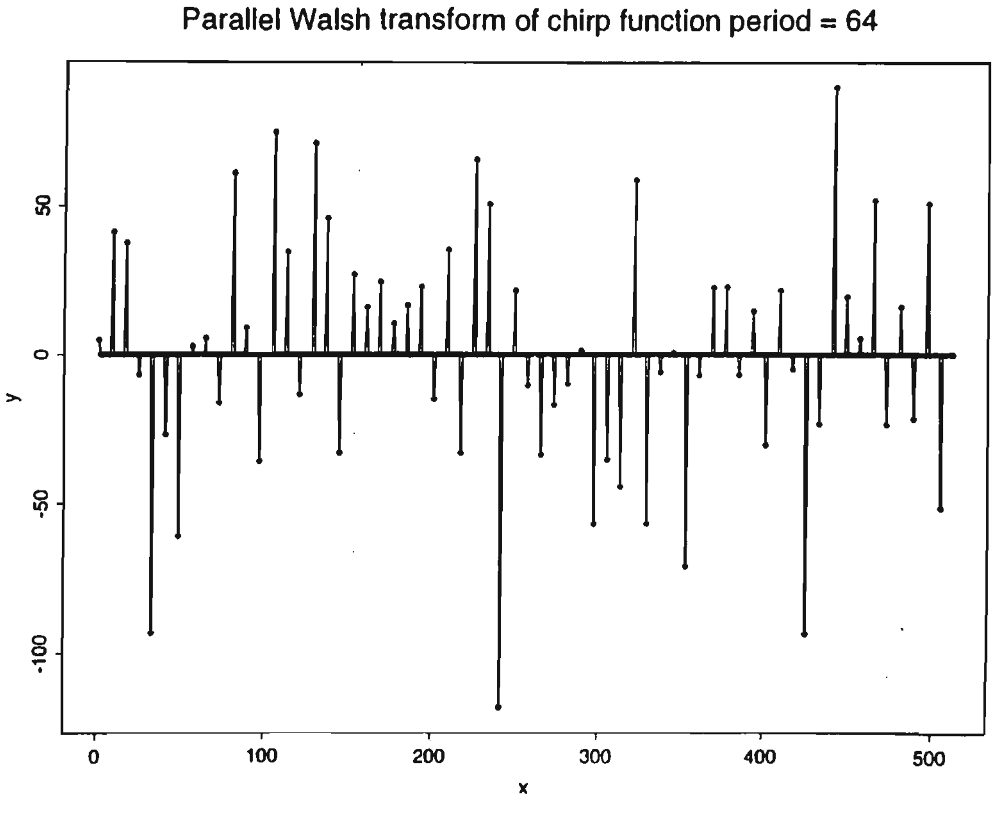
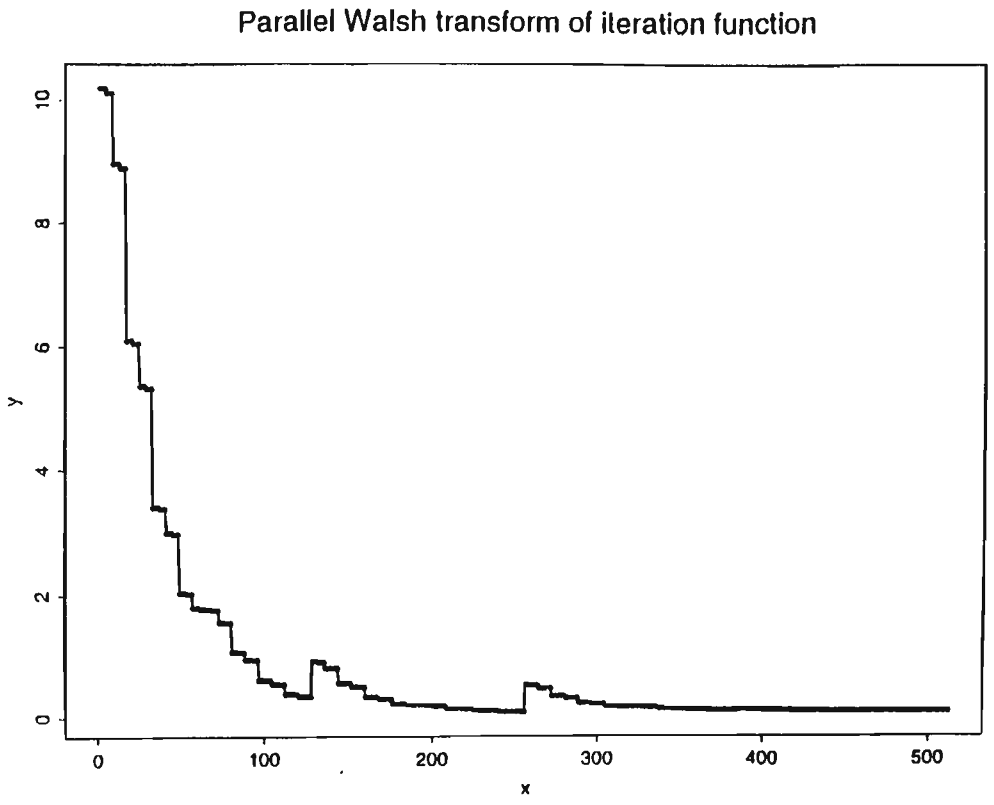


Figure 3.16 Two examples of the discrete Walsh transform of the Chirp function where x = sequency, zero crossings per unit time (Zps) and y = transform coefficient magnitudes.



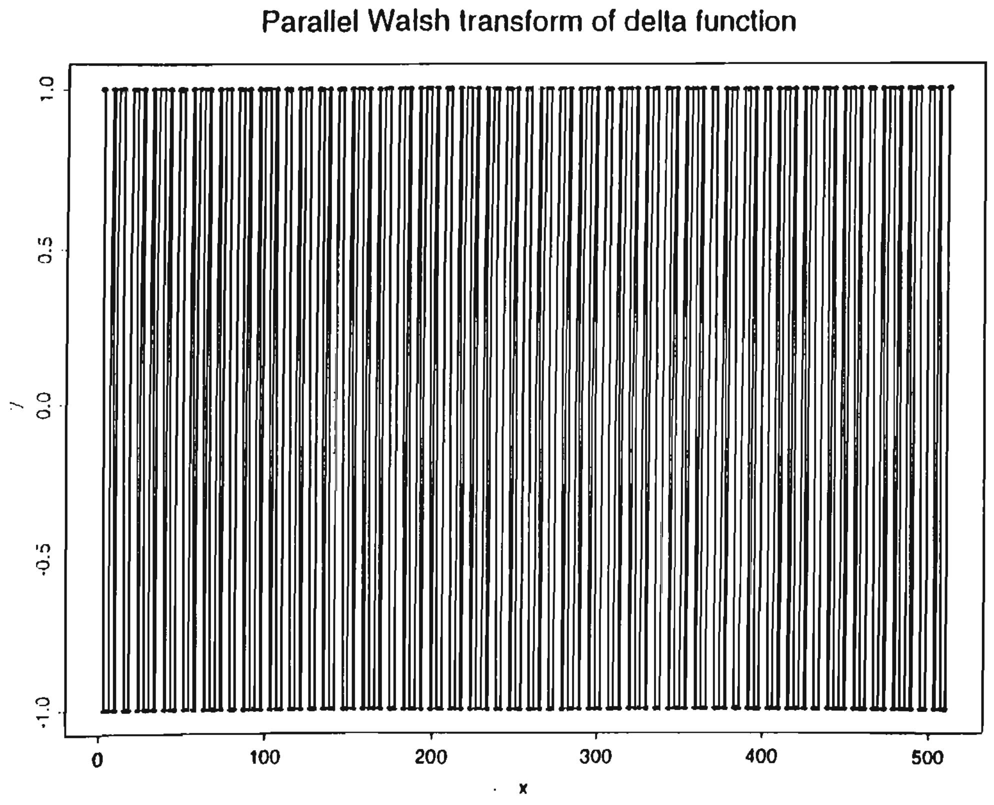


Figure 3.17 Examples of the discrete Walsh transform of the Iteration and Delta functions where x = sequency, zero crossings per unit time (Zps) and y = transform coefficient magnitudes.

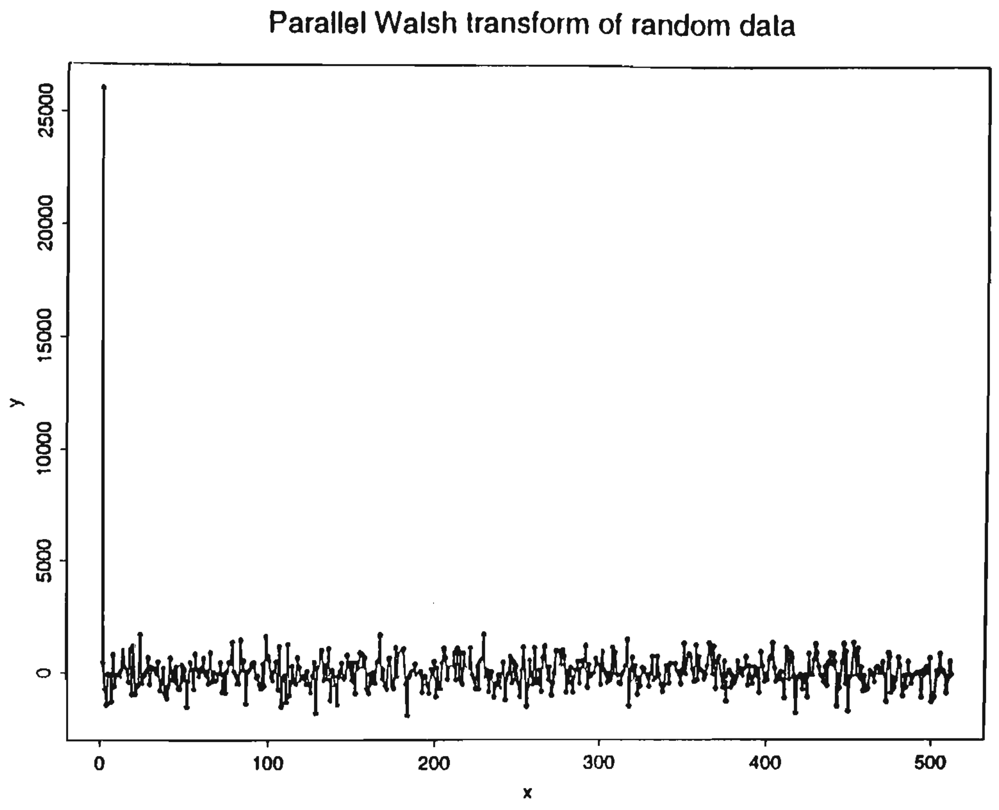


Figure 3.18 The discrete Walsh transform of the Random data function where x = sequency, zero crossings per unit time (Zps) and y = transform coefficient magnitudes.

The iteration and random noise functions are examples of non-periodic smooth and discontinuous functions. The sharp signal fluctuations in the random noise waveform result in a spectrum with properties similar to the chirp function. The large initial spectral component is caused by the random noise data values being restricted to a range between zero and one hundred. The coefficients used by the iteration function resulted in a smoothly decaying non-periodic waveform. Consequently the spectrum exhibits a number of low frequency components dying out in the higher frequency range.

A transformation of the delta function was performed to observe the Walsh transform of a function which is localised in time. The Walsh transform of the delta function produces a complex spectrum with all basis function components represented. Movement of the waveform pulse in time causes only minor variations in its Walsh transform. Detection of time localised signals using the Walsh transform would be a difficult task due to the number of basis function coefficients present in the spectrum and the small variation in the transform caused by the timing of the transient signal.

These results indicate that the Walsh transform is effective when detecting waveforms similar to its basis function set but that this effectiveness declines as sharp variations in amplitude and local signal transients are introduced into the waveform.

3.6 COMPARISON OF SERIAL AND PARALLEL IMPLEMENTATIONS OF THE WALSH TRANSFORM

The comparison of the serial and parallel implementations of the Walsh transform can be divided into two categories. Firstly a comparison of the Walsh transform performed serially on one transputer against both the hypercube and processor farm parallel implementations on a multi-transputer system. Secondly a comparison between the transputer implementations and current commercial microprocessors. The results given in figure 3.19 demonstrate that implementing the fast Walsh transform on a two transputer system approximately halves the total processing time required when compared with the single transputer implementation. Splitting the calculation between two processors should theoretically result in a halving of the required calculation time. The deviation from this figure by approximately 2% is due to the time required to communicate data between the processors as shown figure 3.20.

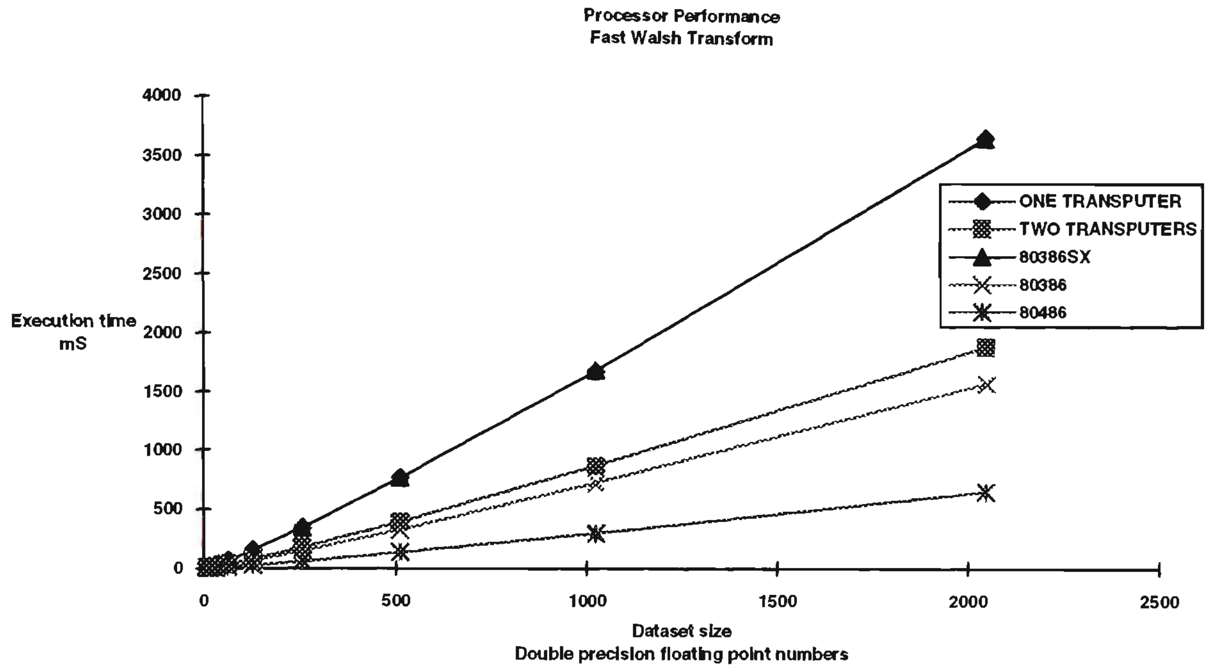


Figure 3.19 Serial and parallel processor performance when implementing the fast Walsh transform.

A factor that must be considered when implementing an algorithm using the MIMD processing model is the communication overhead incurred when disseminating and collecting data over the processor network. The results in figure 3.20 show that a large percentage of total processing time is spent performing transform calculations with very little communications overhead. In the two transputer implementation the transmission of the entire dataset between the two processors was necessary, for larger hypercube systems the communication overheads will not be significantly greater due to the connectivity of the hypercube. As shown in figure 3.4 the number of inter-processor communications required will be N where N is the dimension of the hypercube. Therefore relatively large numbers of processors can be interconnected with minimal communication overhead when transmitting data from or sending data to data storage e.g. a hard disk drive. For example, a seven dimension hypercube consisting of 128 processors will require only seven sets of inter-processor communications to distribute data throughout the network. Further the amount of data which has to be distributed will diminish as communication distance from the point of connection to the data storage increases.

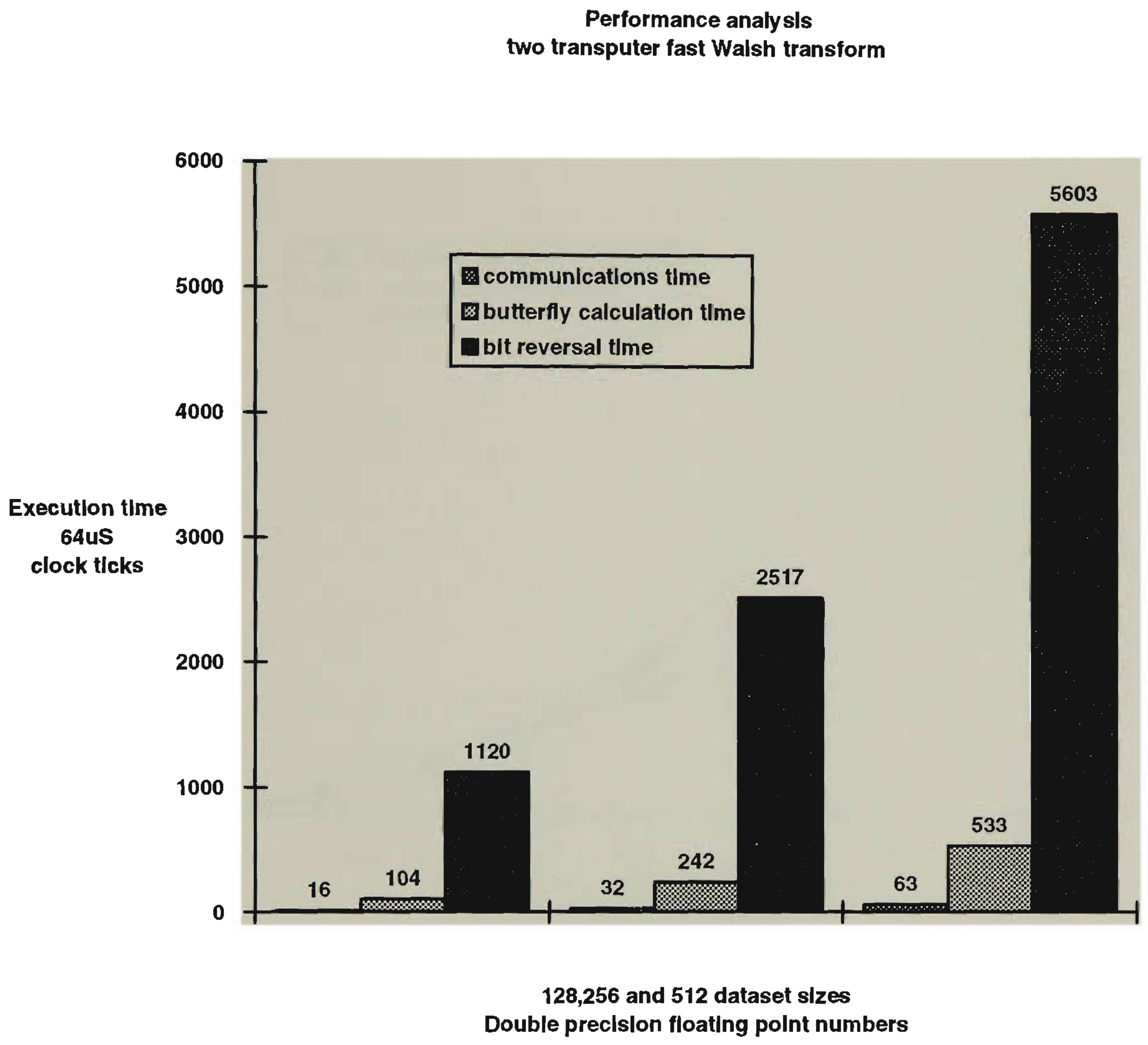


Figure 3.20 An analysis of computational resource demand by major operations within a two transputer fast Walsh transform implementation.

A comparison of the performance of the fast Walsh transform on a two transputer hypercube with a two transputer processor farm implementation of the Walsh-Hadamard matrix form of the transform was also made. The Hadamard matrix used by the processor farm was pre-processed, ordering the matrix rows so that a bit reversal operation was not required. Figure 3.21 indicates that even given the advantage of pre-processing the performance of the processor farm compares unfavourably with the fast transform-hypercube implementation for datasets of 64 numbers or greater, vindicating the choice of the hypercube configuration for fast calculation of the Walsh transform.

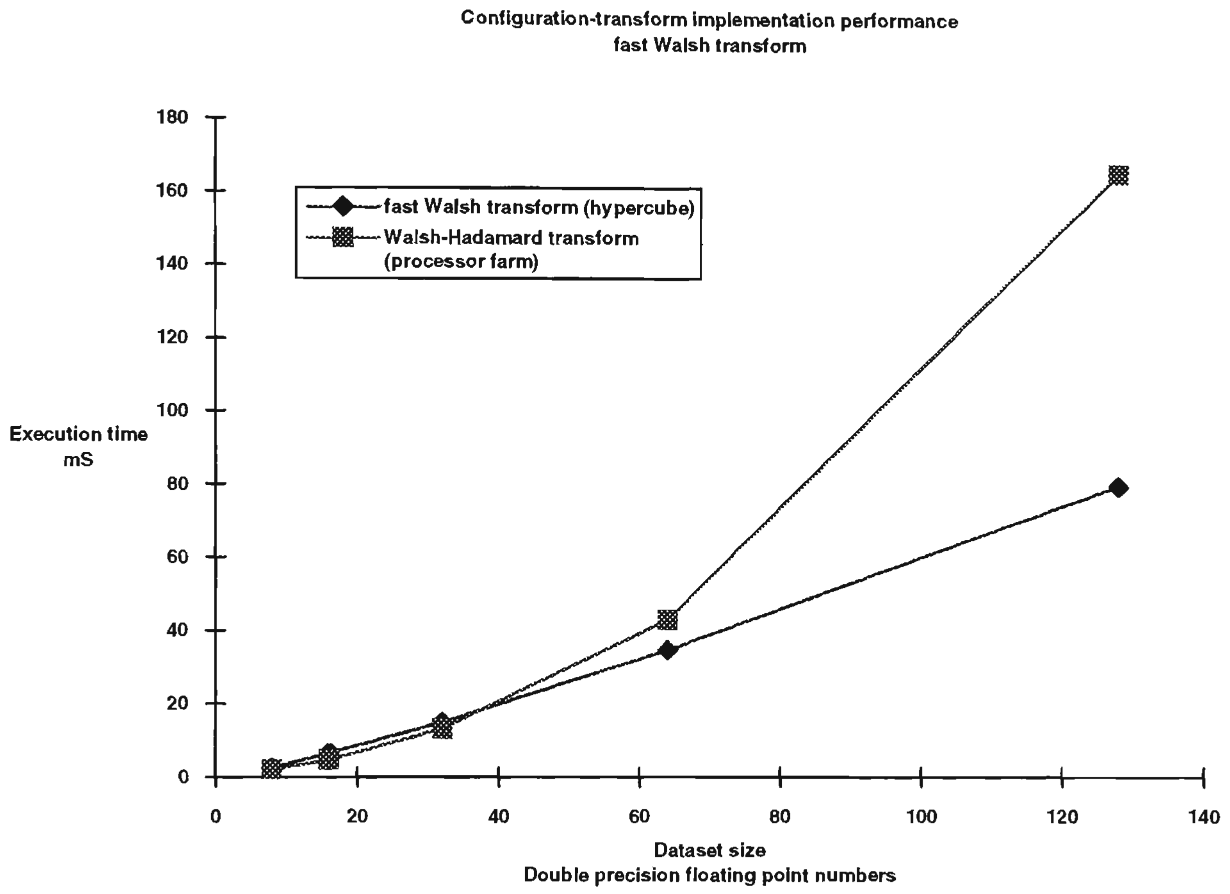


Figure 3.21 Performance comparison of Walsh transform implementations on a two transputer parallel system.

The comparison of the transputer implementations with commercial microprocessors is given in figure 3.19. Neglecting any minor performance effects caused by using different C language compilers, figure 3.19 shows that a serial implementation of the Fast Walsh transform on a single transputer performed at virtually the same speed as a serial implementation on an Intel 80386SX microprocessor. The performance of a two transputer parallel implementation was better than the 80386SX performance and marginally inferior to the Intel 80386 serial implementation. One of the major objectives of parallel processing is to provide a level of performance superior to that available with conventional microprocessors.

This is clearly not the case in the implementation shown above. The reason for this is a historical one. The T800-20 transputers used in these performance measurements were introduced in 1987, making the T800-20 a contemporary of the Intel 80286. By the standards of the day the transputer possessed significant processing capability in its own right as well as being able to be linked into a network of transputers further enhancing performance. The progress of standard microprocessors since that time has rendered the T800 as a stand alone microprocessor obsolete.

There are two possible ways the operation of a transputer system could be improved to provide it with a level of performance better than that demonstrated by the Intel 80486. The first method would be to replace the T800 with the T9000 transputer which is claimed to be significantly faster than the T800 or T805. The second way to improve the performance of the transputer system would be to increase the parallelism of the system by adding more transputers to the system. As shown in figure 3.20 the time required for inter-processor communications when performing the fast Walsh transform is only a small percentage of the total processing time. An extrapolation of the results given in figure 3.19 provide an estimate of the performance enhancement that could be achieved by adding more transputers to the system. The results given in figure 3.19 indicate that a hypercube system comprising eight T800-20 transputers would give a performance better than the Intel 80486 serial implementation.

3.7 DISCRETE COSINE TRANSFORM ALGORITHMS

Due to its widespread use in such areas as image compression (Hein[32]) many algorithms for computing the discrete cosine transform have been developed. As outlined in Chapter 1 the majority of these algorithms can be classified into three categories, those that compute the discrete cosine transform through matrix multiplication or recusive computation (Hou[35],Chen[14],Kou[39],Cho[16b]) and those that compute the cosine transform via another transform such as the fast Fourier transform (Hein[32],Wu[77],Rao[57a]).

The recursive and matrix algorithms demonstrate a performance similar to that of the fast Fourier transform (Chen[14]), but a number of problems arise when parallelisation of these algorithms is attempted. A review of the signal flow diagrams for the algorithms given in the literature shows that given a dataset distributed over a number of processors computation of the discrete cosine transform would involve a large communications overhead owing to the semi-global nature of many of the transform operations. Also the complexity and lack of signal-flow symmetry in many of the transform stages lead to complex programming requirements which have a direct effect on the ease with which any of these algorithms could be scaled onto larger parallel platforms.

Consequently parallel implementations of the discrete cosine transform have favoured implementation via other more amenable transforms (Cho[16a]). Such an algorithm is outlined in Rao[57a]. The cosine transform for N data points is obtained via an N point fast Fourier transformation. This algorithm requires only a minor modification of the fast Fourier transform in order to compute the discrete cosine transform. As the fast Fourier transform has been shown to be easily parallelizable the parallel computation of the cosine transform using the fast Fourier transform provides all the advantages of the parallel implementation of the fast Fourier transform at the minor cost of the increased processing time required to convert the Fourier transform to the cosine transform.

3.8 A PARALLEL DISCRETE COSINE TRANSFORM ALGORITHM

The algorithm for the discrete cosine transform via the fast Fourier transform developed by Narashima and Peterson (Rao[57a]) is shown in figure 3.22. The algorithm sorts the N input data into a sequence given by equation (3.5).

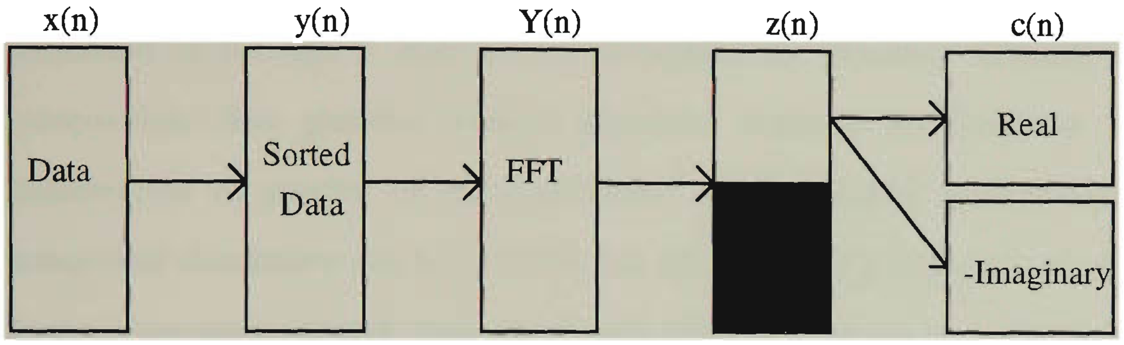


Figure 3.22 N-Point discrete cosine transform via the N-point fast Fourier transform.

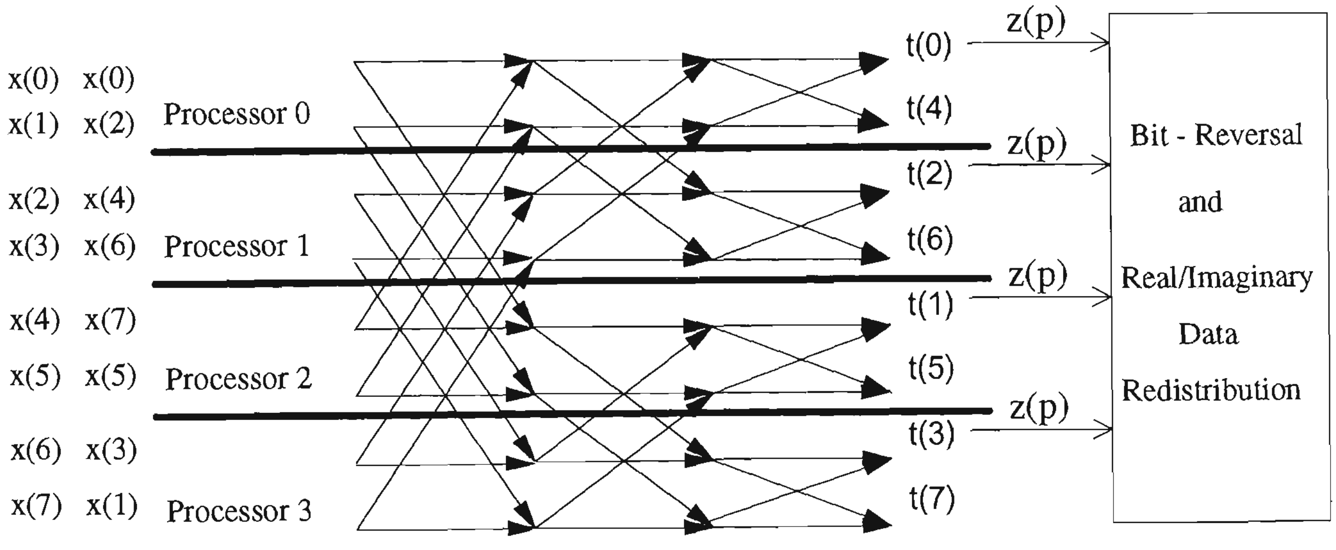
$$\begin{aligned} &y(n) = x(2n) && n = 0,\ldots,\frac{N}{2}-1 \\ &y(N-1-n) = x(2n+1) && (3.5) \\ &N = \text{Dataset.} \end{aligned}$$

A fast Fourier transformation is then performed on the rearranged data. The first $\frac{N}{2}$ transform results are multiplied by a complex constant given by equation 3.6

$$z(n) = e^{-j\frac{n\pi}{2N}}. \qquad (3.6)$$

The resulting $\frac{N}{2}$ real components provide the discrete cosine transform coefficients from 0 - $\frac{N}{2}$ and the negative imaginary components provide the cosine transform coefficients from $\frac{N}{2}$ - N.

The kernel of this transform algorithm is the fast Fourier transform. It has been shown earlier in this chapter that a hypercube processor topology provides an efficient and well proven parallel implementation of the fast Fourier transform. It would seem appropriate therefore to use this topology to implement the discrete cosine transform. The additional computational tasks of sorting the input data can be performed as the data is disseminated throughout the processor network prior to computation. The post-fast Fourier transform constant multiplication can be implemented in parallel on the hypercube. The bit-reversal and real-imaginary component distribution can be performed as the transform coefficients are retrieved from the processor network. An example of these operations is given in figure 3.23.



p = m (bit reversed)

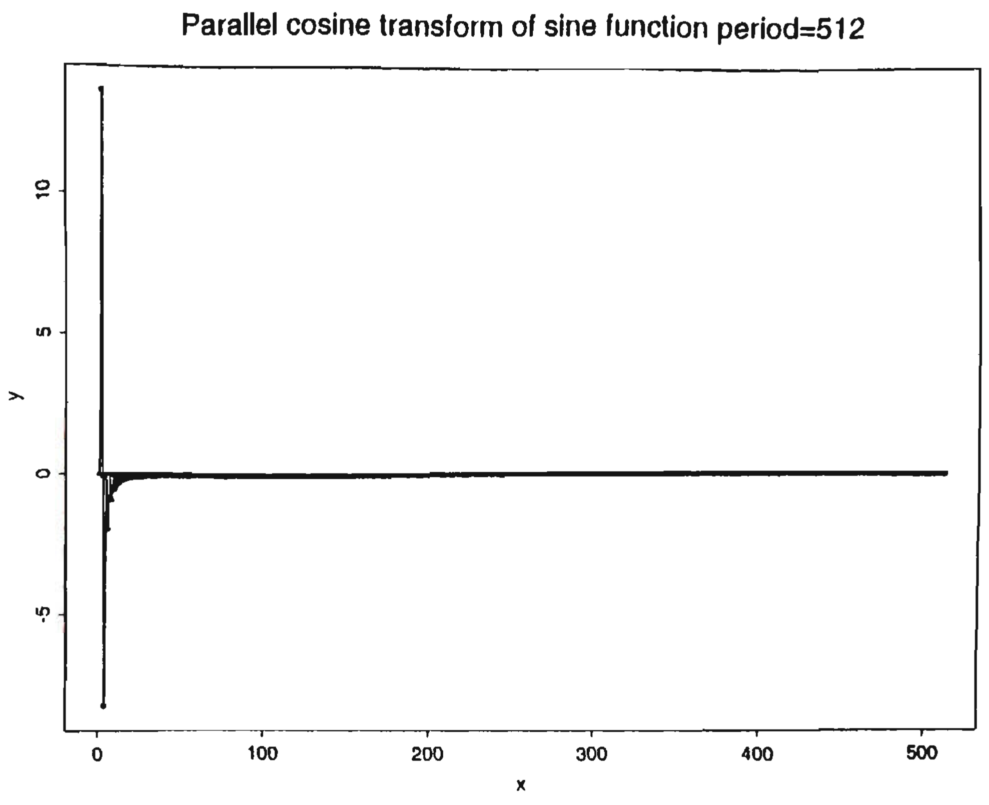
Figure 3.23 Parallel four processor hypercube implementation of the discrete cosine transform via the fast Fourier transform for N=8.

3.9 APPLICATION OF THE COSINE TRANSFORM TO PERIODIC AND NON-PERIODIC FUNCTIONS

The discrete cosine transform was performed on the test waveforms given in section 3.5. This was done in order to examine the ability of the discrete cosine transform with its smooth, global basis functions to distinguish a number of different features evident in the test functions. The discrete cosine transform of the trigonometric functions produces a compact spectrum comprising few spectral components. This is due to the close correlation between the functions and the transform basis functions.

The periodic discontinuties of the step functions produce a series of diminishing higher frequency spectral components. The chirp function and the higher frequency step function transforms demonstrate that as the input function waveform shape departs from the transform basis function set the ability of the transform to easily detect features or serve as a means of data compression is reduced.

The discrete cosine transform demonstrated a similar response to the iteration, delta and random data functions as was seen with the discrete Walsh transform. Non-periodic, discontinuous waveforms or waveforms localised in time produce "noisy" spectra with many frequency components which have no easily detectable features and provide little scope for data compression.



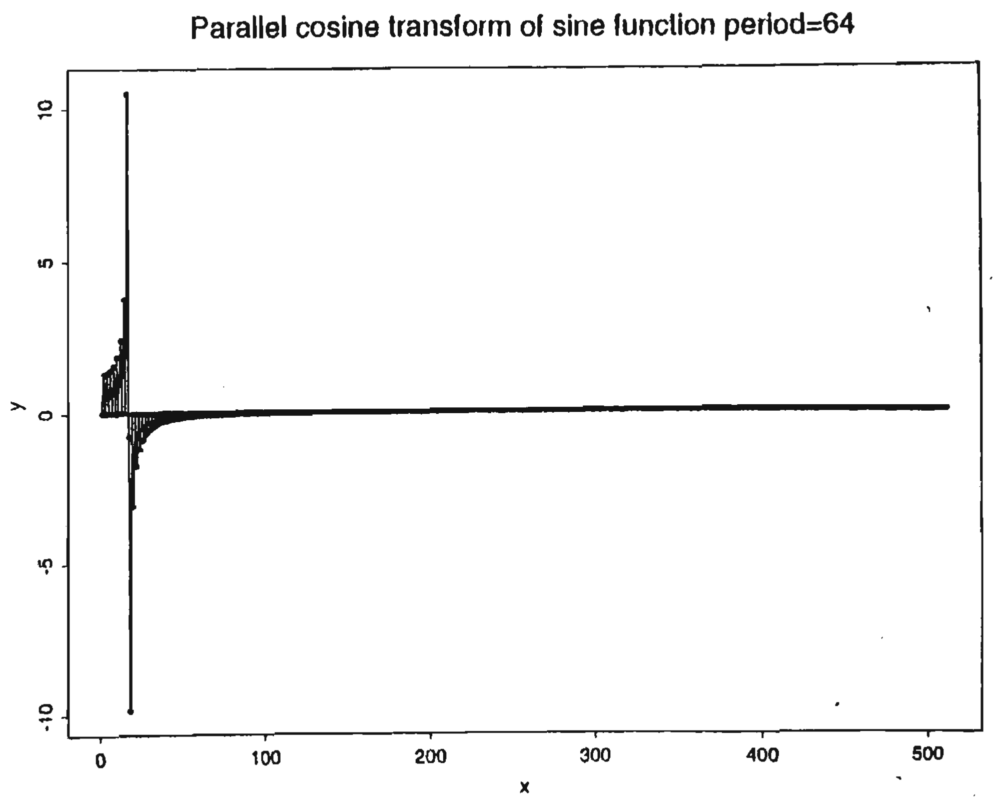
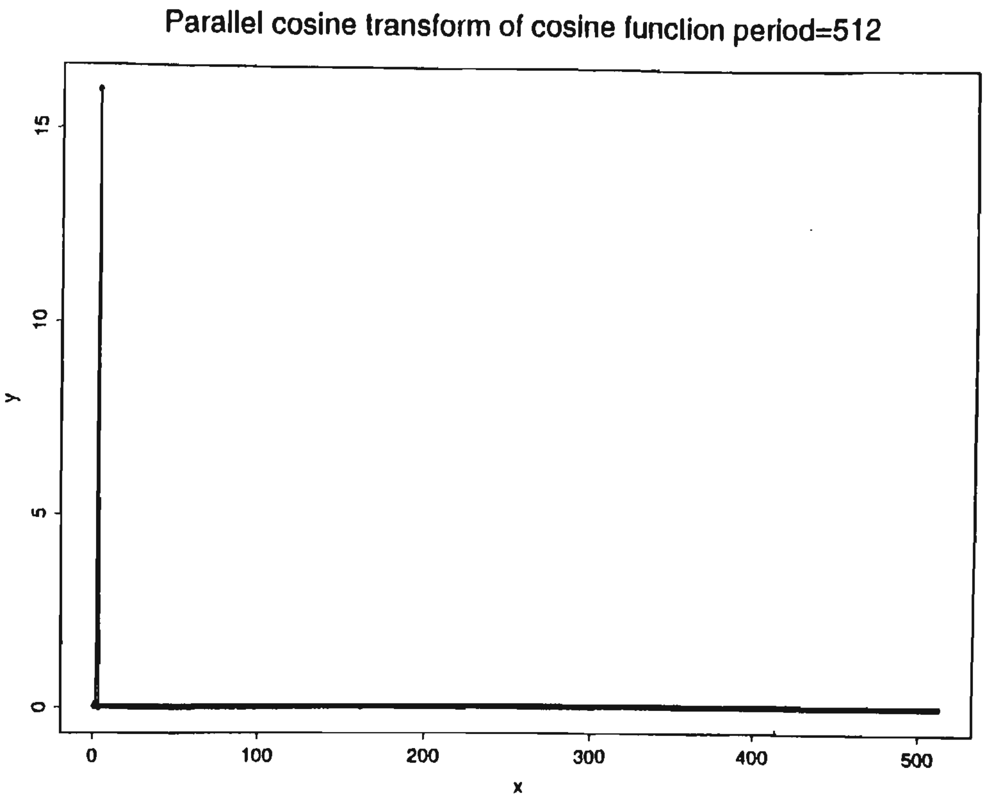


Figure 3.24 Two examples of the discrete cosine transform of the sine function where x = frequency (Hz) and y = transform coefficient magnitudes.



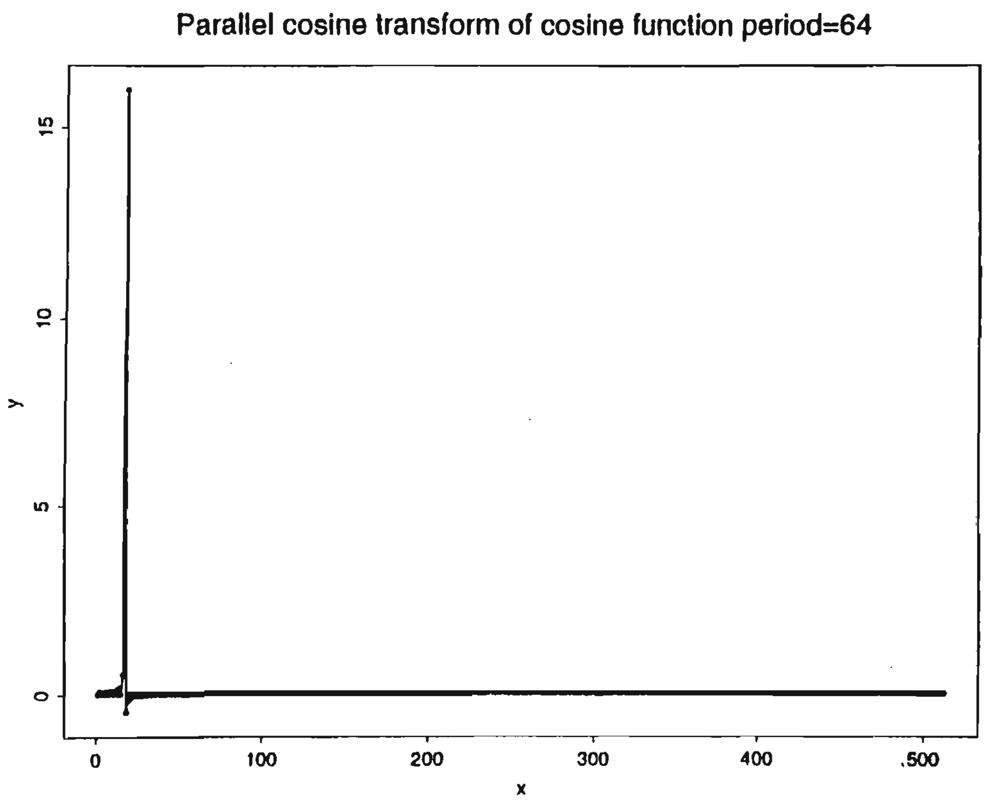
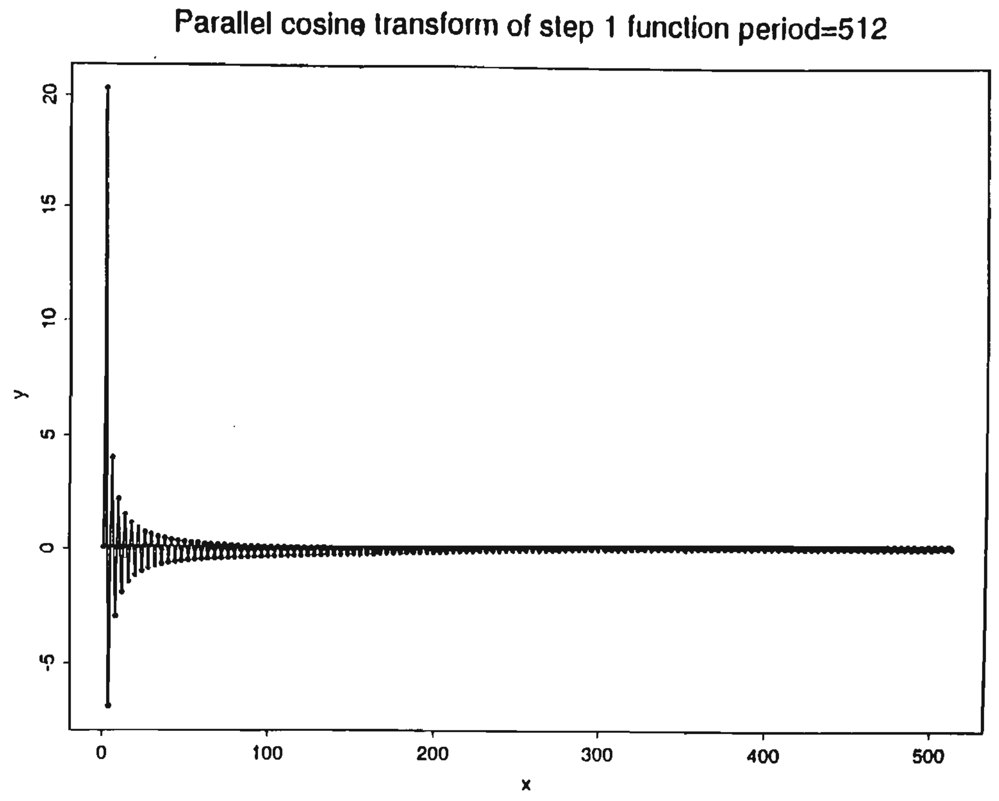


Figure 3.25 Two examples of the discrete cosine transform of the cosine function where x = frequency (Hz) and y = transform coefficient magnitudes.



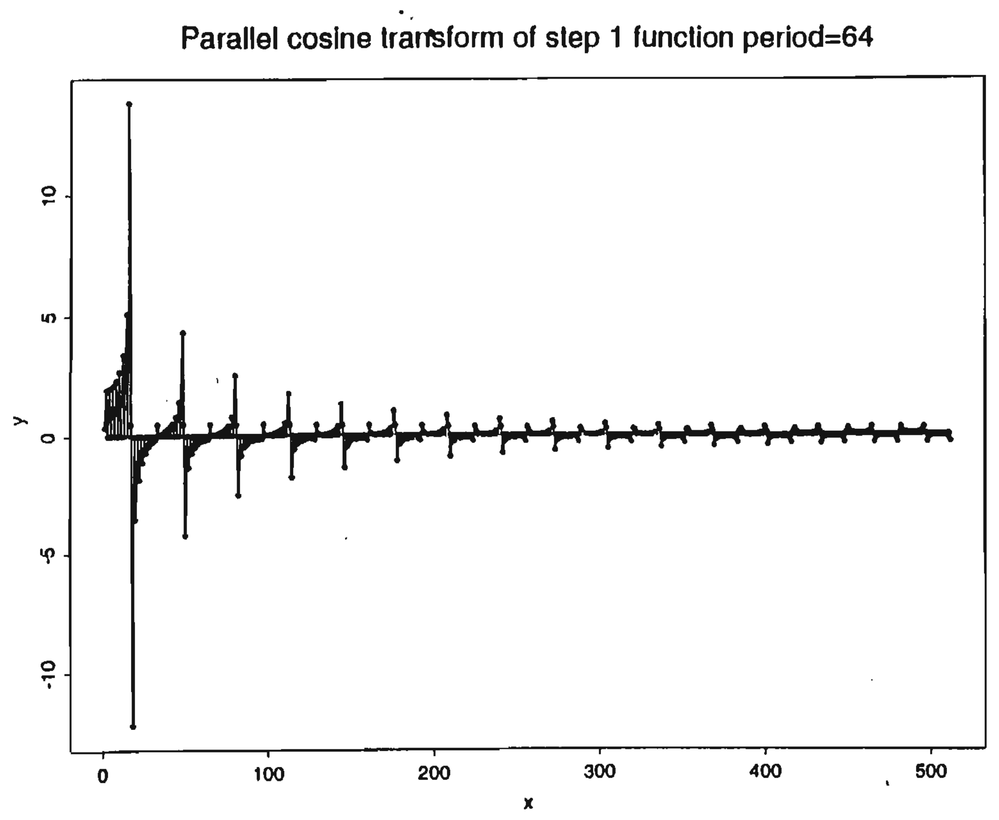
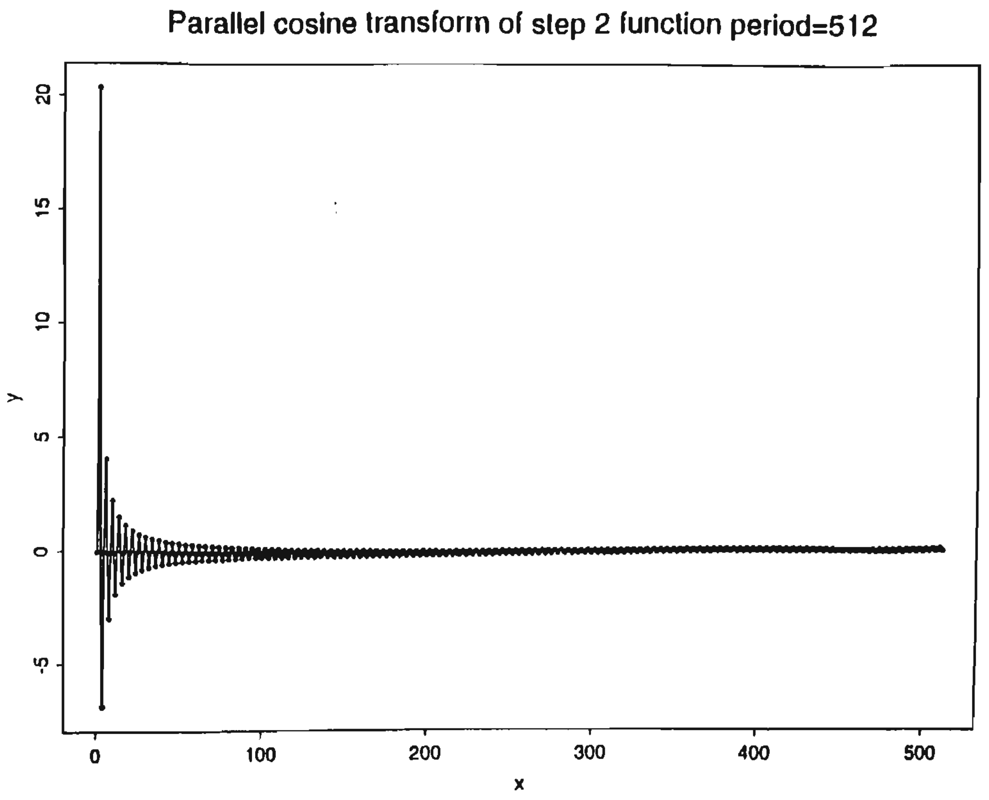


Figure 3.26 Two examples of the discrete cosine transform of the step 1 function where x = frequency (Hz) and y = transform coefficient magnitudes.



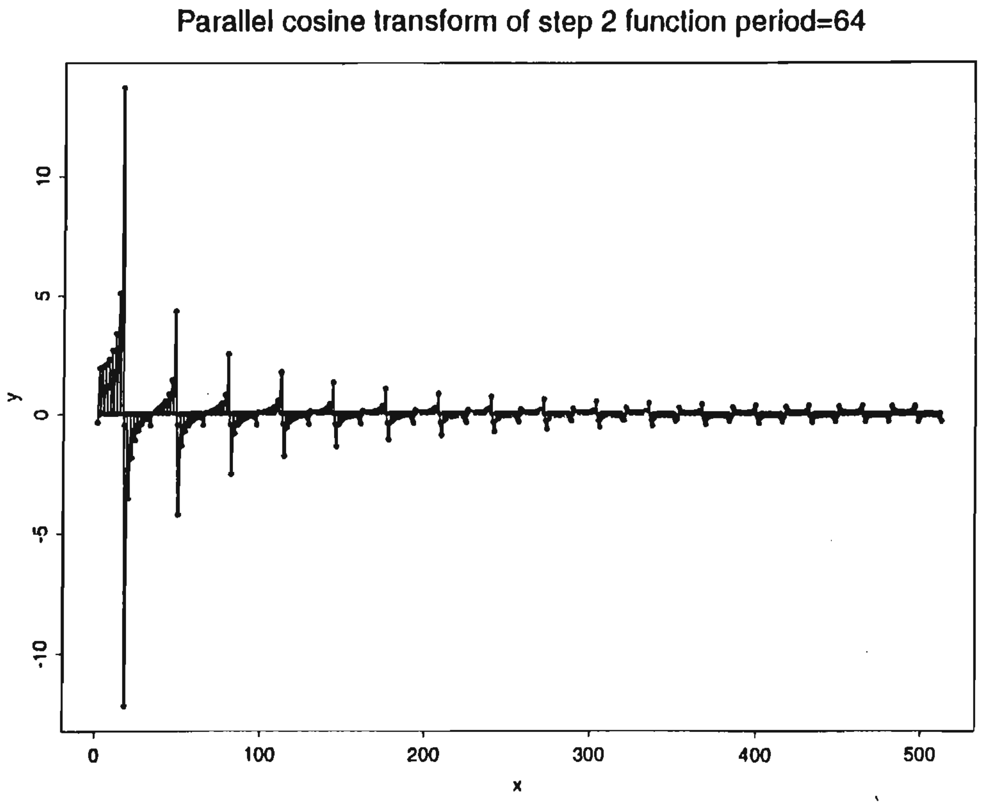
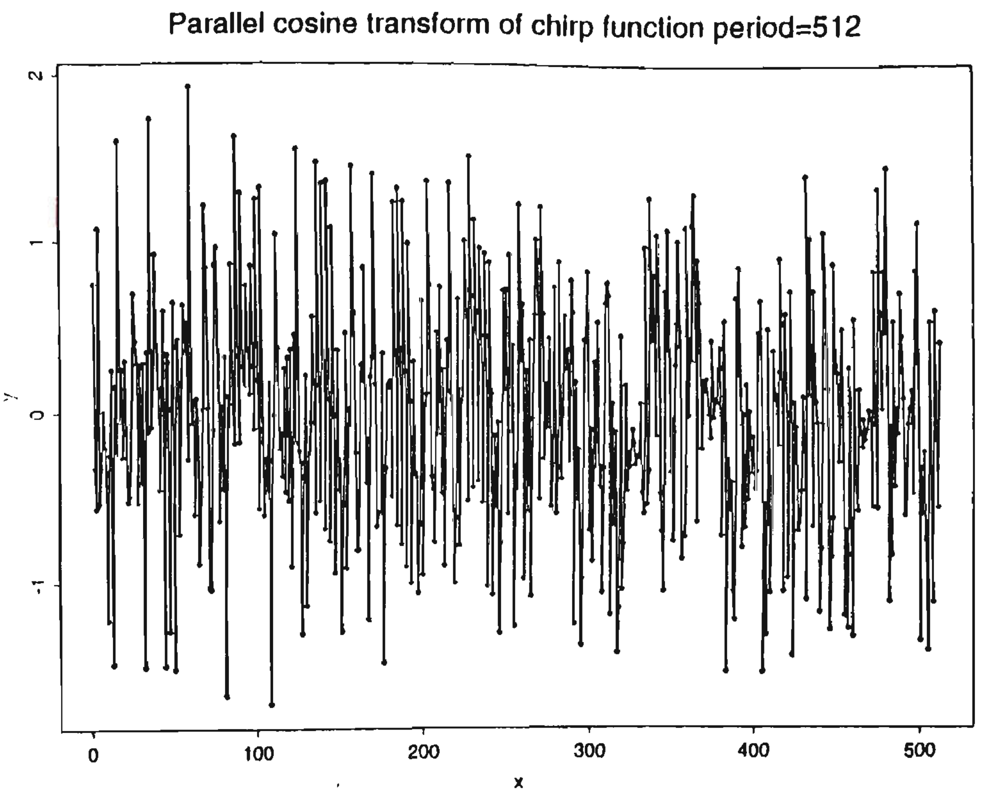


Figure 3.27 Two examples of the discrete cosine transform of the step 2 function where x = frequency (Hz) and y = transform coefficient magnitudes.



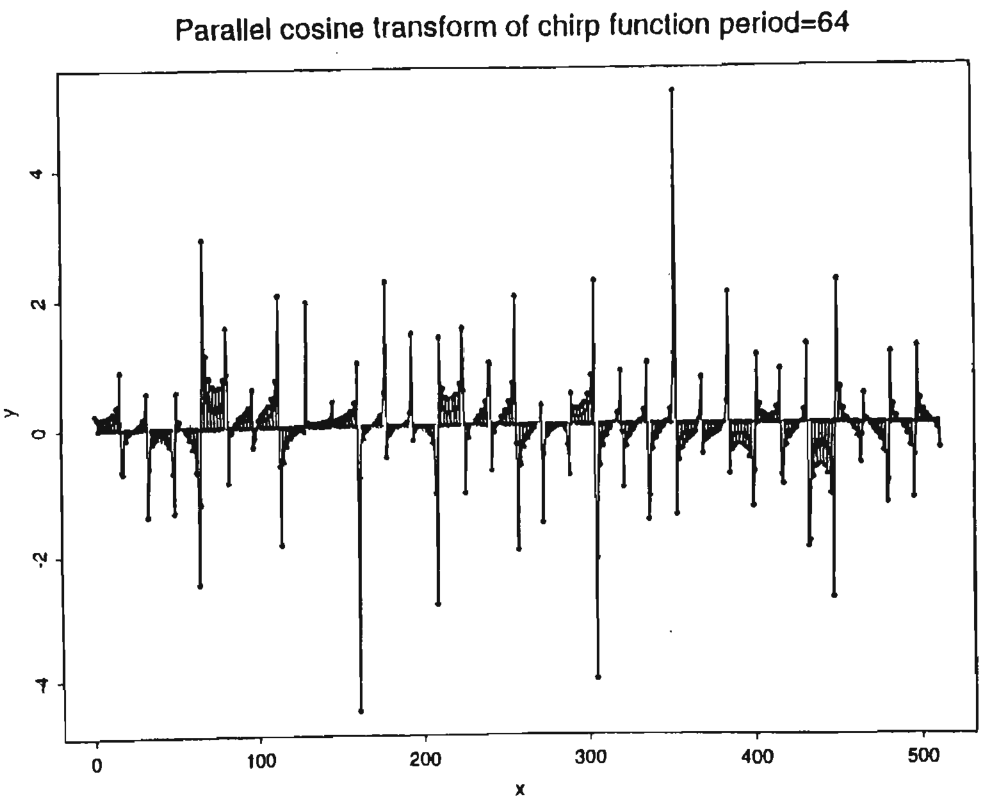
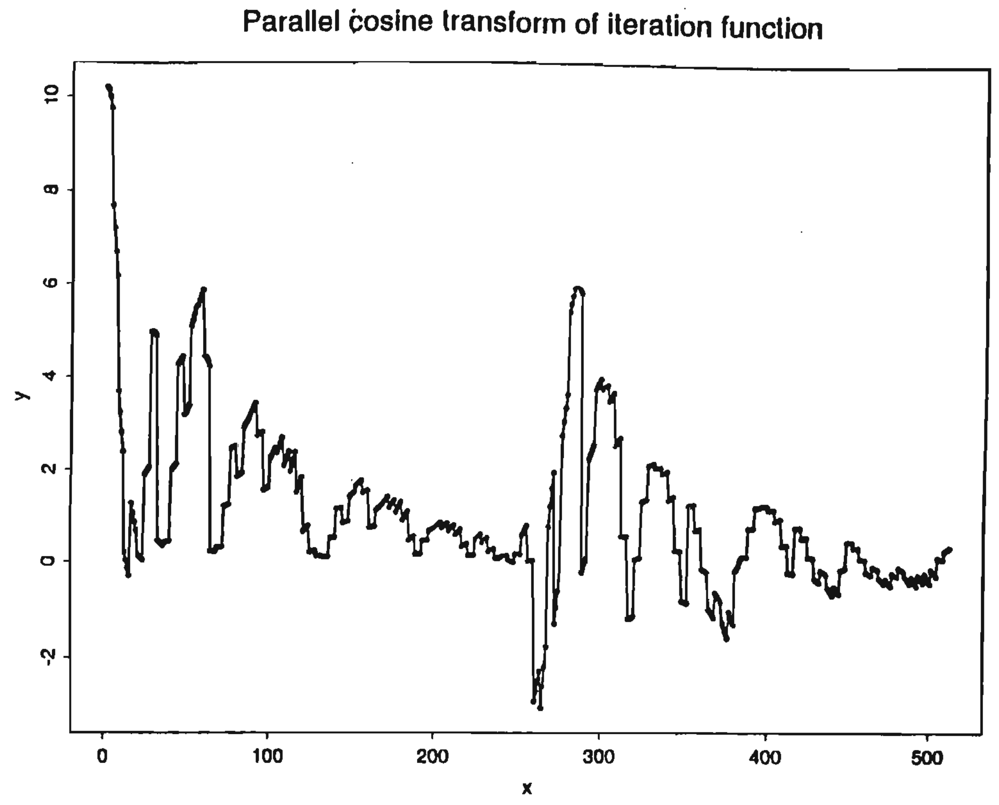


Figure 3.28 Two examples of the discrete cosine transform of the chirp function where x = frequency (Hz) and y = transform coefficient magnitudes.



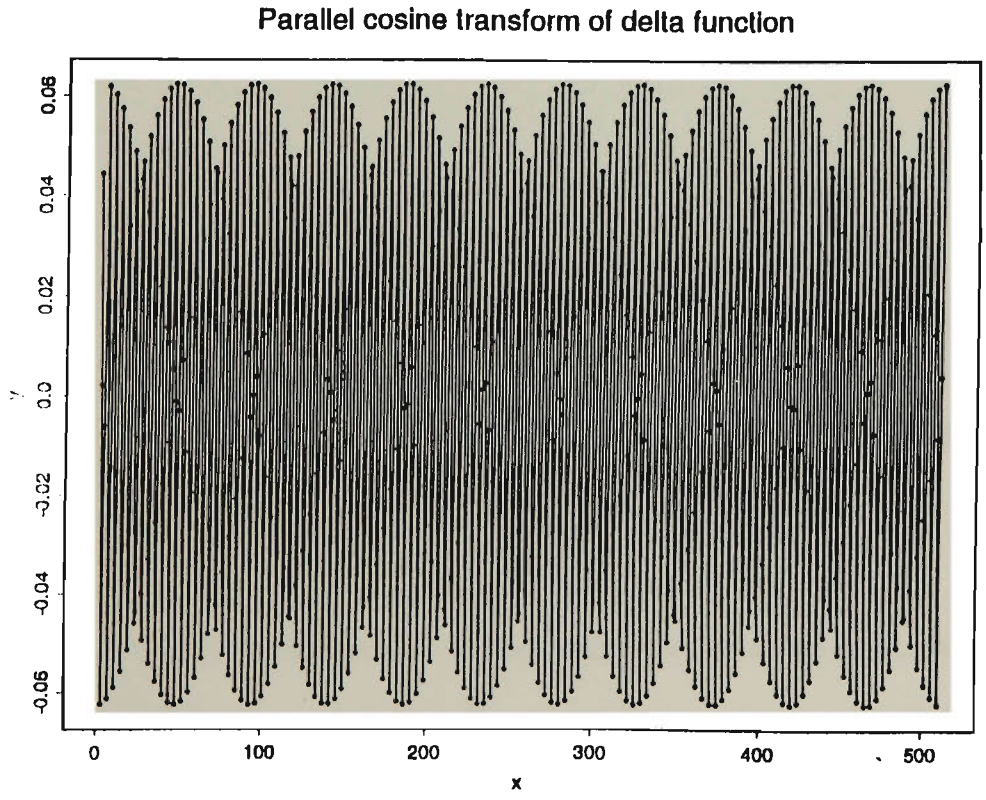


Figure 3.29 Examples of the discrete cosine transform of the iteration and delta functions where x = frequency (Hz) and y = transform coefficient magnitudes.

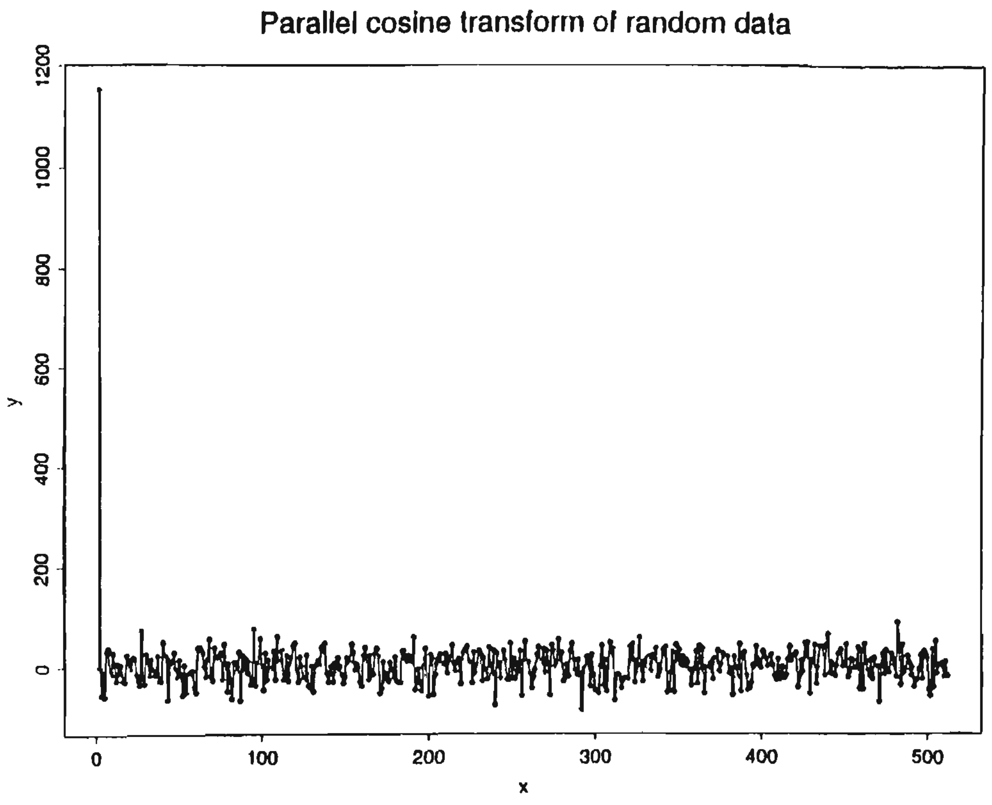


Figure 3.30 The discrete cosine transform of the random data function where x = frequency (Hz) and y = transform coefficient magnitudes.

3.10 COMPARISON OF SERIAL AND PARALLEL IMPLEMENTATIONS OF THE DISCRETE COSINE TRANSFORM

A comparison of the processor farm and hypercube implementations of the discrete cosine transform was neglected. The similarity between the parallel Walsh and cosine implementations would indicate a similar result for the discrete cosine transform as was found for the discrete Walsh transform.

A comparison of the two Transputer implementation of the discrete cosine transform versus a range of commercial microprocessors is given in figure 3.31.

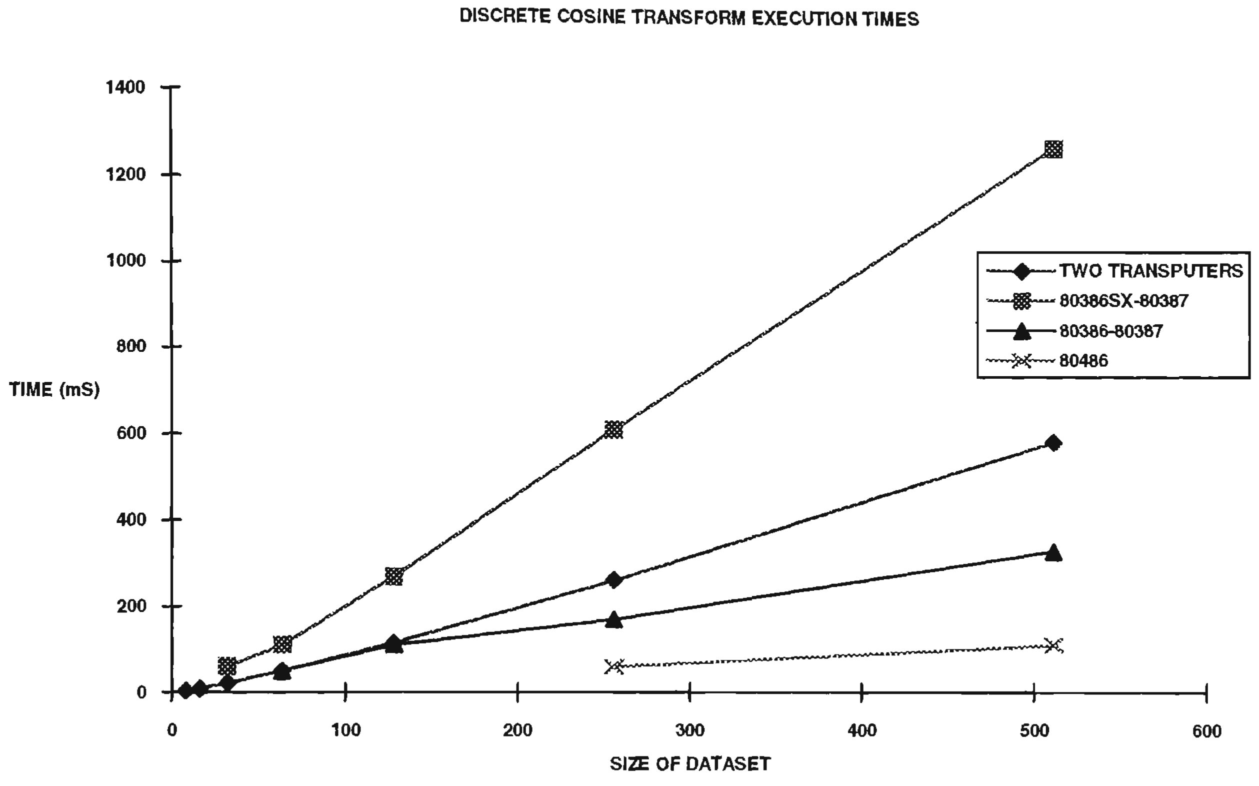


Figure 3.31 Serial and parallel processor performance when implementing the discrete cosine transform.

The microprocessor performance results for the discrete cosine transform are similar to the Walsh transform performance results. This can be attributed to the similarity between the discrete Walsh and cosine parallel algorithms. These results further confirm the obsolescence of the T800 Transputer. The results given in figure 3.31 indicate that a Transputer system consisting of five or more Transputers would be required to provide a performance better than that of the Intel 80486.

3.11 SUMMARY

The chapter describes a number of possible parallel algorithms for the discrete Walsh and cosine transforms. Of the possible implementations of the Walsh transform a hypercube based fast Walsh transform algorithm was found to be the most suitable for a Transputer system. A parallel algorithm for the discrete cosine transform was also developed. The calculation of the discrete cosine transform via the fast Fourier transform and implemented on a hypercube processor topology was found to be the most appropriate for the Transputer system.

A comparison between a parallel Transputer implementation of these transforms and a sequential implementation on a range of commercial microprocessors was conducted. Results of these comparisons indicate that the T800 version of the Transputer cannot provide the level of performance found in the current generation of microprocessors. Transputer systems of four or more Transputers were required in order to provide a superior performance to the current range of sequential microprocessors.

The discrete Walsh and cosine transformations were performed on a number of test waveforms in order to determine their ability to distinguish a number of periodic or localised signal features. The transform spectra obtained agreed with the expected behaviour of the transforms. Both transforms providing simple spectra for input functions which matched the transform basis functions. It was found that both transforms were unable to provide spectra of any practical value for waveforms which either contained a large number of discontinuties or were localised in time. The reason for this being the shape and global nature of the transform basis function set of the two transforms.

CHAPTER 4

HYPERCUBE IMPLEMENTATION OF TRANSFORMS

4.1 INTRODUCTION

It was shown in chapter one that the discrete Walsh transform matrix can be determined by performing a recursive Kronecker product operation on Hadamard matrices (equation(1.25)). Granata[26] shows that the discrete Fourier, cosine, and Hartley transforms can also be expressed in terms of Kronecker products.

This chapter describes a technique which converts Kronecker products to matrix or Hadamard products and allows Kronecker product derived transforms to be easily mapped onto hypercube processor topologies, providing an alternative method of parallel implementation.

4.2 KRONECKER DECOMPOSITION AND ITS RELEVANCE TO A HYPERCUBE IMPLEMENTATION OF THE WALSH TRANSFORM

The discrete Walsh transform can be expressed as a matrix multiplication of a data vector and a transform coefficient matrix. Equation(1.23) shows that the coefficient matrix can be determined by performing recursive Kronecker products on Hadamard matrices as shown by equation(1.25). A parallel algorithm for calculating Kronecker products could therefore be easily adapted to calculate a Walsh transformation.

Brewer[7] shows that given equation(1.26) the following results

$$\mathbf{A}_1\mathbf{A}_2 \otimes \mathbf{B}_1\mathbf{B}_2 = (\mathbf{A}_1 \otimes \mathbf{B}_1)(\mathbf{A}_2 \otimes \mathbf{B}_2). \qquad (4.1)$$

where $\mathbf{A}_1, \mathbf{A}_2, \mathbf{B}_1, \mathbf{B}_2$ are matrices of the type defined above. Choosing $\mathbf{A}_1 = \mathbf{I}_A, \mathbf{B}_2 = \mathbf{I}_B$ gives

$$\mathbf{I_A A \otimes B I_B = A \otimes B = (I_A \otimes B)(A \otimes I_B)}. \qquad (4.2)$$

Kronecker products are thus expressible as the product of Kronecker products of the original matrix and the appropriate identity matrix. Thus repeated Kronecker products can be calculated as matrix products by implementing (4.2) recursively.

Equation (4.2) can be viewed as the product of two matrices which have undergone a restructuring operation. An alternative method is available for effecting the matrix restructuring performed by the identity-matrix Kronecker products shown in equation(4.2).

The Kronecker product can be expressed as

$$\mathbf{A \otimes B} = \begin{cases} \mathbf{B^* A^*} & p \geq m \\ \mathbf{A^\# B^\#} & p < m \end{cases} \qquad (4.3)$$

where $\mathbf{B}^* = \mathbf{B}$ for $m = 1$, $\mathbf{A}^* = \mathbf{A}$ for $q = 1$, $\mathbf{B}^\# = \mathbf{B}$ for $n = 1$, $\mathbf{A}^\# = \mathbf{A}$ for $p = 1$

given that $\mathbf{A}^*, \mathbf{B}^*, \mathbf{A}^\#, \mathbf{B}^\#$ are restructured forms of the original matrices **A** and **B**.

Equation (4.3) shows that the Kronecker product can be viewed as two distinct operations, a matrix mapping or restructuring operation and an algebraic operation. If the matrices can be restructured appropriately the Kronecker product can be reduced to a matrix multiplication.

Where restructuring is required the structure of the matrices is determined by altering the binary representation of the matrix row and column indices by adding a new "dummy" variable with the required number of bits to the row and column indices to create a matrix of the same size as the Kronecker product resultant matrix. This is illustrated with the following example.

Example 1:

$$\text{Given } \mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

The row and column indices of matrix **B** can be represented in binary form as $R_0 = 0 \rightarrow 1$, $C_0 = 0 \rightarrow 1$ as shown in table 1. Addition of the dummy bit x_0 to the row and column indices $x_0R_0 = 00 \rightarrow 11$, $x_0C_0 = 00 \rightarrow 11$ creates the matrix $\mathbf{B}^*$ as shown in table 2.

Table 1:

Matrix B

Row R_0	Column C_0	Matrix Element
0	0	b_{11}
0	1	b_{12}
1	0	b_{21}
1	1	b_{22}

Table 2:

Matrix $\mathbf{B}^*$

Row x_0R_0	Column x_0C_0	Matrix Element
00	00	b_{11}
10	10	b_{11}
00	01	b_{12}
10	11	b_{12}
01	00	b_{21}
11	10	b_{21}
01	01	b_{22}
11	11	b_{22}

Expressing these tables in matrix form gives

$$\mathbf{B}=\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \qquad \mathbf{B}^*=\begin{bmatrix} b_{11} & b_{12} & 0 & 0 \\ b_{21} & b_{22} & 0 & 0 \\ 0 & 0 & b_{11} & b_{12} \\ 0 & 0 & b_{21} & b_{22} \end{bmatrix}.$$

The matrix $\mathbf{A}^*$ is formed by moving the dummy bit one place to the right in the row and column indices to give R_0x_0, C_0x_0. Expressing this in matrix form gives

$$\mathbf{A}=\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \qquad \mathbf{A}^*=\begin{bmatrix} a_{11} & 0 & a_{12} & 0 \\ 0 & a_{11} & 0 & a_{12} \\ a_{21} & 0 & a_{22} & 0 \\ 0 & a_{21} & 0 & a_{22} \end{bmatrix}.$$

The product of these restructured matrices is $\mathbf{B}^*\mathbf{A}^* = \mathbf{A}\otimes\mathbf{B}$.

When p<m the restructuring uses the same technique as for the case $p \geq m$ with the exception that the position of the dummy bits added to the column index are reversed in position. For example given

Example 2:

$$\mathbf{A}=\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \qquad \mathbf{B}=\begin{bmatrix} b_{11} & b_{12} \end{bmatrix}$$

The matrix $\mathbf{B}^\#$ has the row index R_0x_0, and the column index x_0C_0, this gives the matrix

$$\mathbf{B}^\#=\begin{bmatrix} b_{11} & b_{12} & 0 & 0 \\ 0 & 0 & b_{11} & b_{12} \end{bmatrix}$$

The Kronecker product can then be given as $A \otimes B = AB^{\#}$. For the case of square matrices of size 2^n the above method can be expressed as

$$\mathbf{C}_1 \otimes \mathbf{C}_2 \otimes \ldots \otimes \mathbf{C}_n = \prod_{i=n}^{1} \mathbf{C}_i^* \qquad (4.4)$$

where the binary row-column indices restructuring is given by

Matrix	Row index	Column index
$\mathbf{C}_n$	$x_j \ldots x_0 R_i \ldots R_0$	$x_j \ldots x_0 C_i \ldots C_0$
$\mathbf{C}_2$	$x_j R_i \ldots R_0 \ldots x_0$	$x_j C_i \ldots C_0 \ldots x_0$
$\mathbf{C}_1$	$R_i \ldots R_0 x_j \ldots x_0$	$C_i \ldots C_0 x_j \ldots x_0$

Kronecker products can be reduced to a matrix multiplication of matrices which have been restructured by the addition of an independent dummy variable to the row and column index values of the matrix elements. This can be taken a step further. The introduction of two independent dummy variables in the matrix restructuring results in the algebraic operations which have to be performed on the matrices being reduced to a simple element-element multiplication. This is known as the Hadamard product (Horn[34]).

Example 3.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$\mathbf{A} \otimes \mathbf{B} = \mathbf{B}^* . \mathbf{A}^*$$

given $\mathbf{B}^*_{\text{row index}} = x_1 R_0$, $\mathbf{B}^*_{\text{column index}} = x_0 C_0$, $\mathbf{A}^*_{\text{row index}} = R_0 x_1$, $\mathbf{A}^*_{\text{column index}} = C_0 x_0$, and where the dot operator represents the Hadamard product.

This element mapping results in the matrices given in equation(4.5)

$$\mathbf{A}\otimes\mathbf{B}=\begin{bmatrix} b_{11} & b_{12} & b_{11} & b_{12} \\ b_{21} & b_{22} & b_{21} & b_{22} \\ b_{11} & b_{12} & b_{11} & b_{12} \\ b_{21} & b_{22} & b_{21} & b_{22} \end{bmatrix} \cdot \begin{bmatrix} a_{11} & a_{11} & a_{12} & a_{12} \\ a_{11} & a_{11} & a_{12} & a_{12} \\ a_{21} & a_{21} & a_{22} & a_{22} \\ a_{21} & a_{21} & a_{22} & a_{22} \end{bmatrix} \qquad (4.5)$$

The matrix restructuring operations can be given a geometric interpretation by combining the matrix row-column indices to give the address of the corresponding matrix element in a geometric structure. For example the binary representations of the row and column indices of matrix **B** in the first example can be considered as the addresses of the elements on a two-dimensional hypercube.

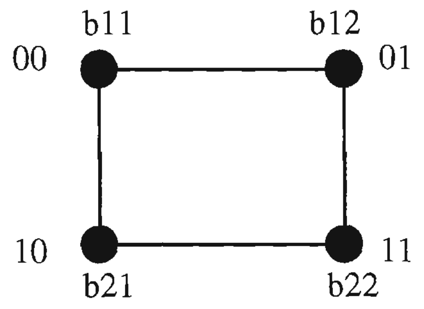


Figure 4.1 Hypercube representation of matrix **B**.

From this viewpoint the matrix restructuring in this example represents a mapping of the matrices **A** and **B** onto a four dimensional base 2 hypercube, and the matrix multiplications required to determine the Kronecker product correspond to data transfers and multiplication of matrix elements or vertex values.

The introduction of the dummy variable provides a partial mapping to the hypercube, requiring a matrix multiplication to perform the Kronecker product. If the matrices **A** and **B** are mapped onto the hypercube using two independent dummy variables the Kronecker product reduces to a simple multiplication of each of the elements at each hypercube vertex. This can be illustrated using example 3. Combining the row-column indices of the matrix elements and mapping the elements onto a four dimensional hypercube results in the structure given in figure 4.2.

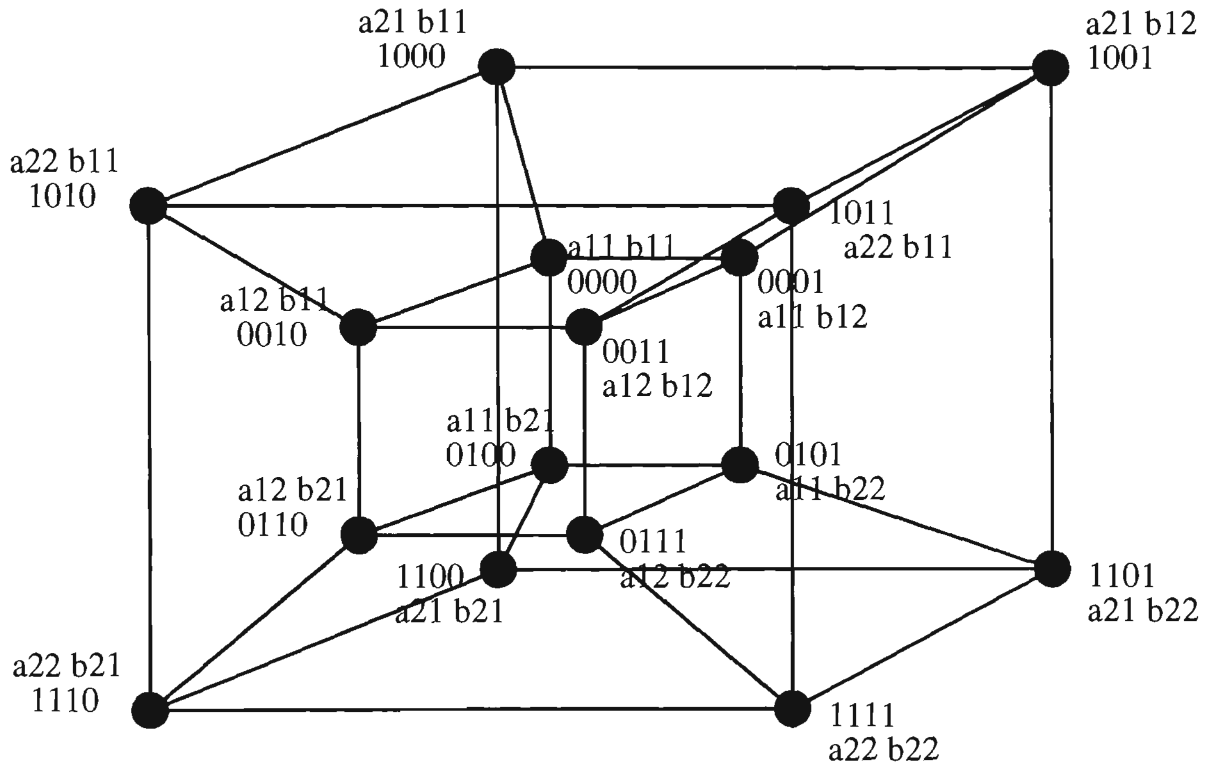


Figure 4.2 Mapping of matrix elements of **A** and **B** to a four dimensional hypercube.

The Kronecker product can now be found by multiplying the elements residing on each node of the cube.

The number of dummy variables required when restructuring a matrix can be found using equation(4.6)

$$\begin{aligned} pq &= b_A^{\,nm+nn} \\ mn &= b_B^{\,np+nq} \end{aligned} \qquad (4.6)$$

where b_A = numbering base of matrix A, nm = number of row dummy variables of matrix A, nn = number of column dummy variables of matrix A. The unit index of a vector or its transpose are not used in the restructuring process with the dummy variables for a unit index being set to zero.

Laksmivarahan[42] shows that the hypercube is a member of the family of (n,b,k) cubes where n is the number of vertices or nodes, b is the base of the node numbering system and k is the dimension of the cube. A complete cube satisfies equation (4.7)

$$n = b^k. \qquad (4.7)$$

Kronecker products of matrices of size other than $2^a \times 2^b$ can be represented as element wise multiplications on (n,b,k) cubes. The criteria for the choice of cube being that the (mnpq) elements of the Kronecker product resultant matrix map efficiently to the nodes of the cube, allowing even balancing of processor workload.

The mapping of a Kronecker product to an incomplete, base three cube is given in example 4.

Example 4.

$$\text{Given } \mathbf{A} = \begin{bmatrix} a_{11} \\ a_{21} \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & b_{13} \end{bmatrix}$$

The matrix resulting from the Kronecker product will have mnpq = 6 elements therefore a cube with six nodes would be optimal, a (6,3,2) incomplete cube provides the required number of nodes. Restructuring **A** and **B** using equation (4.6) gives the tables shown below.

Matrix A

R_0x_0	Element
00	a_{11}
01	a_{11}
02	a_{11}
10	a_{21}
11	a_{21}
12	a_{21}

Matrix B

x_0C_0	Element
00	b_{11}
10	b_{11}
01	b_{12}
11	b_{12}
02	b_{13}
12	b_{13}

The matrix elements can now be mapped to the (6,3,2) cube vertices with the addresses given in the tables.

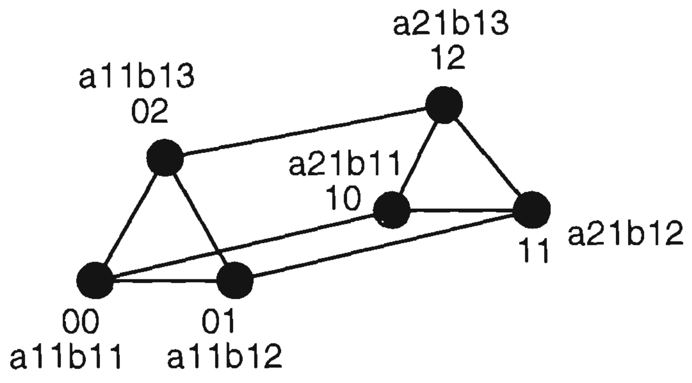


Figure 4.3 Mapping of matrix elements **A** and **B** to a (6,3,2) cube.

Similarly Kronecker sums can be mapped to hypercube structures, the only variation in the operation being that the elements mapped to cube nodes are added rather than multiplied. As the operations performed on each node are the same these algorithms could be easily ported to either MIMD or SIMD architectures.

If the number of processors does not match the size of the resultant matrix each node may be loaded with more element operations by mapping onto the hypercube the restructured matrices determined for the matrix multiplication technique.

Example 5.

Calculating a Kronecker product on a four node hypercube

$$\mathbf{A}=\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \qquad \mathbf{B}=\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$\mathbf{A}\otimes\mathbf{B}=\mathbf{B}^{*}\mathbf{A}^{*}$$

The matrix rows are divided between the available hypercube processors as shown below.

$$\begin{bmatrix} b_{11} & b_{12} & 0 & 0 \\ \hdashline b_{21} & b_{22} & 0 & 0 \\ \hdashline 0 & 0 & b_{11} & b_{12} \\ \hdashline 0 & 0 & b_{21} & b_{22} \end{bmatrix}\begin{bmatrix} a_{11} & 0 & a_{12} & 0 \\ \hdashline 0 & a_{11} & 0 & a_{12} \\ \hdashline a_{21} & 0 & a_{22} & 0 \\ \hdashline 0 & a_{21} & 0 & a_{22} \end{bmatrix}\begin{array}{l} \text{Processor 1} \\ \text{Processor 2} \\ \text{Processor 3} \\ \text{Processor 4} \end{array}$$

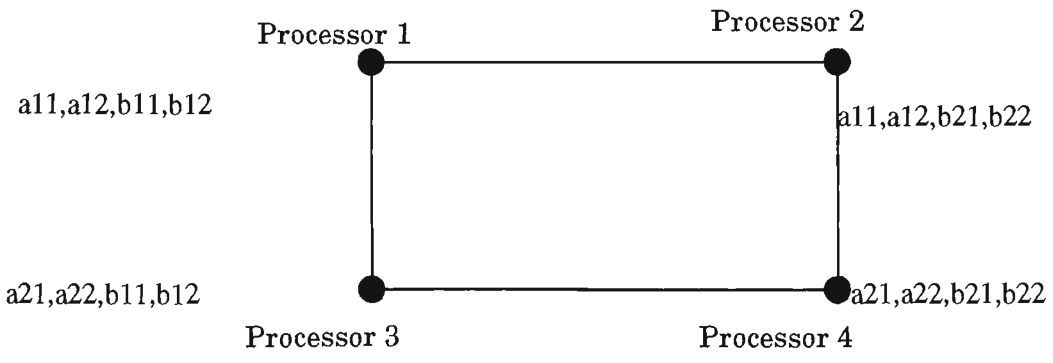


Figure 4.4 Mapping of matrix rows onto hypercube.

Multiplication of elements on each processor results in the following resultant matrix distributed over the hypercube.

$$\begin{array}{l}\left[\begin{array}{cccc} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} \\ \hdashline a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22} \\ \hdashline a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} \\ \hdashline a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} \end{array}\right]\begin{array}{l}\text{Processor 1}\\ \text{Processor 2}\\ \text{Processor 3}\\ \text{Processor 4}\end{array}\end{array}$$

4.3 SUMMARY

To summarise, the Kronecker product can be determined by means of either a matrix multiplication, or a set of simple matrix element multiplications dependant upon the structure of the constituent matrices. This ability makes them amenable to parallel calculation on hypercubes utilising a simple mapping algorithm. The Walsh transform can then be performed by mapping the data vector elements to the processors in the same manner as the matrix elements.

The advantages of these techniques are that first, Walsh transforms via Kronecker products can be calculated in parallel on hypercubes without the need for application specific processor topologies. Secondly the programming complexity commonly associated with parallel algorithms is avoided and thirdly the algorithm can be easily scaled to fit the available processor topology.

The drawback of the Kronecker method of determining the Walsh transform is that it requires N^4 operations as compared to the $N \log N$ operations of the fast Walsh transform. But when creating a parallel implementation other factors should be considered. For a two transputer system approximately 10% of the processing time used to perform the fast Walsh transform was taken up performing inter-processor communications and the bit-reversal. These figures will fluctuate for varying size datasets and processor topologies. The lack of inter-processor communications or bitreversal operations for the Kronecker product Walsh transform implies that the theoretical performance improvement will be proportional to the number of processors used. Also the mapping of the Kronecker product on a hypercube topology is less complex than that required for the fast Walsh transform, particularly for large datasets.

It is undeniable that the fast Walsh transform is a more efficient algorithm, requiring fewer operations. However given the advantages of the Kronecker algorithm an investigation of the performance of the two algorithms for varying size datasets and processor networks would be recommended. The two transputer system used to implement the fast Walsh algorithm is not large enough to permit valid performance comparisons.

CHAPTER 5

PARALLELISING THE HAAR AND D4 WAVELET TRANSFORMS

5.1 INTRODUCTION

The Haar transform is an example of a transform with both global and local basis functions. The basis functions of the D4 wavelet transform are all local in extent. Transforms such as these are being widely used in applications where detection of transient features is required. Many of these applications would benefit from the improved computational performance which can be achieved by applying parallel processing techniques.

5.2 IMPLEMENTATION OF THE HAAR TRANSFORM USING TRANSPUTERS

The signal flow graph of the one-dimensional Haar transform (figure 1.5) displays a pyramidal structure. Algorithms for two-dimensional Haar transforms for image processing have reinforced this pyramidal structure suggesting a matching parallel implementation on a pyramidal processing topology (Corrioli[9]). This suggests that for the one-dimensional Haar transform a mapping onto a binary tree topology may provide a efficient parallel implementation.

Figure 5.1 illustrates a binary tree topology for an eight datapoint Haar transform, each node performing an addition and subtraction of the data transmitted from the nodes further up the tree. The results are then multiplied by the appropriate coefficients. Members of the resulting transform vector appear on the nodes shown, transmission of these results up the tree would allow the vector to be "assembled" at the root node by the end of the calculation. Large datasets could be distributed across smaller binary tree structures by grouping calculations on nodes with a corresponding increase in the complexity of operations required at each node, this could be seen as a move in the continuum from "fine-grained" parallel processing to a "coarse-grained" parallel processing approach.

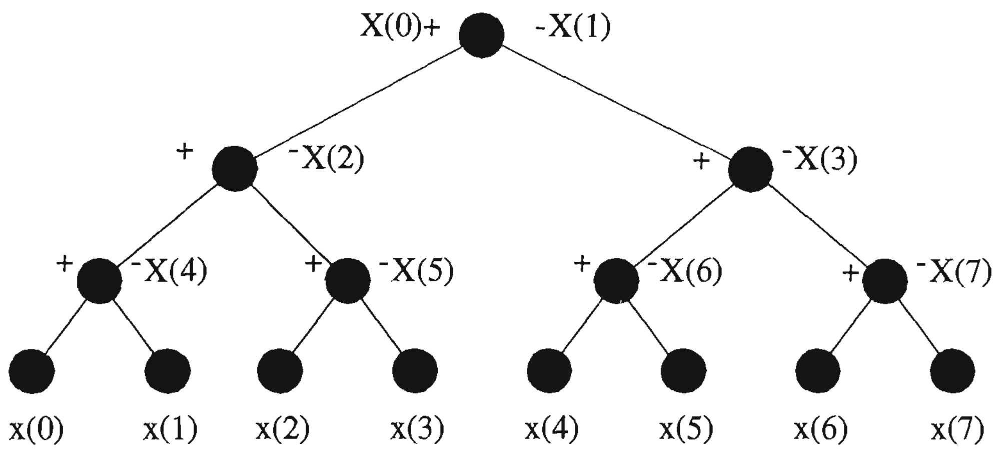


Figure 5.1 Implementation of the Haar transform on a binary tree processor topology.

One of the criticisms of parallel processing algorithms is that many algorithms require their own unique processor topology for optimal operation. In the case of the Haar transform it is possible to remove this shortcoming by transferring the binary tree topology to the hypercube processor topology which has already been used for the parallel implementation of the fast Walsh and fast Fourier transforms. Figure 5.2 shows one way of distributing the Haar transform data across a hypercube and the data communications that would be required between nodes in order to perform the Haar transform. It can be seen that there is no communications performance loss when transferring between topologies In the example given when distributing data there is a maximum transmission length of three, the dimension of the hypercube and the depth of the binary tree. When performing the transform three sets of inter-nodal communications are required for both topologies.

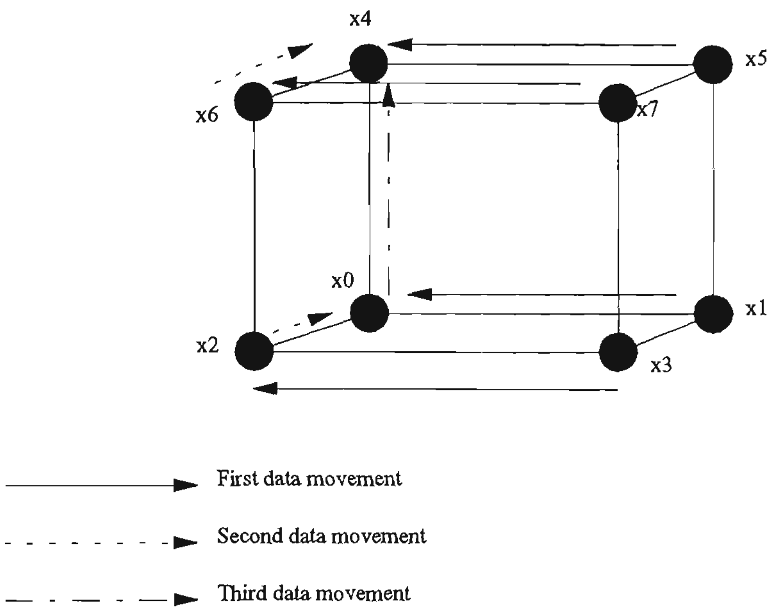


Figure 5.2 Implementation of an eight datapoint Haar transform on a Hypercube processor topology.

The hypercube data transfers required can be generalised. A data transfer algorithm expressed as pseudo code was found and is shown below.

```
PARALLEL
    FOR J = 0 TO CUBE_DIMENSION-1
        FOR I = 0 TO NUMBER_OF_NODES-2    STEP 2
            $X(I) \leftarrow X(I) - X(2^J + I)$
END PARALLEL
```

5.3 APPLICATION OF THE HAAR TRANSFORM TO PERIODIC AND NON-PERIODIC FUNCTIONS

As mentioned in 1.3.3 the Haar transform consists of basis functions some of which are defined globally over the transformation dataset and some which are defined locally over the transformation dataset. This gives the Haar transform a sensitivity to local singularities not found in more traditional transformations such as the Walsh, cosine and Fourier transforms.

The Haar transform was performed on a number of waveforms in order to observe its response to test data ranging from periodic functions to datasets containing local singularities.

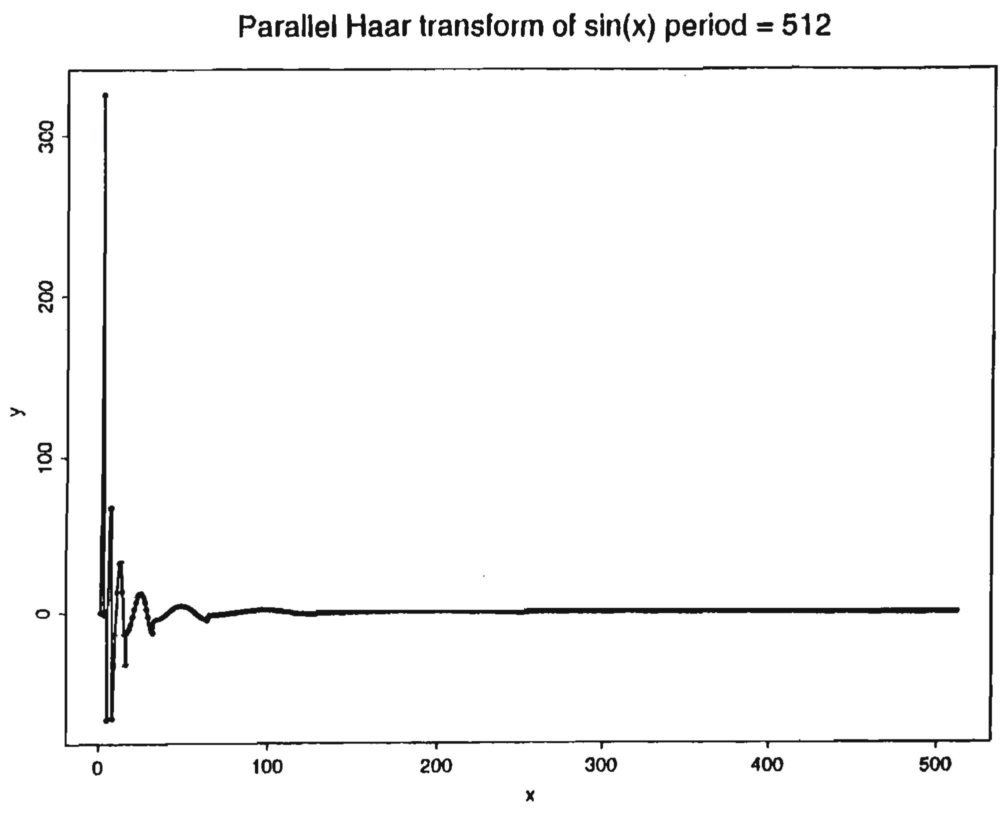
The Haar transform of the trigonometric functions shown in figures 5.3-5.4 reveal a distinctive spectra with components not as differentiated as equivalent Walsh transform spectra due to the localised square wave basis functions of the Haar transform. As with the Walsh transform higher frequency waveforms cause a spreading of spectra components. The increase in the number of spectral components for increased signal frequencies is due to the local nature of the basis functions. Higher frequency global input functions require a number of higher sequency Haar basis functions to provide an accurate representation. This can be seen using the example of a simple function such as a square wave. Determine the Haar basis functions required to represent a simple square wave, then compare this with the basis functions required to construct the same function at a higher frequency. A greater number of basis function/spectral components are required due to the local nature of the basis functions.

The decrease in the amplitude of the Haar transform coefficients when the input function frequency is increased is due to the increasing amplitude of the higher sequency Haar basis functions as given by equation(1.29).

The Haar transform provides an easily distinguishable spectra for the step functions similar to the Walsh transform. The prominent periodic sequency components of the spectra caused by representing a periodic wave train with higher sequency basis functions which are localised in time.

As with the Walsh transform the Haar transform of the chirp function produces a complex spectra not easily amenable to detection. The sharp variations in amplitude of the single cycle chirp function generate many spectra components. Higher frequency chirp functions give a Walsh transform of unique global spectra components while the Haar transform shows a repetitive spectral structure because the basis functions are localised in time and so repeat along the spectrum.

The Haar transform provides a simple spectra for the delta function. The inclusion in the basis function set of localised square waves allows the delta function to be represented by a few discrete spectral components. A decaying non-periodic function such as the iterative function can also be represented by a decaying series of localised square waves as shown in figure 5.8.



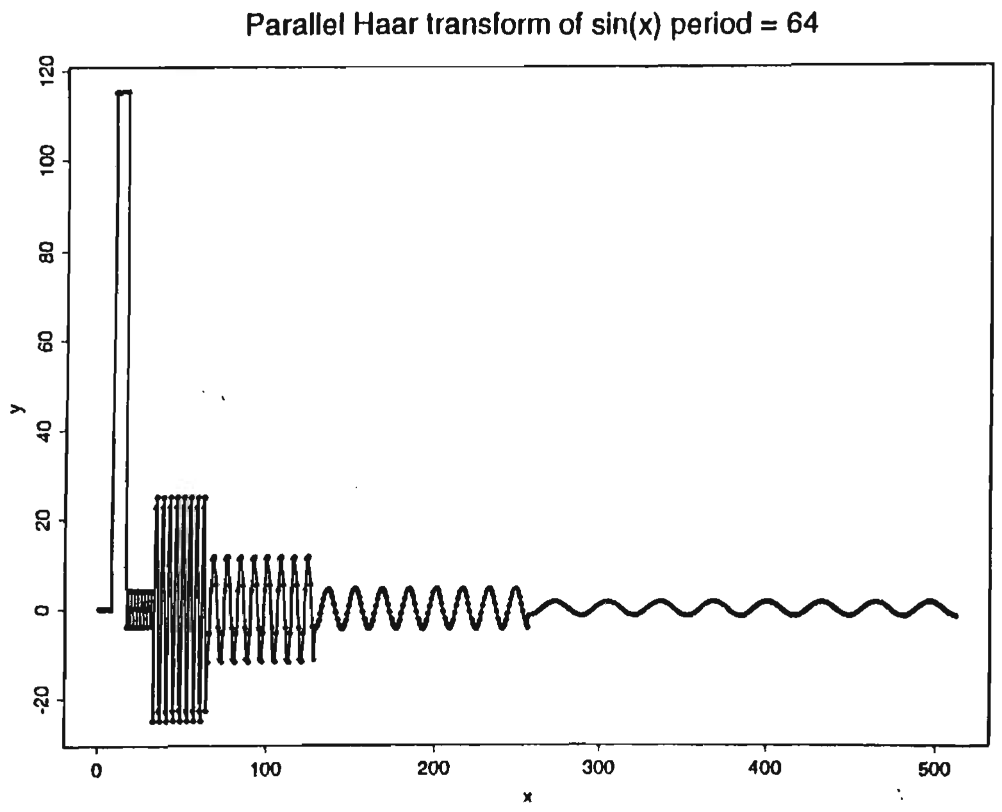
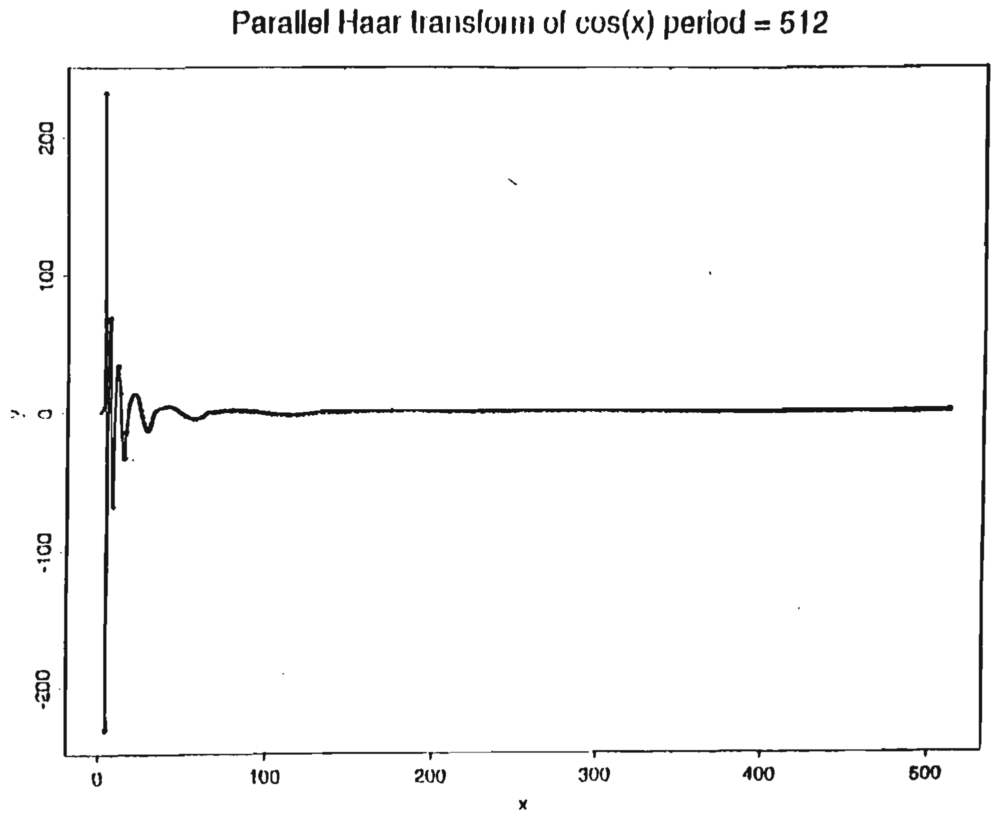


Figure 5.3 Two examples of the discrete Haar transform of the sine function where x = Haar basis function sequence number and y = transform coefficient magnitude.



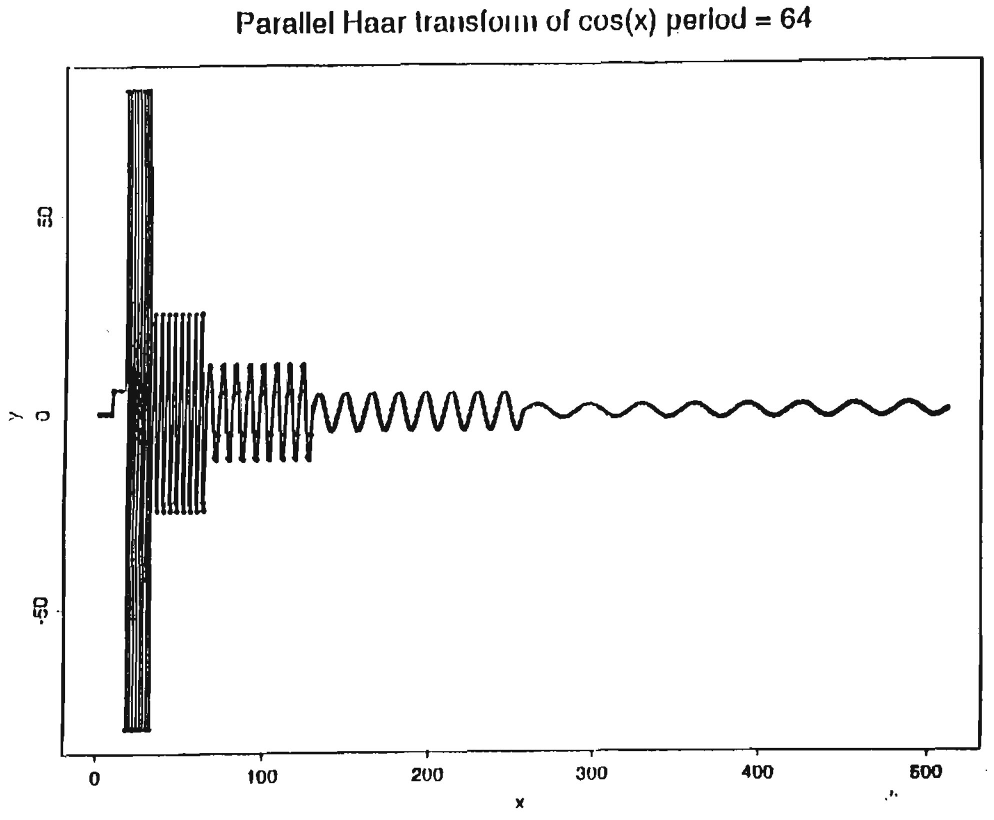
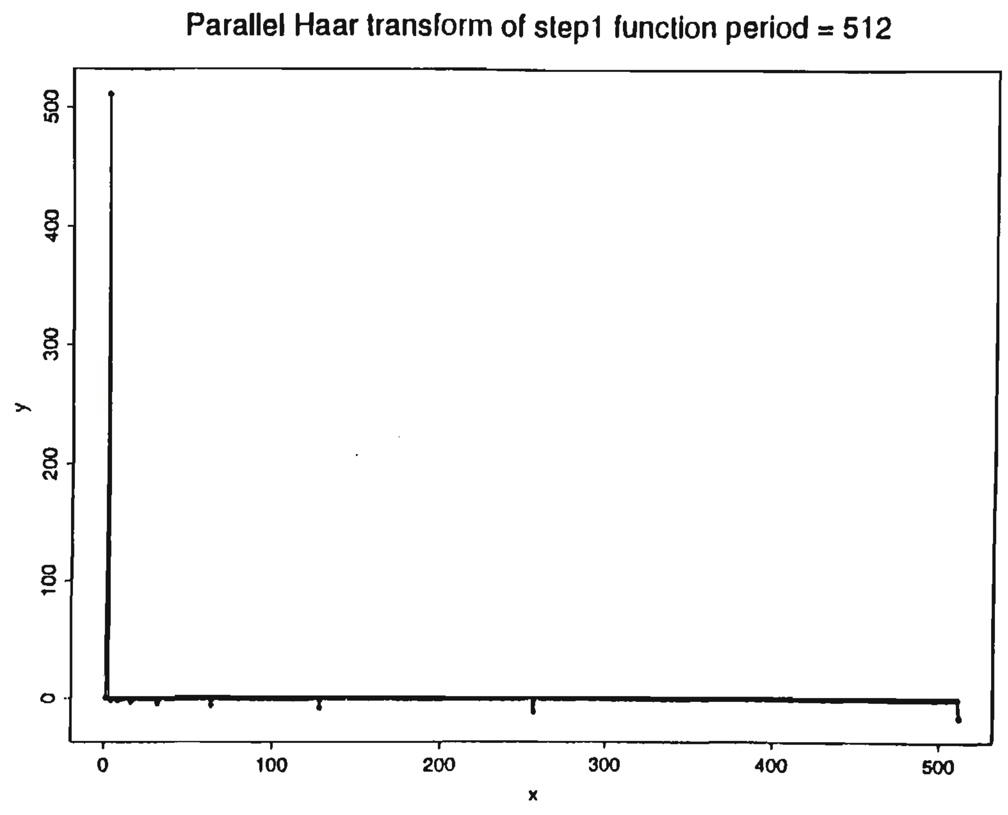


Figure 5.4 Two examples of the discrete Haar transform of the cosine function where x = Haar basis function sequence number and y = transform coefficient magnitude.



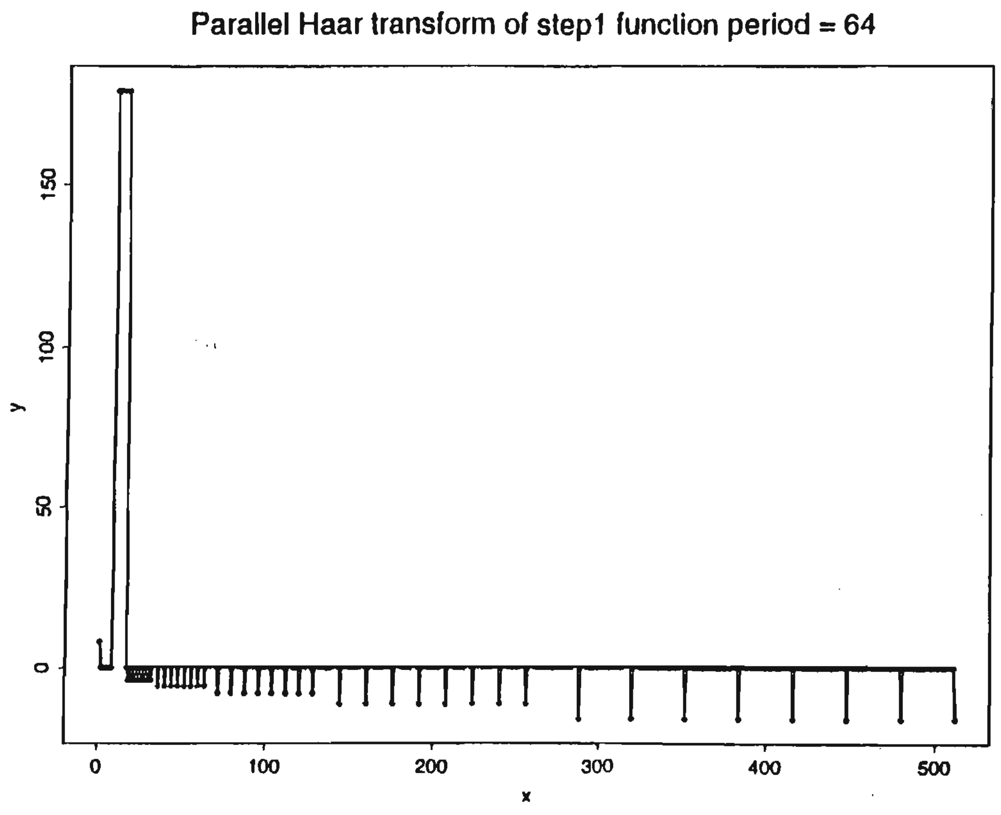
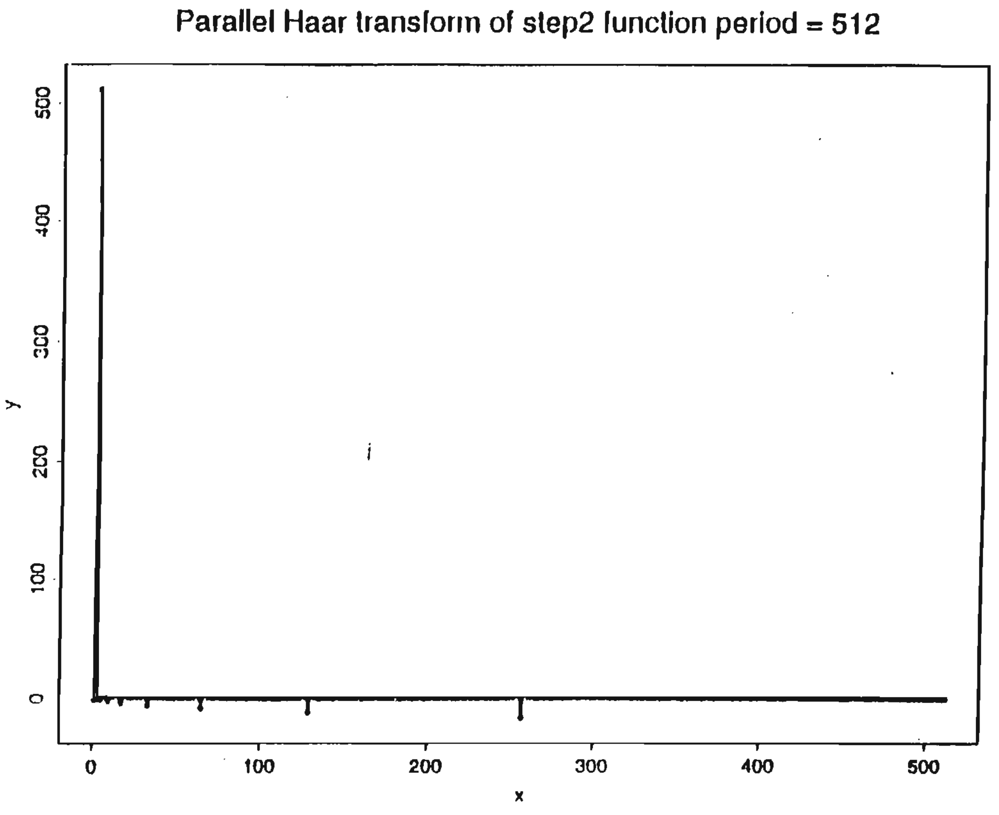


Figure 5.5 Two examples of the discrete Haar transform of the step 1 function where x = Haar basis function sequence number and y = transform coefficient magnitude.



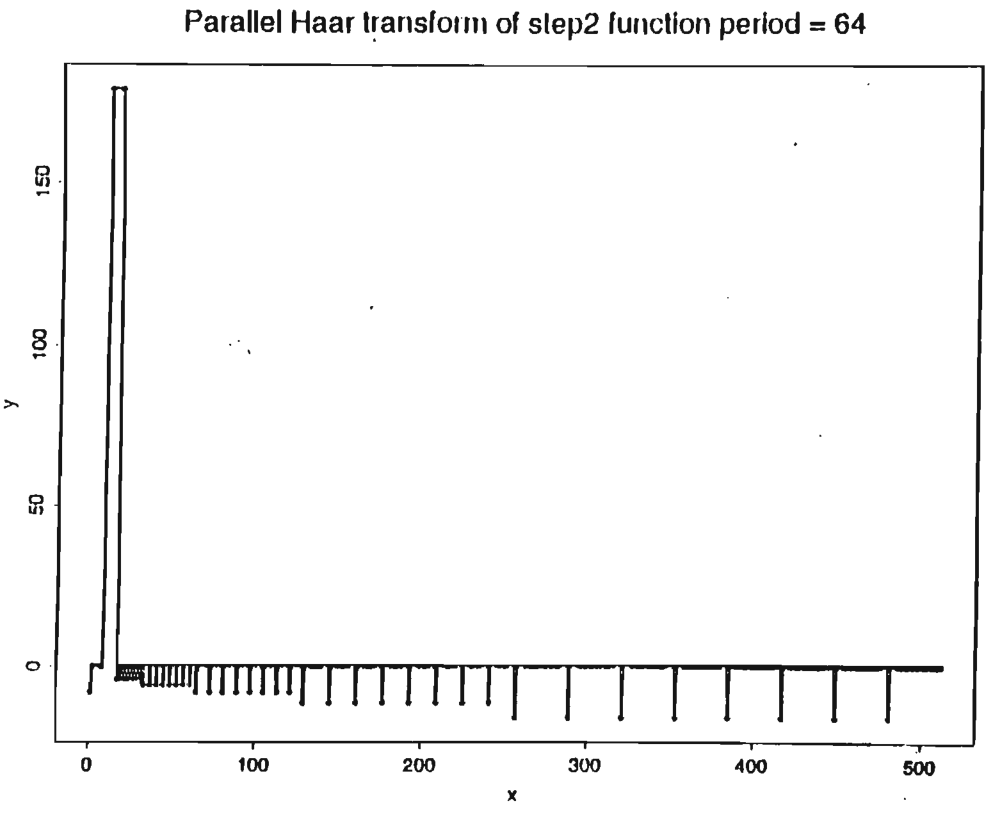
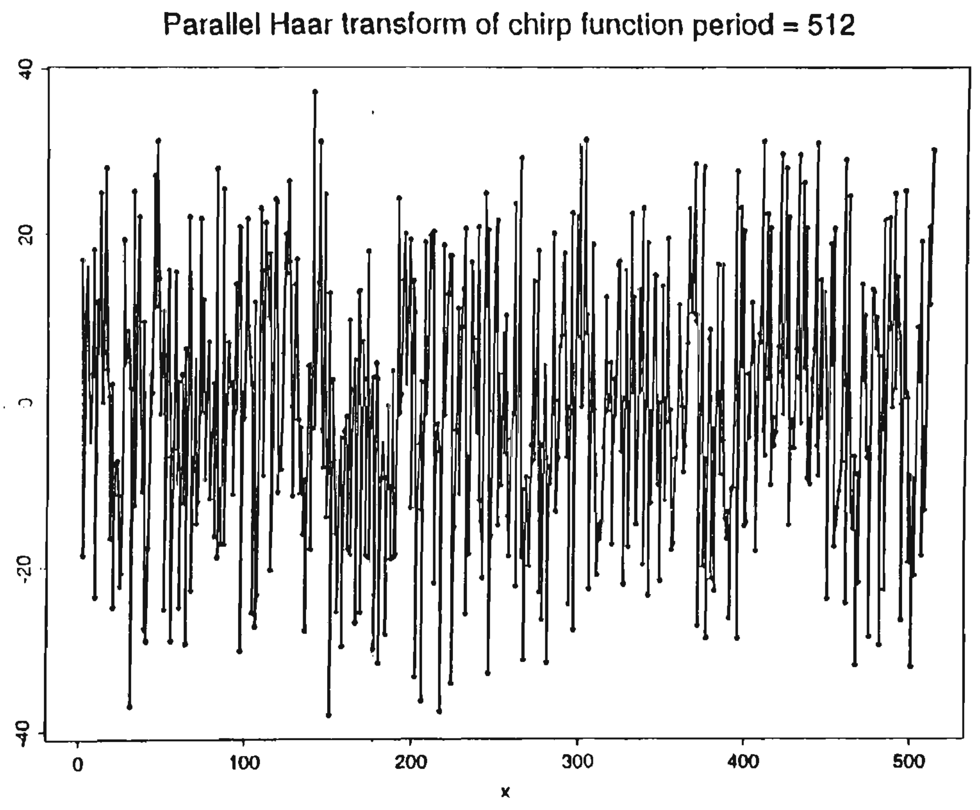


Figure 5.6 Two examples of the discrete Haar transform of the step 2 function where x = Haar basis function sequence number and y = transform coefficient magnitude.



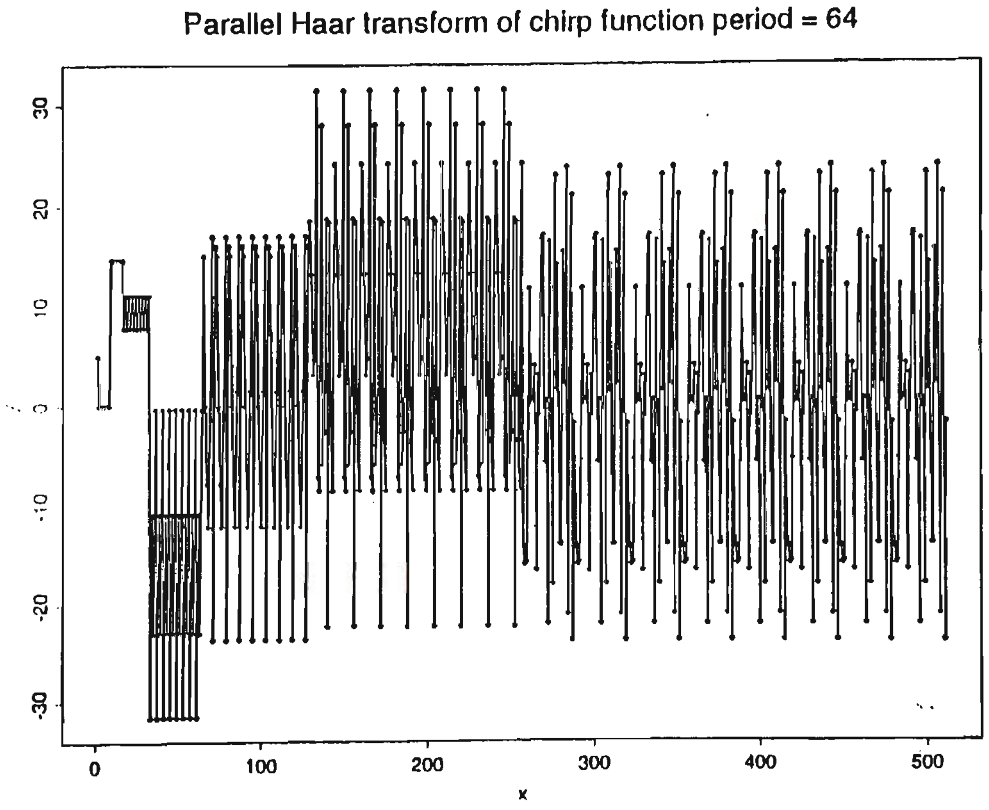
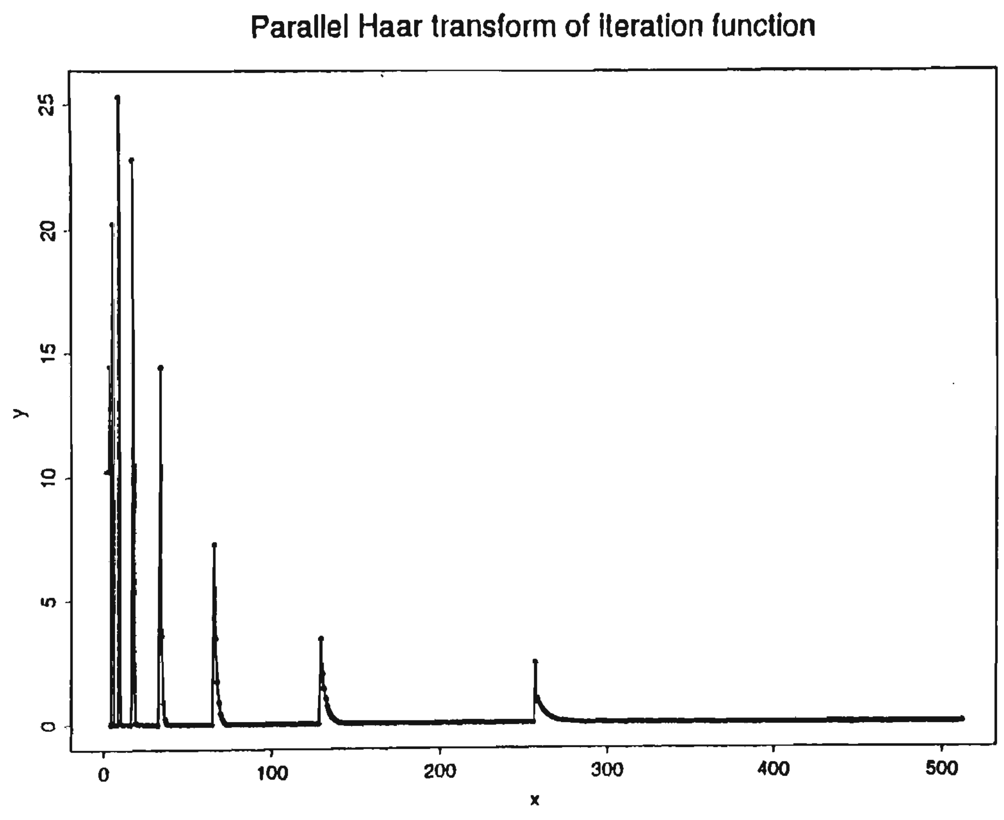


Figure 5.7 Two examples of the discrete Haar transform of the chirp function where x = Haar basis function sequence number and y = transform coefficient magnitude.



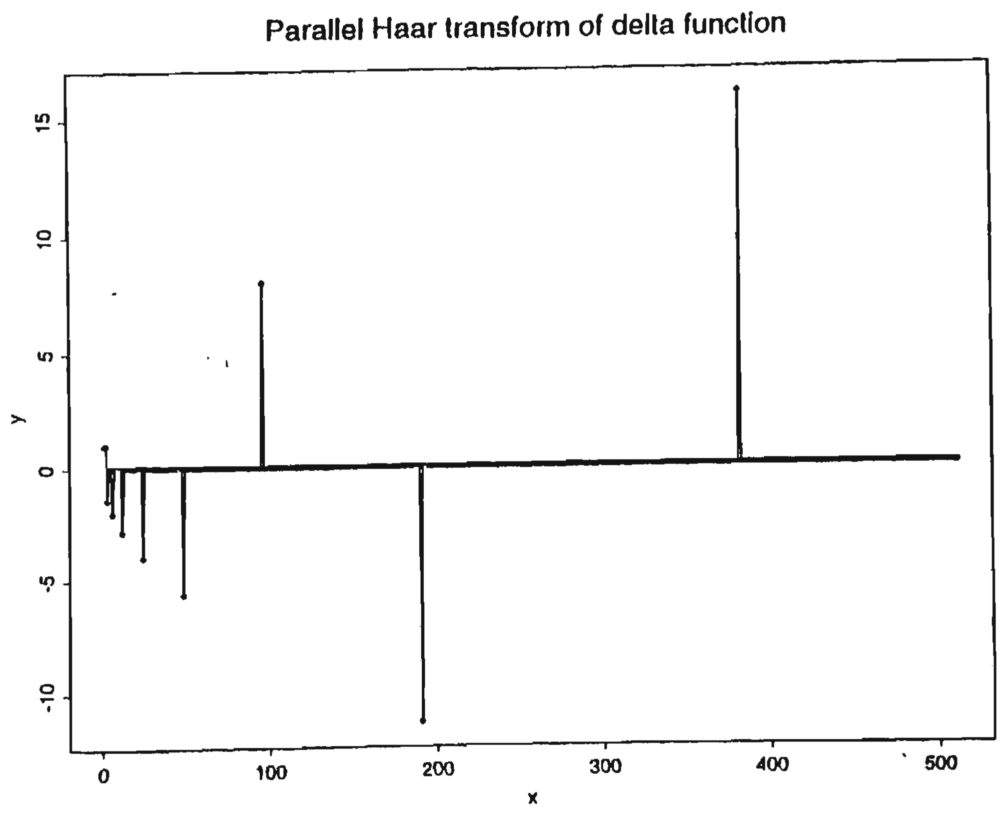


Figure 5.8 Examples of the discrete Haar transform of the iteration and delta functions where x = Haar basis function sequence number and y = transform coefficient magnitude.

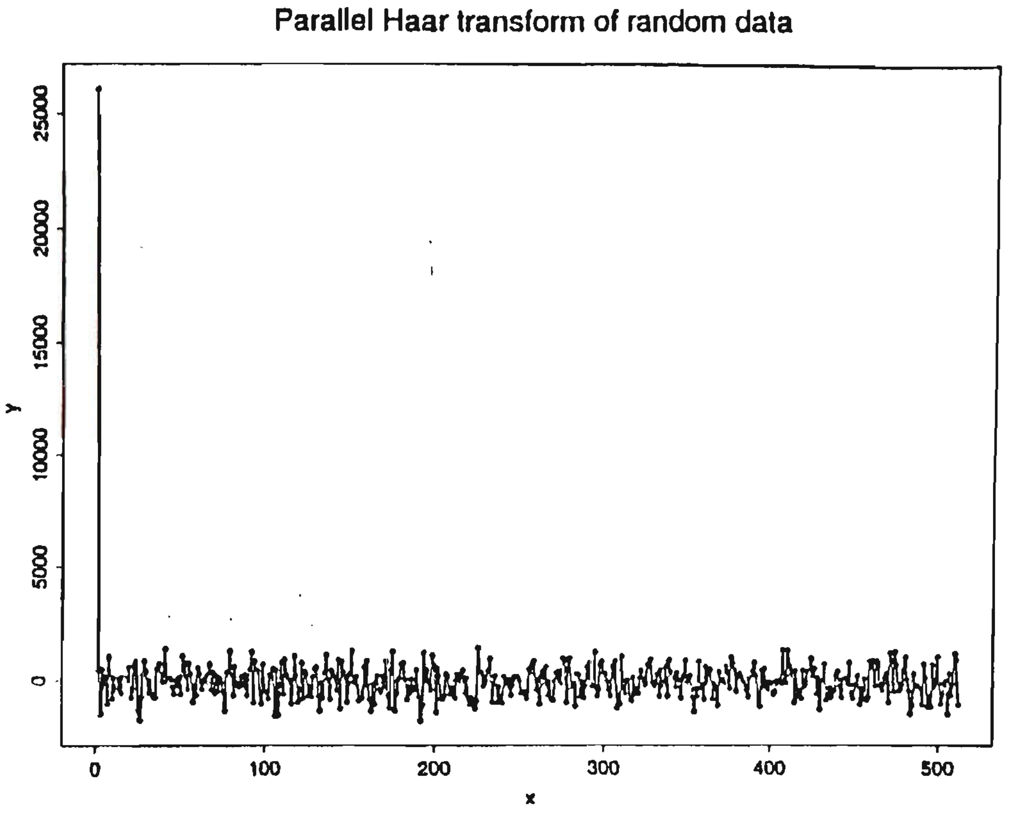


Figure 5.9 The discrete Haar transform of the random data function where x = Haar basis function sequence number and y = transform coefficient magnitude.

5.4 COMPARISON OF SERIAL AND PARALLEL HAAR TRANSFORM IMPLEMENTATIONS

The processor performance of the serial and parallel implementations of the fast Haar transform given in figure 5.10 exhibit the same behaviour as that found for the other transforms under study. The overall execution time for a given processor of the Haar transform is faster than that of the Walsh transform but slower than the D4 wavelet transform. This is in keeping with expectations given the observed theoretical calculation requirements of each of these transforms. The relative performance of the different serial and parallel implementations is the same as that found for the other transforms. As the parallel algorithms employed for the various transforms are dissimilar this reinforces the view that the T800 Transputer as a microprocessor has an inferior performance to the current generation of commercial microprocessors.

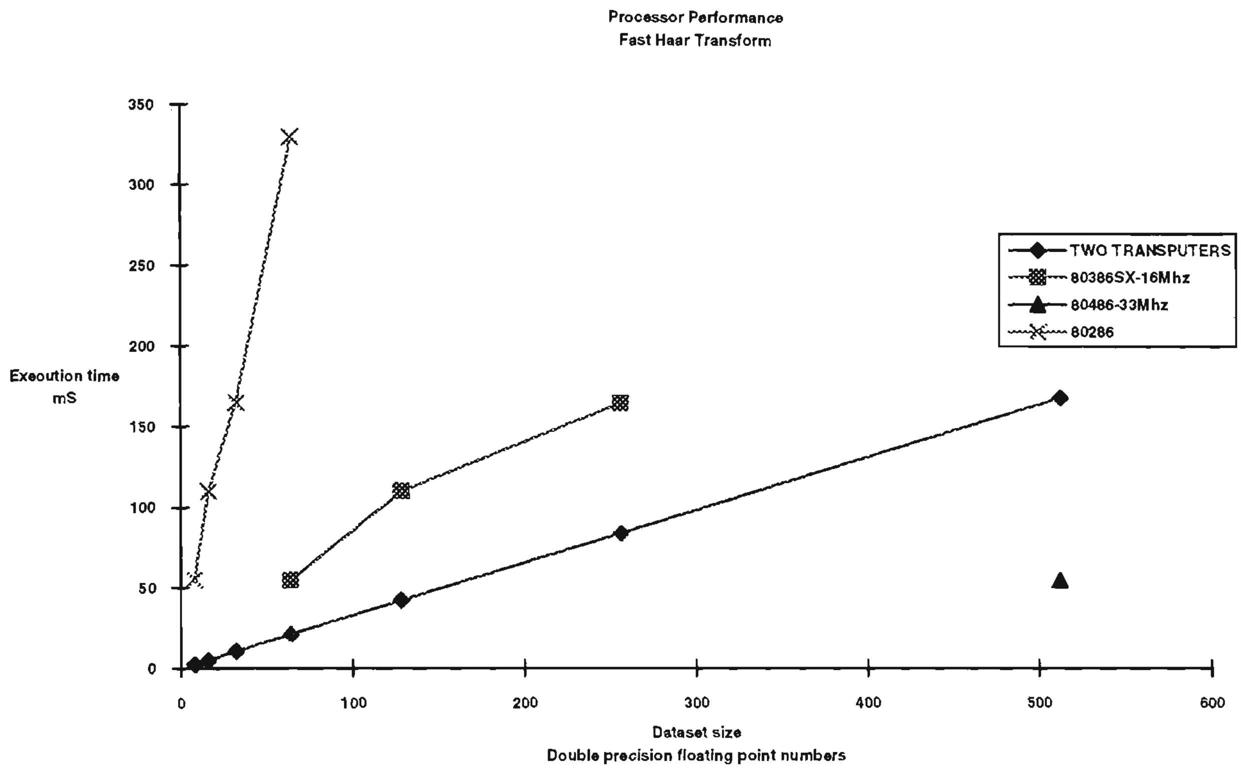


Figure 5.10 Serial and parallel processor performance when implementing the fast Haar transform.

5.5 THE HAAR TRANSFORM AS A PARTICULAR EXAMPLE OF WAVELETS

As was mentioned in section 1.3.3 the basis functions of the Haar transform can be expressed as a set of periodic rectangular waveforms as given by equation(1.29), or they can be generated using the techniques devised for developing wavelet functions. Equation(1.30) shows that a wavelet function sometimes referred to as a "mother" wavelet is created by the summation of a series of translations and dilations of a scaling function ϕ. The scaling function in turn is determined by the iteration of an initial scaling function ϕ_0 (see equation(1.31)).

As outlined by Strang[69] a specific wavelet function is determined by the choice of the initial scaling function ϕ_0 and the coefficients C_k. For example using the box function shown in figure 5.11 as the initial scaling function and assigning the coefficients $C_0 = C_1 = 1$ to equation (1.31) results in an invariant scaling function shown by equation(5.1).

$$\phi_1(x) = \phi_0(2x)+\phi_0(2x-1). \tag{5.1}$$

Using ϕ_1 in equation(1.30) generates the wavelet function

$$\psi(x) = \phi(2x) - \phi(2x-1). \tag{5.2}$$

Figure 5.12 shows this function to be one of the Haar basis functions.

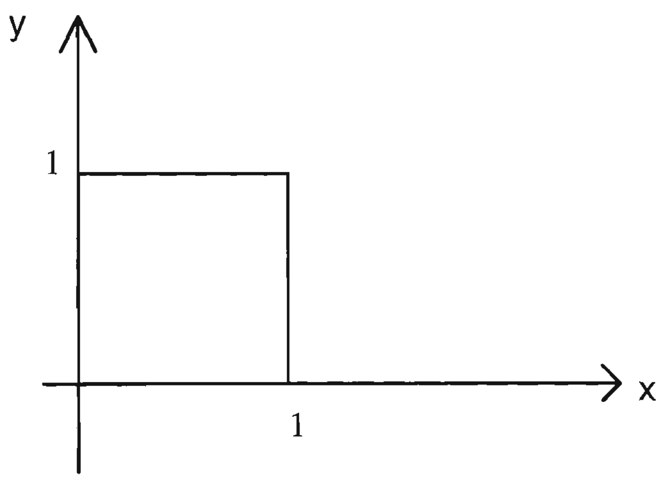


Figure 5.11 Box function used as initial scaling function.

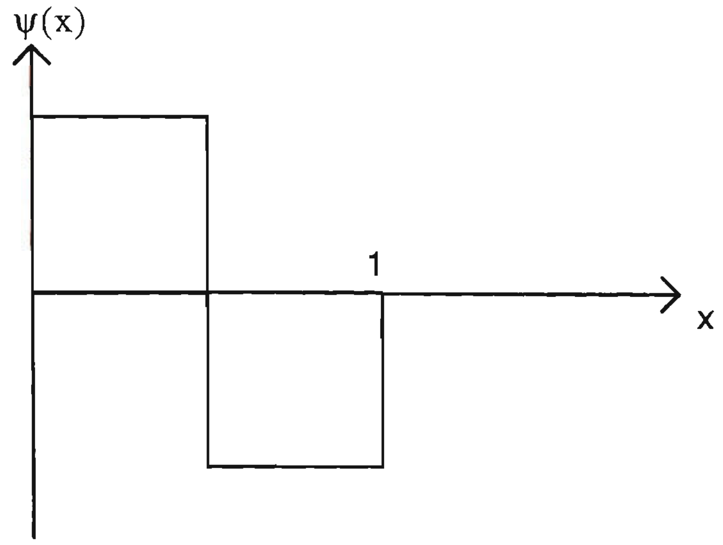


Figure 5.12 Haar wavelet function.

Daubechies[20] shows that for a given wavelet basis function families of wavelet functions consisting of translations and dilations of the basis function may be generated using equation (5.3)

$$\psi_{m,n}(x) = 2^{\frac{m}{2}} \psi(2^m x - n) . \qquad (5.3)$$

The application of equation(5.3) to the Haar wavelet function produces a family of equations some of which are given in equation(5.4). Figure 5.13 shows that these equations represent the Haar basis functions

$$\begin{aligned} \psi_{0,0}(x) &= \psi(x) \\ \psi_{1,0}(x) &= \sqrt{2}\, \psi(2x) \\ \psi_{1,1}(x) &= \sqrt{2}\, \psi(2x-1). \end{aligned} \qquad (5.4)$$

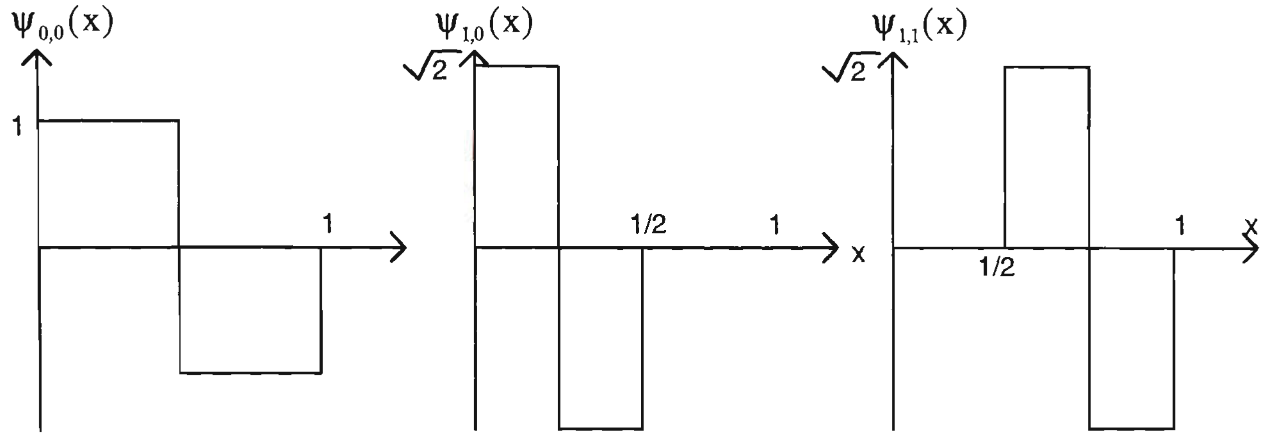


Figure 5.13 First three Haar wavelet functions.

Given these basis functions it is possible to create a discrete matrix of values (equation(1.32)) which can be employed as the coefficient matrix in the discrete Haar transform (equation(1.33)). A wide variety of wavelet functions and consequently wavelet transforms can be created using these techniques.

5.6 PARALLEL IMPLEMENTATION OF A D4 WAVELET TRANSFORM AND COMPARISON WITH A SERIAL IMPLEMENTATION

5.6.1 PARALLEL IMPLEMENTATION OF A D4 WAVELET TRANSFORM

The development of the wavelet transform is outlined by Strang[69]. The discrete Daubauchies D4 wavelet transform is shown by Press[54] to be based on the matrix

$$\begin{bmatrix} c_0 & c_1 & c_2 & c_3 & . & . & . & 0 \\ c_3 & -c_2 & c_1 & c_0 & . & . & . & 0 \\ 0 & 0 & c_0 & c_1 & c_2 & c_3 & . & 0 \\ 0 & 0 & c_3 & -c_2 & c_1 & -c_0 & . & 0 \\ 0 & . & . & . & . & . & . & . \\ . & . & . & . & & . & . & . \\ c_2 & c_3 & . & . & . & . & c_0 & c_1 \\ c_1 & -c_0 & . & . & . & . & c_3 & -c_2 \end{bmatrix}. \tag{5.5}$$

The one dimensional discrete D4 wavelet transform consists of a series of matrix multiplications and vector sort operations. Initially on the full dataset N and subsequently on successive bisections of the data vector. The transform is completed when the dataset to be operated on is reduced to a trivial number usually two. Equation (5.6) demonstrates a D4 wavelet transform given a dataset size N = 8.

step 1:

$$\begin{bmatrix} c_0 & c_1 & c_2 & c_3 & 0 & 0 & 0 & 0 \\ c_3 & -c_2 & c_1 & -c_0 & 0 & 0 & 0 & 0 \\ 0 & 0 & c_0 & c_1 & c_2 & c_3 & 0 & 0 \\ 0 & 0 & c_3 & -c_2 & c_1 & -c_0 & 0 & 0 \\ 0 & 0 & 0 & 0 & c_0 & c_1 & c_2 & c_3 \\ 0 & 0 & 0 & 0 & c_3 & -c_2 & c_1 & -c_0 \\ c_2 & c_3 & 0 & 0 & 0 & 0 & c_0 & c_1 \\ c_1 & -c_0 & 0 & 0 & 0 & 0 & c_3 & -c_2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \end{bmatrix} \rightarrow \begin{bmatrix} z_0 \\ z_2 \\ z_4 \\ z_6 \\ z_1 \\ z_3 \\ z_5 \\ z_7 \end{bmatrix} \tag{5.6}$$

step 2:

$$\begin{bmatrix} c_0 & c_1 & c_2 & c_3 \\ c_3 & -c_2 & c_1 & -c_0 \\ c_2 & c_3 & c_0 & c_1 \\ c_1 & -c_0 & c_3 & -c_2 \end{bmatrix} \begin{bmatrix} z_0 \\ z_2 \\ z_4 \\ z_6 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} \rightarrow \begin{bmatrix} y_0 \\ y_2 \\ y_3 \\ y_4 \\ z_1 \\ z_3 \\ z_5 \\ z_7 \end{bmatrix}$$

The successive halving of the number of operations with each step of the transform in a similar manner to the Haar transform reveals the pyramidal nature of the algorithm. This can be seen by representing the D4 wavelet transform for N = 8 as a signal flow graph as shown in figure 5.14. A graphical representation of the transform assumes a prism-shaped topology.

The application of the D4 transform to larger datasets results in a larger prism consisting of layers of smaller prisms as shown in figure 5.15.

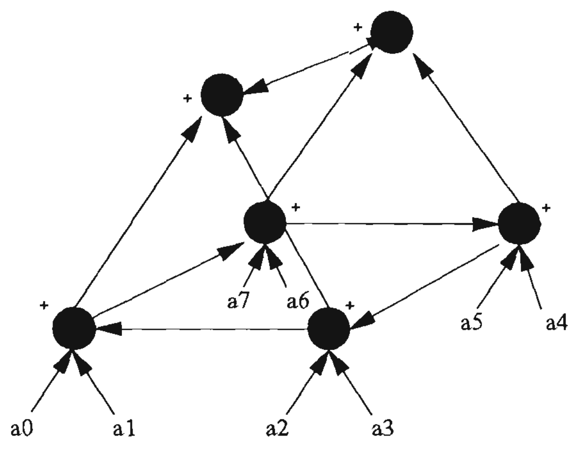


Figure 5.14 Signal flow graph of D4 transform for N = 8.

Multiplication of the data by the appropriate coefficients has been deleted from figures 5.14 and 5.15 as it is intended to provide a general outline of the transform operation. A more detailed examination of the operations at each node is given in figure 5.16.

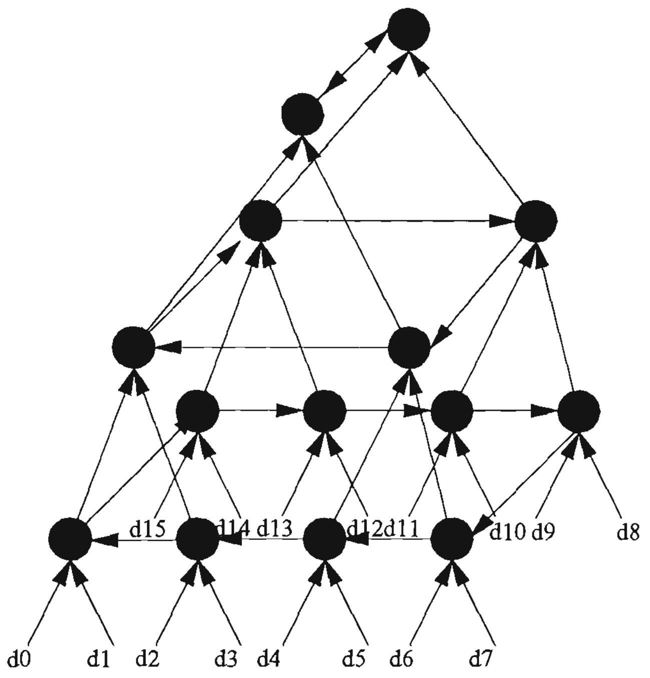


Figure 5.15 Signal flow graph of D4 transform N = 16.

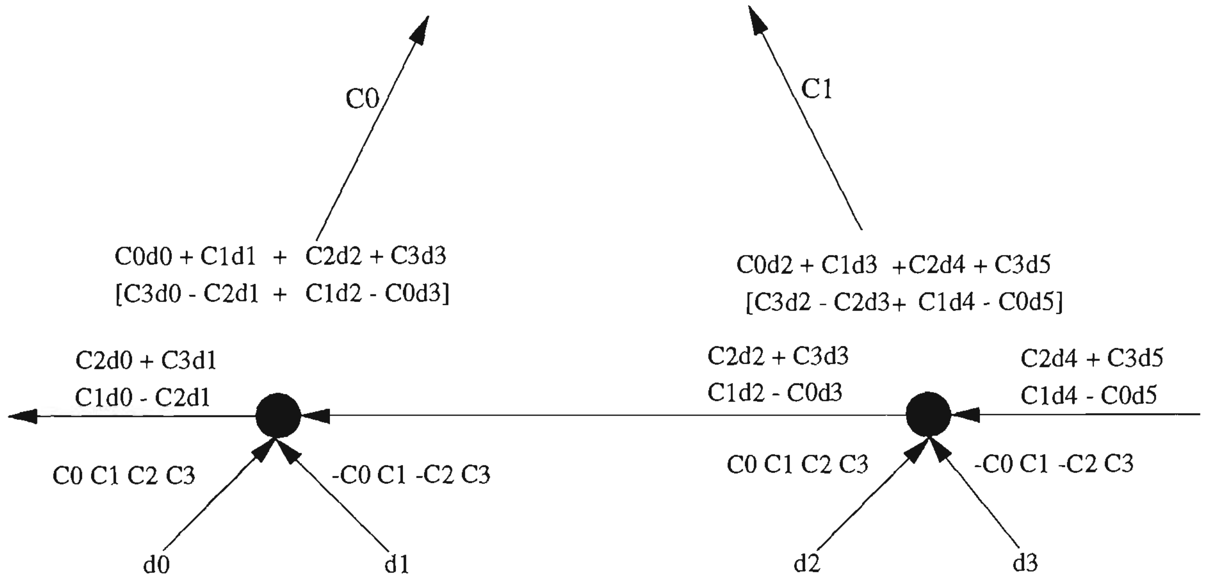


Figure 5.16 A detail of the operations performed at two nodes of the D4 transform.

Figure 5.16 shows the operations that would occur at two nodes in the prism topology. The operations performed at all nodes consist of appropriate coefficient multiplication as shown and addition of terms. The terms shown in square brackets travel along their arcs without being multiplied by the coefficient associated with that arc.

A drawback of this implementation of a parallel D4 wavelet transform is that it requires a relatively exotic processor topology in order to perform the transformation. Embedding the prism topology in a more widely used topology would eliminate this problem. Investigations revealed that it is possible to embed the prism topology in a hypercube structure using a variation on the technique given by Leighton[43] for embedding binary trees in hypercubes.

In order to describe the algorithm for embedding a prism in a hypercube it is first necessary to define terms. The front faces of the prism processor structure resemble a binary tree. It is convenient therefore to borrow from tree terminology and refer to a node closer to the top of the structure as the ancestor node of the nodes connected to it further down the prism. The nodes closer to the base of the prism are descendants of the node to which they are connected higher up the structure.

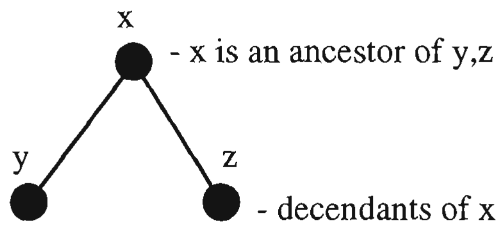


Figure 5.17 Node relationships.

Given these definitions the prism embedding algorithm takes the following form.

a. All ancestor nodes are transferred to a descendant node.

b. Nearest neighbour nodes on the same level within a prism structure must remain nearest neighbours after embedding (see figure 5.19).

The nearest neighbour continuity criterion is met by transferring an ancestor node to a descendant node using equation 5.7, given the node numbering system given in figure 5.18.

$$n_d = 2n_a + \frac{-1^{n_a} + 1}{2}$$
$$n_d = \text{Decendant node} \qquad (5.7)$$
$$n_a = \text{Ancestor node}$$

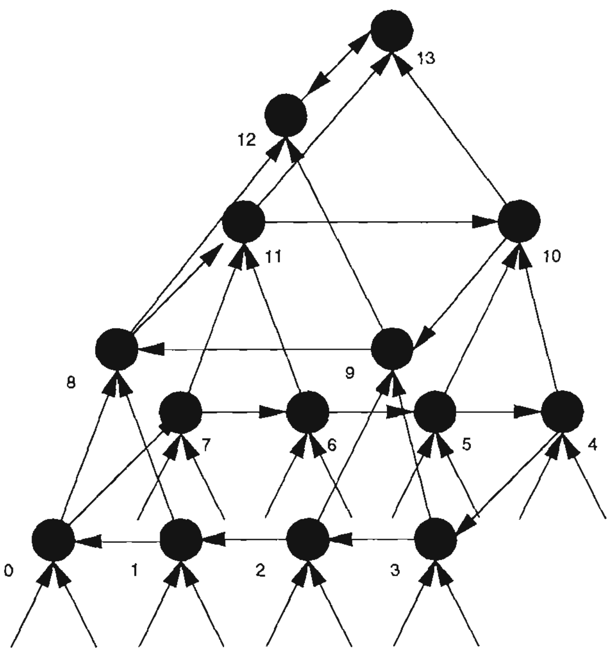


Figure 5.18 Node numbering system for hypercube embedding..

A step by step example of embedding a prism for N = 8 into a hypercube is given in figure 5.19.

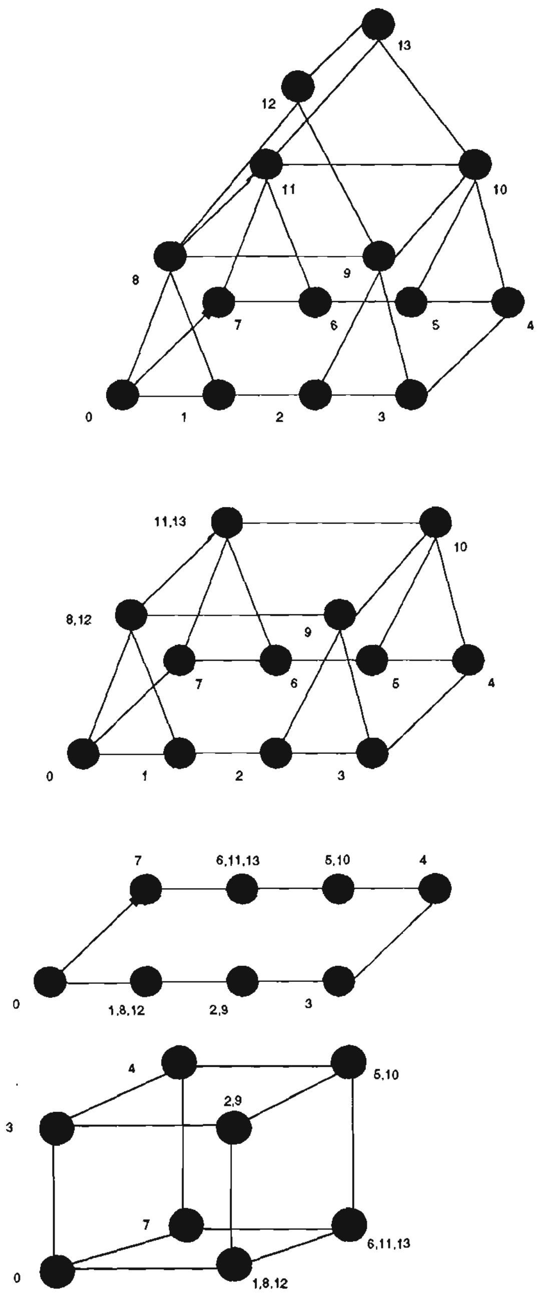


Figure 5.19 Embedding of the prism for N = 8 into a hypercube.

The transfer of data through the prism layers is removed by hypercube embedding as ancestor nodes are transferred to descendant nodes. The number of concurrent data transfers required was found to be

$$\begin{aligned} &\text{concurrent data transfers} = 2n \\ &n = \text{number of prism layers} \end{aligned} \tag{5.8}$$

For a hypercube implementation of the D4 wavelet transform the communication overhead can be seen to grow linearly with the size of the transformation.

Therefore the D4 wavelet transform can be performed efficiently in parallel on the same hypercube topology as was used for the Fourier, Walsh, cosine and Haar transforms. This means that the discrete D4 wavelet or any semi-circulant matrix transform can be performed on the widely used hypercube parallel topology which has also been found to be suitable for the other transforms studied in this thesis.

5.6.2 COMPARISON OF SERIAL AND PARALLEL IMPLEMENTATIONS OF THE D4 WAVELET TRANSFORM

A comparison of the serial and parallel implementations of the D4 wavelet transform is given in figure 5.20.

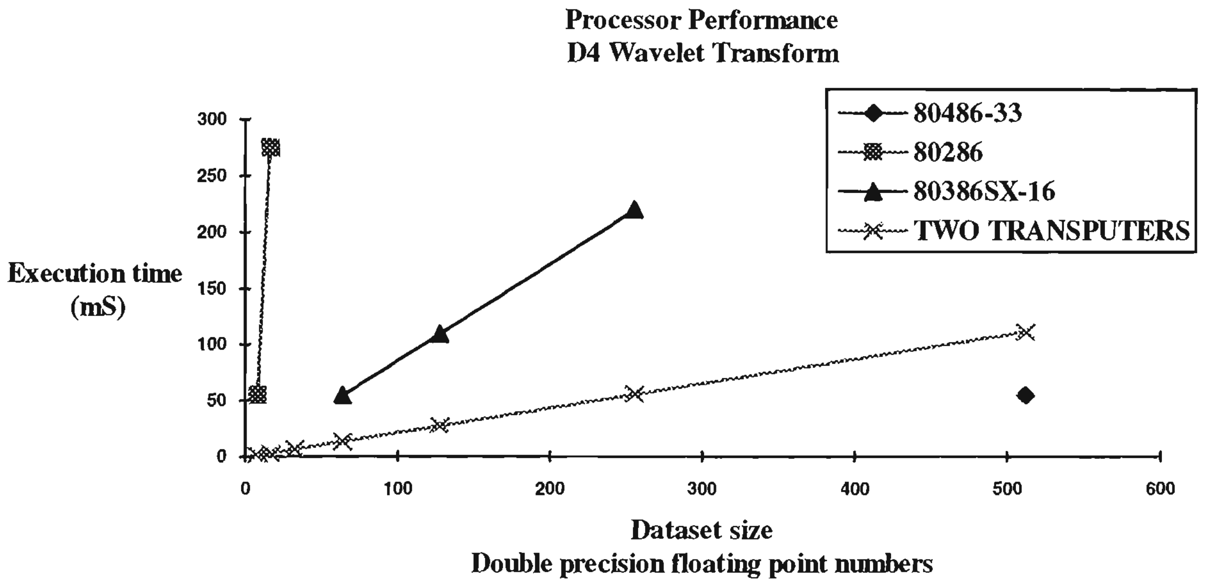


Figure 5.20 Serial and parallel processor performance when implementing the D4 wavelet transform.

The processor performance of the serial and parallel implementations of the D4 wavelet transform are similar to those given for the fast Walsh transform. Overall performance of all D4 wavelet implementations is better than that given for the fast Walsh transform. This is understandable given the greater computational effort required to perform the global Walsh transform in comparison to the wavelet transform which is performed on an ever decreasing scale.

Processor comparisons show once again that two transputers perform faster than the 80386SX microprocessor. The two transputers although being older processors, split the task. Also an on-chip floating point unit operating in parallel with the CPU give it an improved performance over the more modern 80386SX, particularly in situations such as the wavelet transform where the majority of calculations are floating point operations.

The performance of the transputer compared with the 80486 which also possesses on-chip FPU is not impressive. Figure 5.20 shows a marginal improvement over the fast Walsh transform performance. This can be attributed to the lower communications overhead of the D4 wavelet transform as shown by comparing figure 5.21 and figure 3.20. Given the linear growth in communication overheads for the hypercube implementation given in 5.6.1, figure 5.20 indicates that a hypercube consisting of four or more T800 transputers would give a performance equal to or better than the 80486 for performing the D4 wavelet transform. This estimate would be less for the more modern T9000 transputer.

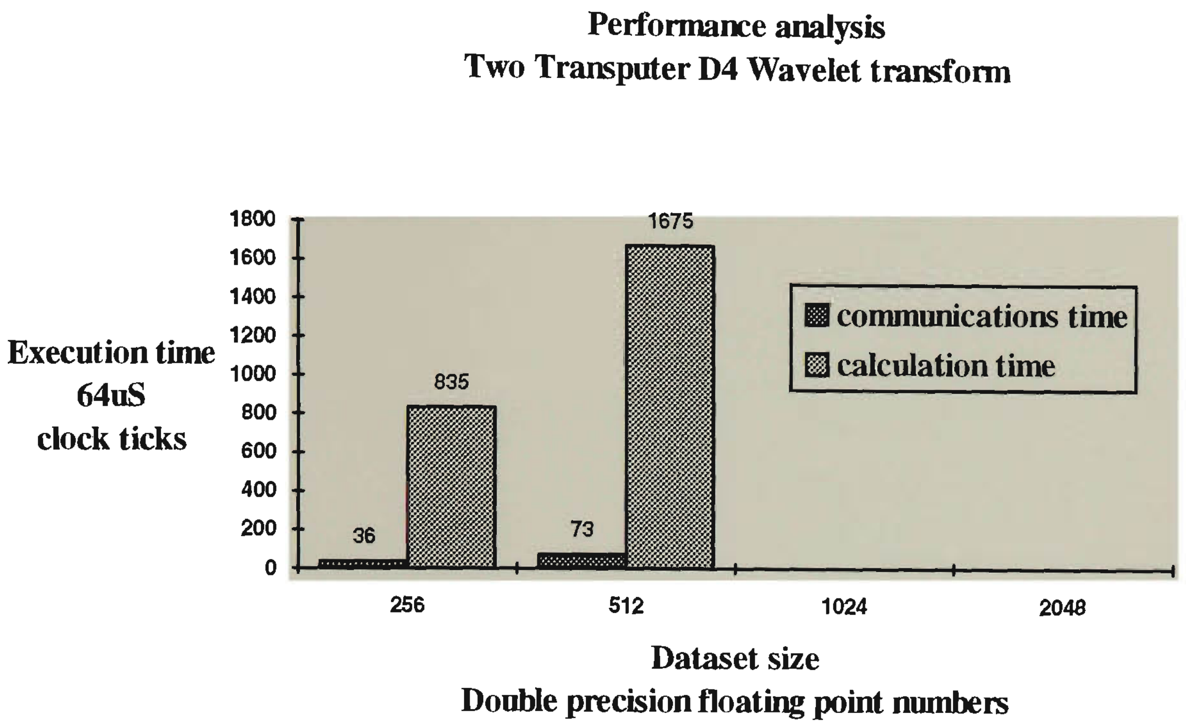


Figure 5.21 An analysis of computational resource demand by major operations within a two transputer D4 Wavelet transform implementation.

5.6.3 APPLICATION OF THE D4 WAVELET TRANSFORM TO PERIODIC AND NON-PERIODIC FUNCTIONS

In many instances the spectra resulting from the D4 wavelet transformation were similar to those of the Haar transform. Like the Haar transform the locally defined basis functions of the D4 wavelet transform provide it with a sensitivity to local singularities or transient signals. This can be seen in the D4 wavelet transform of the delta function. In the case of smooth periodic and non-periodic functions such as the trigonometric and iteration functions the D4 wavelet transform produced more spectral components than the globally-defined transforms but fewer than the Haar transform. Consequently the D4 wavelet transform may be better at identifying transient signals embedded in smooth signal fluctuations. The step function transforms were not as clear as those from the Haar transform due to the close similarity between the test function and the Haar basis functions. As with all the transforms examined functions with large numbers of discontinuties such as the chirp and random noise functions provided a D4 wavelet transform with many components and no clearly identifiable features.

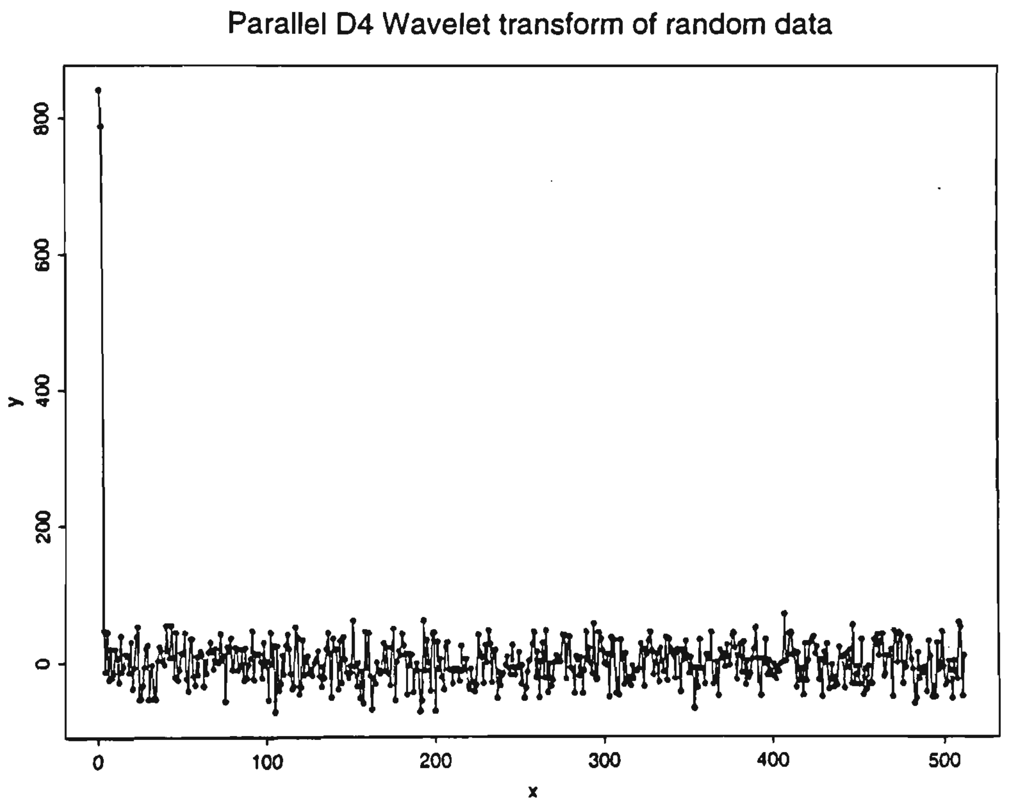
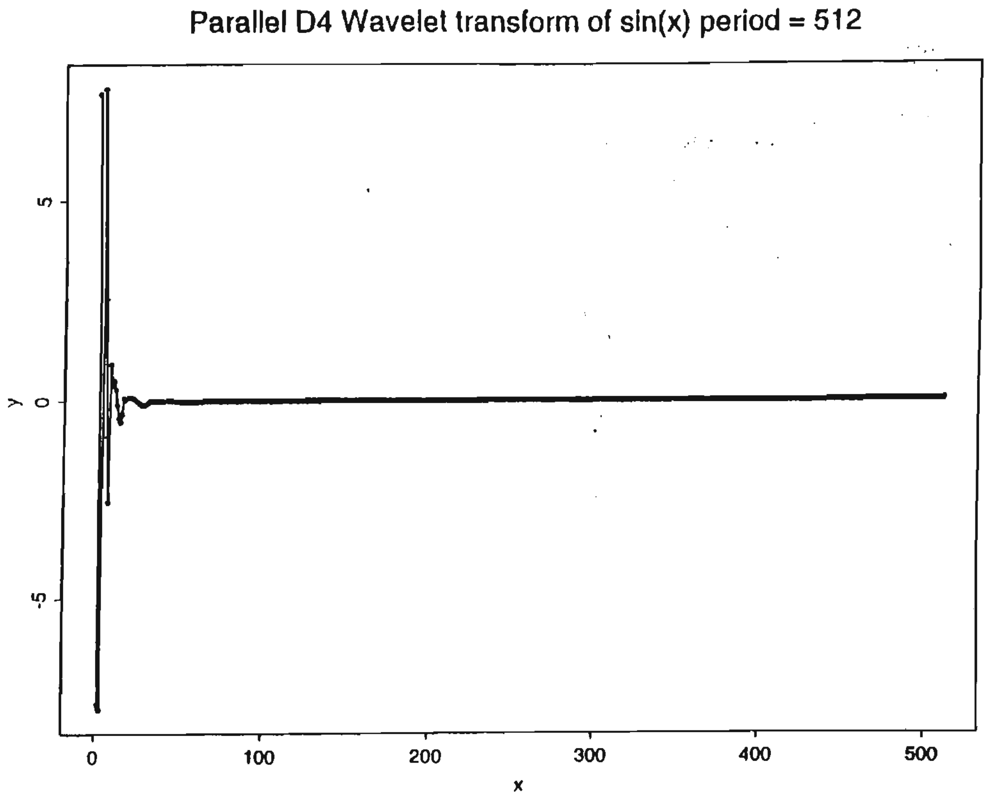


Figure 5.22 D4 Wavelet transform of random data.



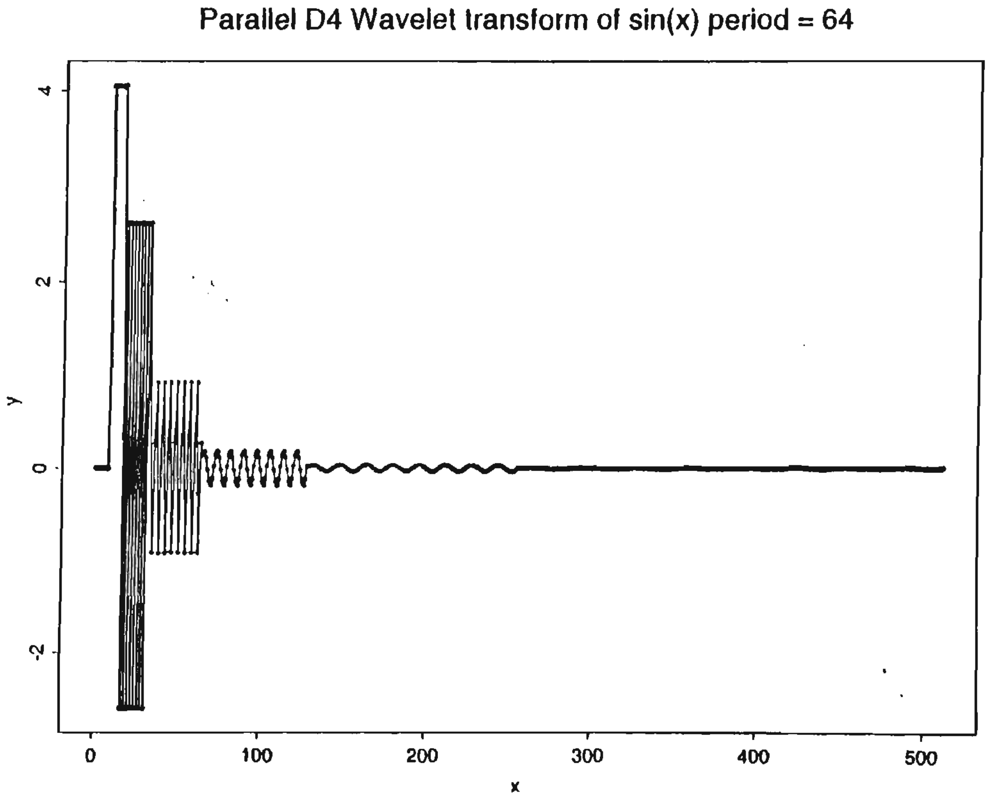
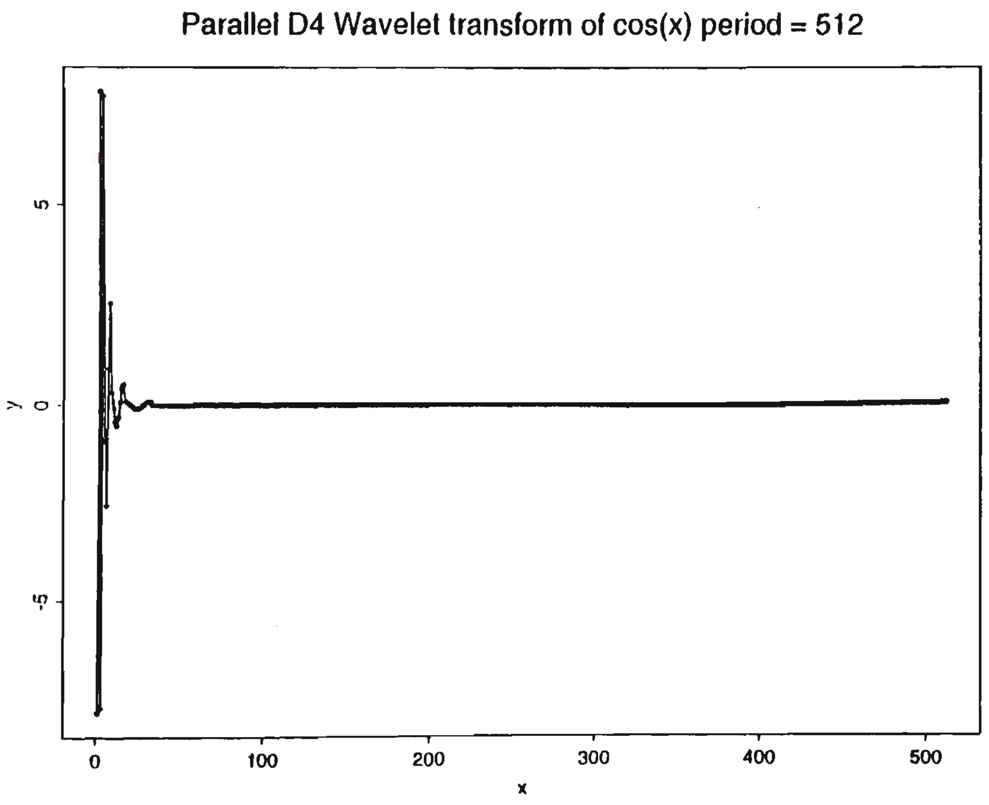


Figure 5.23 D4 Wavelet transform of the sine function.



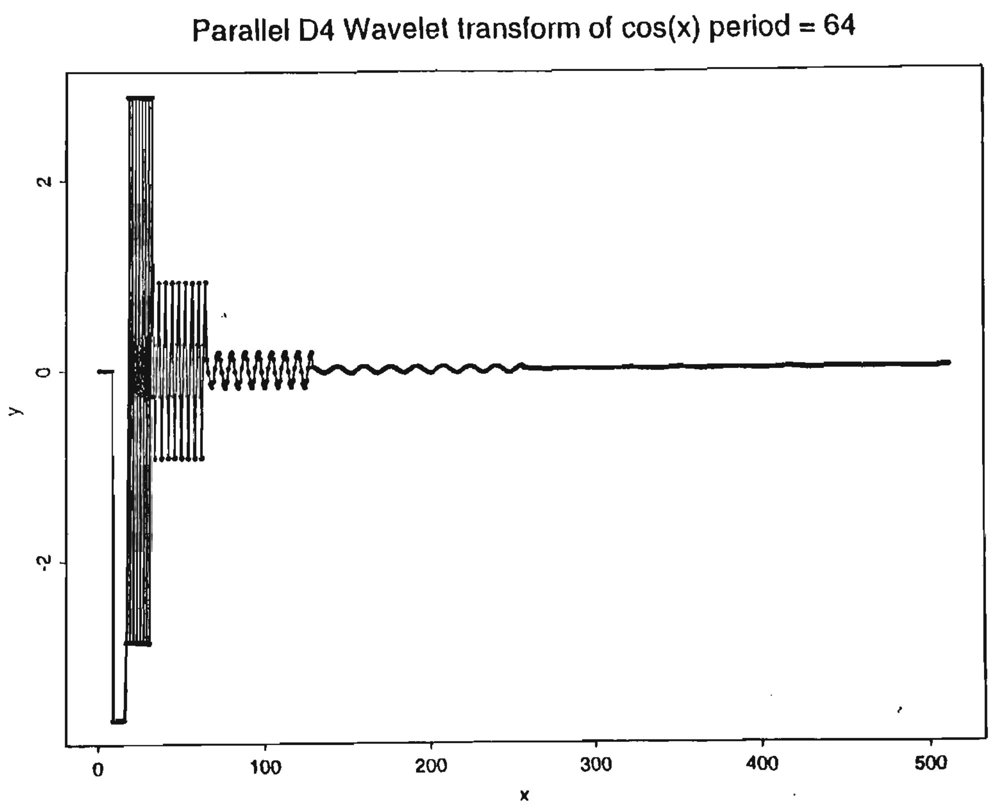
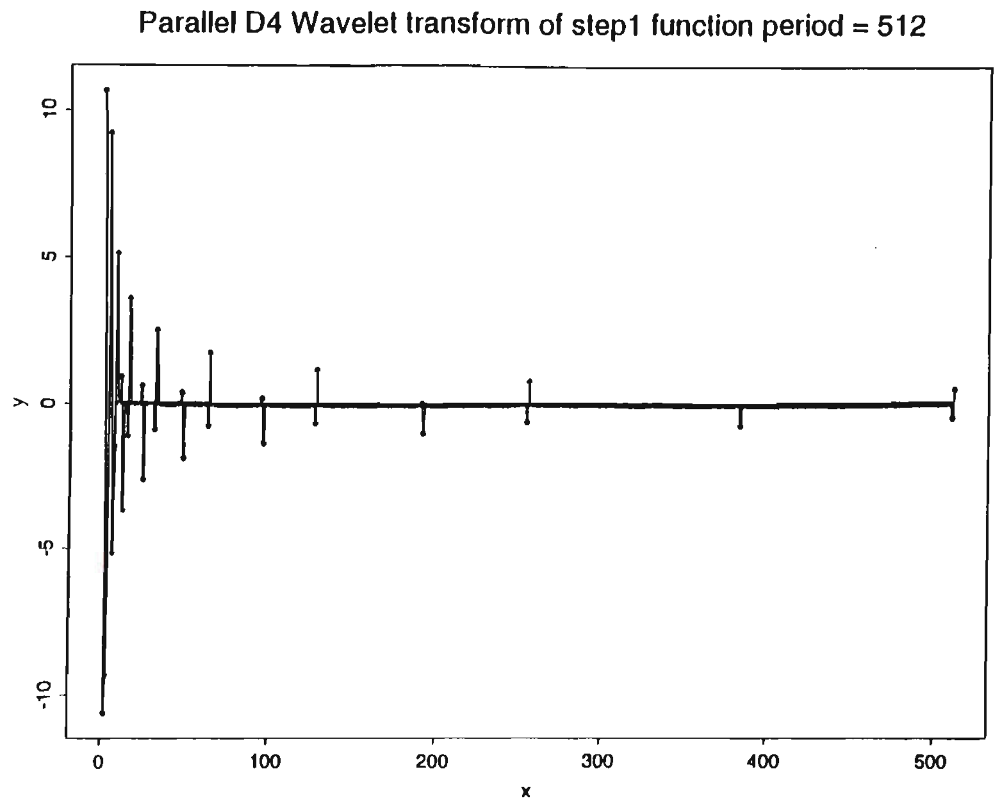


Figure 5.24 D4 Wavelet transform of the cosine function.



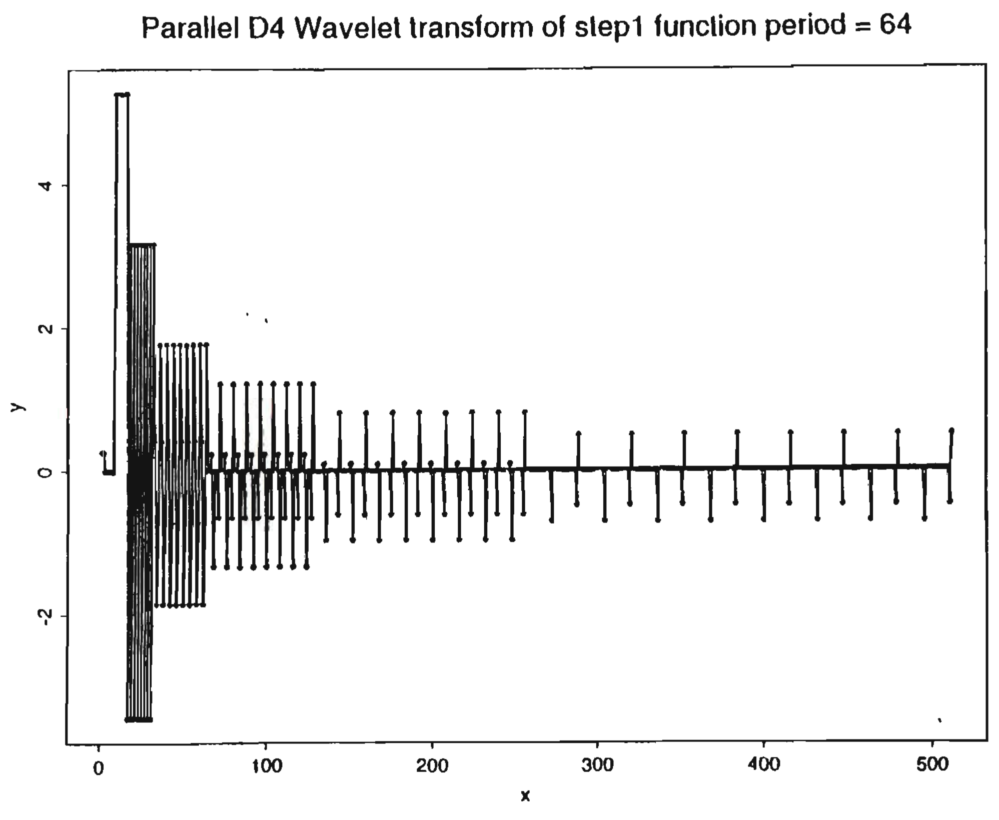
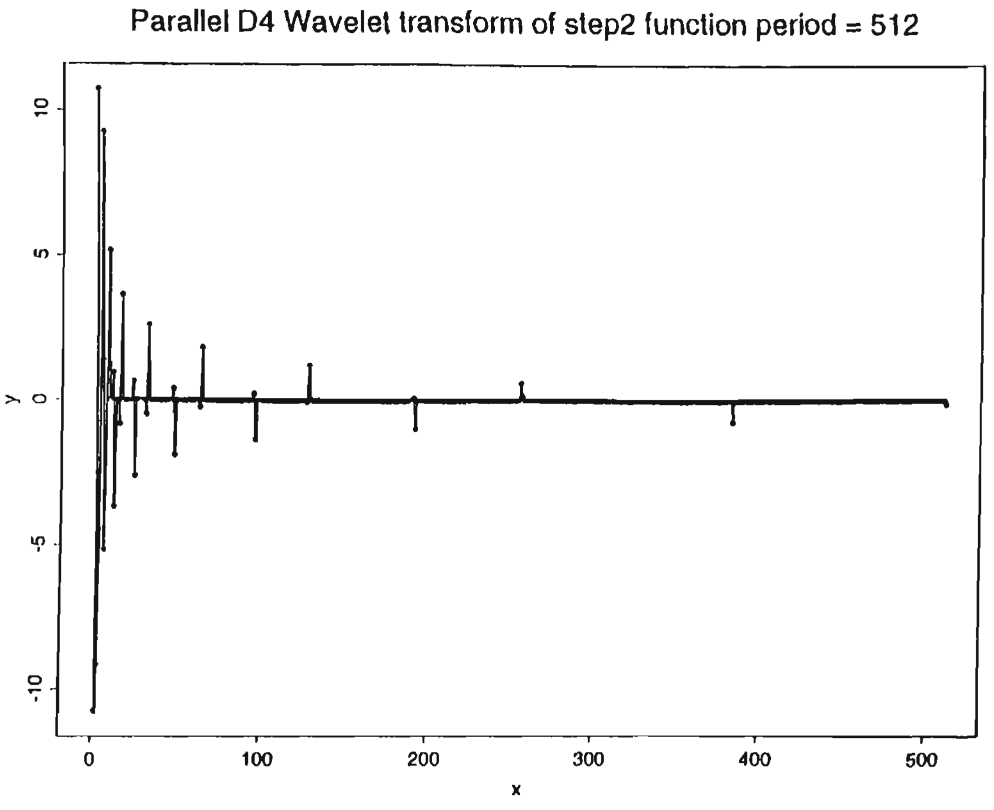


Figure 5.25 D4 Wavelet transform of the Step1 function.



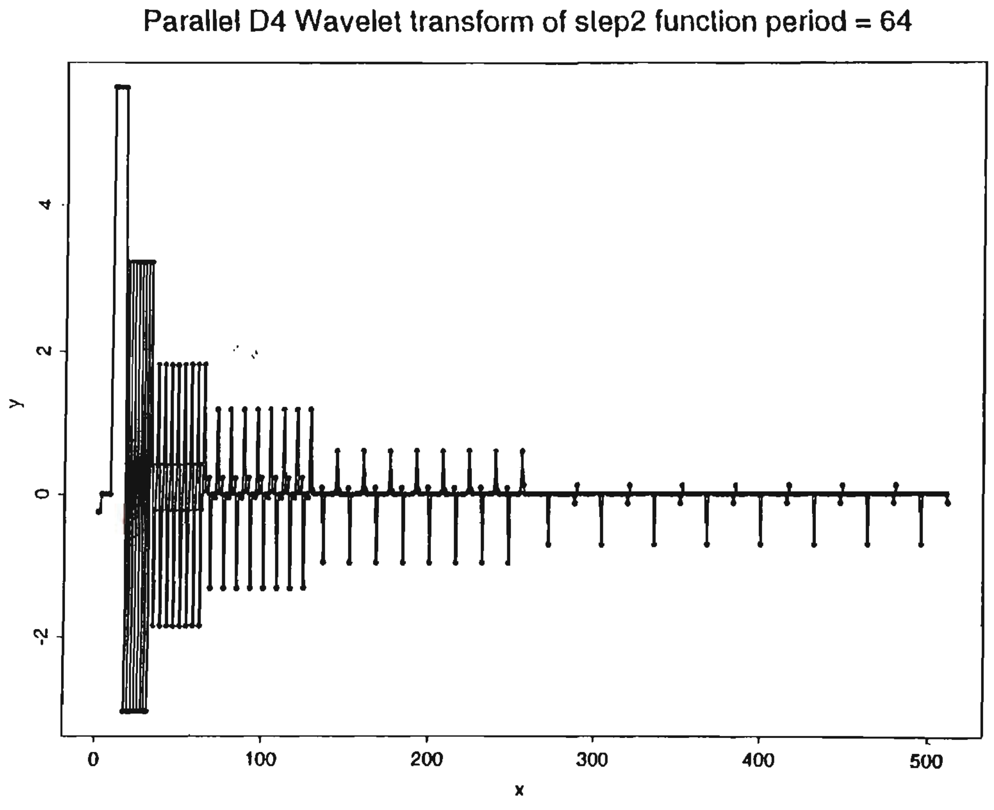
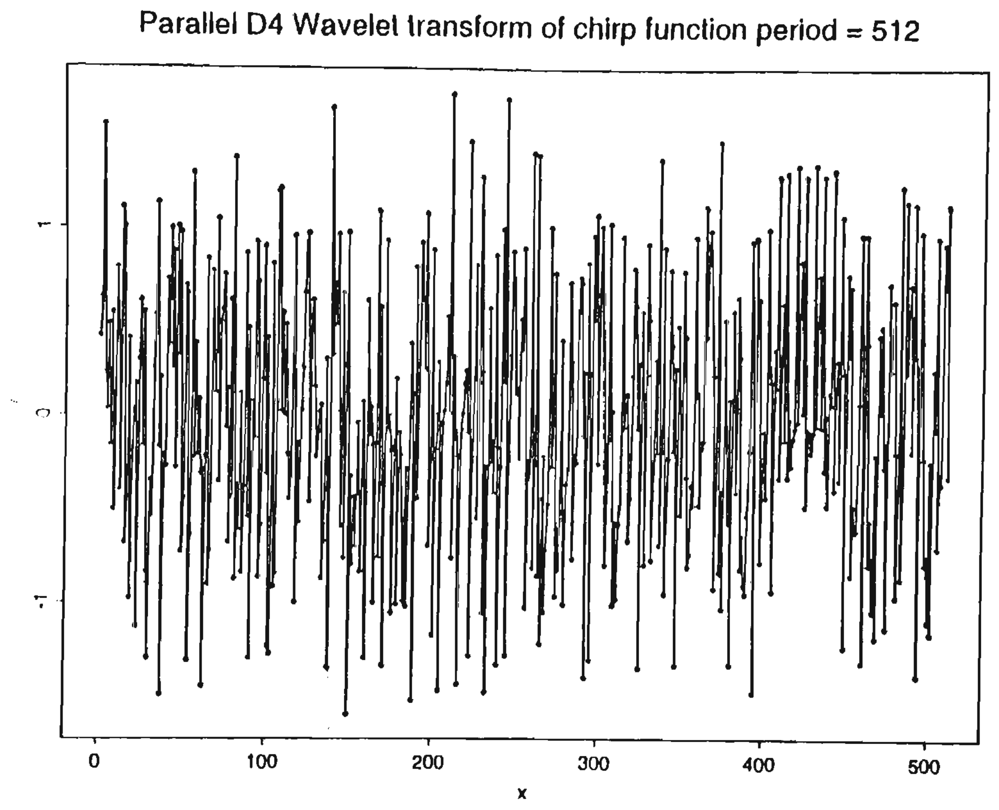


Figure 5.26 D4 Wavelet transform of the Step2 function.



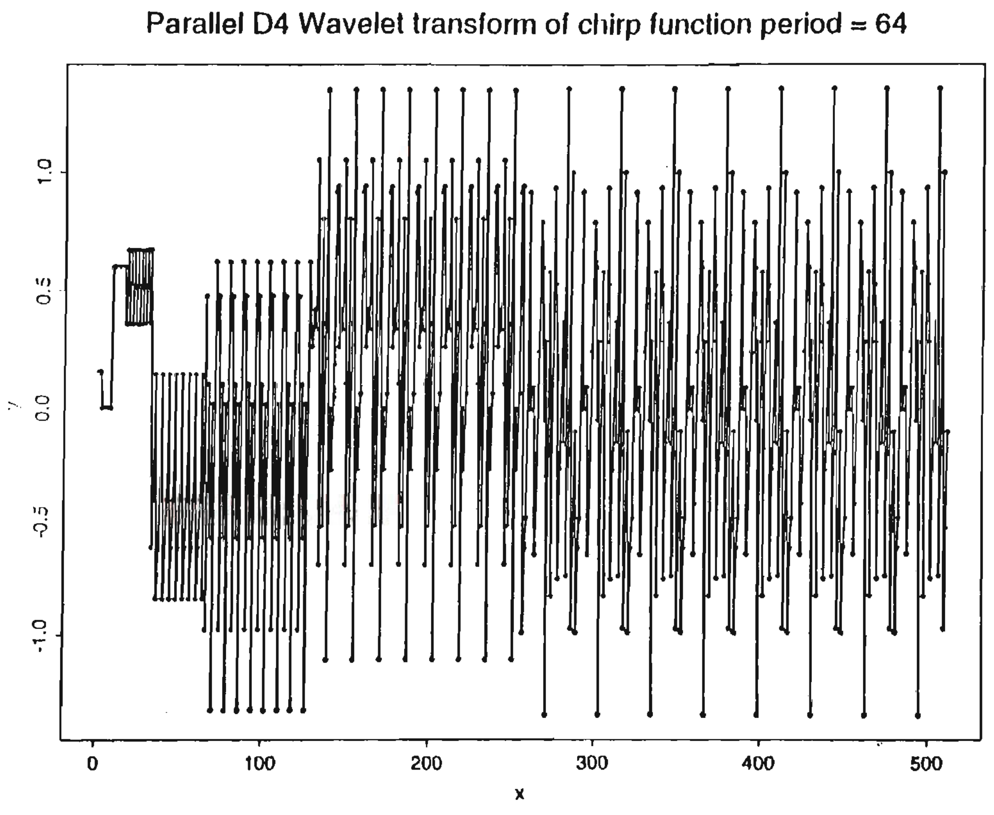
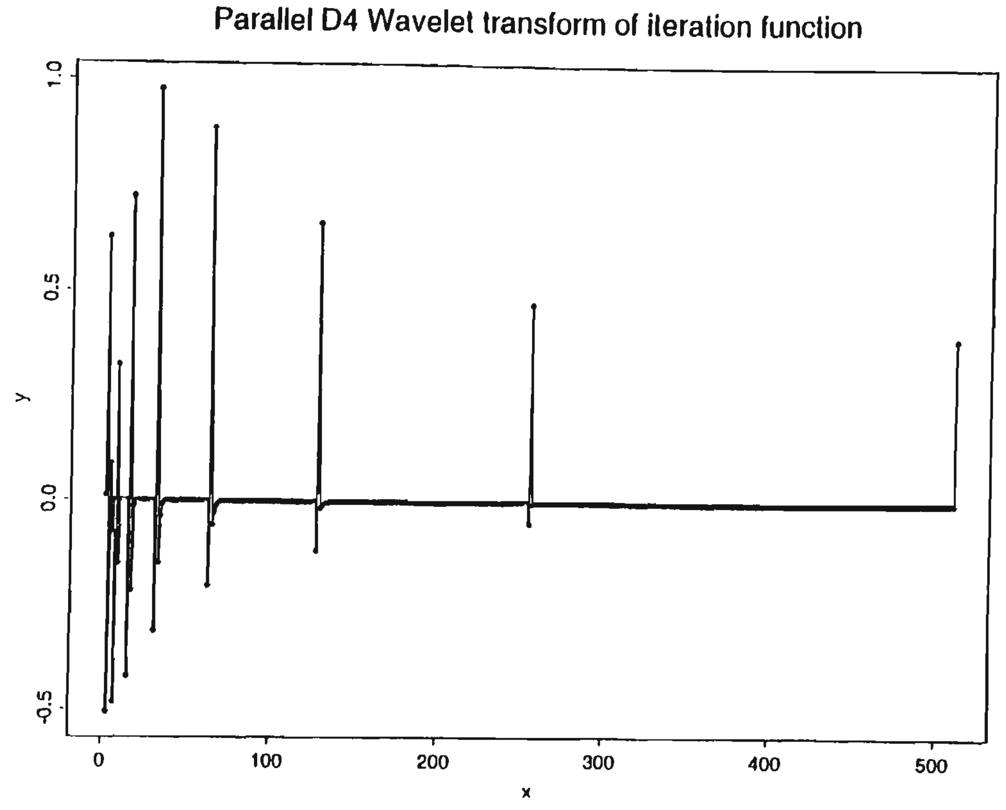


Figure 5.27 D4 Wavelet transform of the Chirp function.



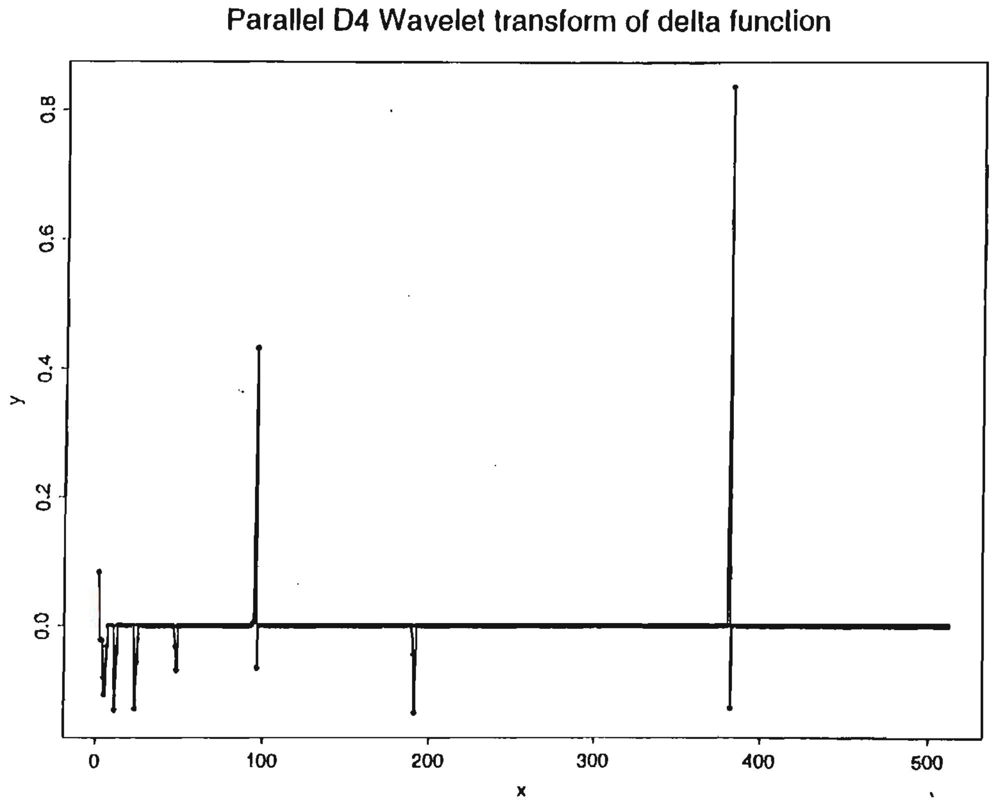


Figure 5.28 D4 Wavelet transform of the Iteration and Delta functions.

5.7 SUMMARY

This chapter has shown that the Haar transform can be implemented in parallel using a binary tree topology which can be easily embedded in a hypercube processor topology. Also the D4 wavelet transform can be represented by a pyramidal or prism shaped structure which also can be embedded into a hypercube in a manner similar to that used to embed binary trees.

The performance comparisons of both the D4 wavelet and Haar transforms on Transputers and commercial microprocessors were similar to that found throughout this thesis. In order to provide performance superior to standard personal computers Transputer systems of four or more Transputers are required.

Both transforms demonstrated a capability to identify transient signals as well as providing interpretable spectra of smooth periodic functions. The D4 wavelet transform produced a similar spectra to that of the Haar transform. But for most of the test waveforms produced fewer spectral components, demonstrating a greater ability to compress data than the Haar transform. Both transforms could not provide a clear spectra of highly discontinuous functions.

CHAPTER 6

CONCLUSION

6.1 INTRODUCTION

The aim of this thesis has been to determine parallel processing algorithms and architectures for a number of representative discrete transforms in order to improve transform processing performance. Another aspect was to investigate the ability of these discrete transforms to detect various types of features in signals. This chapter details the conclusions drawn from this work and suggests possible areas of future investigation.

6.2 CONCLUSIONS AND SUGGESTIONS FOR FUTURE WORK

Discrete transformations can be used to facilitate the detection of features in a signal. The closer the similarity between the feature being searched for and the transform basis functions the simpler the spectrum. The ability of a transform to detect a particular type of feature therefore is dependant upon its basis function set. This was demonstrated by the spectrum of localised waveforms produced by transforms with global basis function sets. Transforms with basis function members which were also local in extent were more successful in detecting localised features.

It has been shown that it is possible to implement discrete transforms either in software as programs or in hardware as purpose designed microchips. The hardware implementation has the advantage of speed, being faster than the software implementation. But hardware designs are dedicated to one transform and operate on small datasets whereas a software implementation can be more generalised. A number of different transforms can be implemented in software to allow optimal performance when attempting to detect different features. The choice between these two alternatives is dependant upon the application requirements.

The software alternative can be increasingly employed in areas currently requiring a hardware solution by improving its performance. This can be brought about by applying parallel processing techniques. Investigations conducted during the course of this thesis have found that a number of different parallel implementations of transforms are possible, the form of the implementation causing great variations in processing performance. The major pitfalls in many of the inefficient algorithms being large communications overheads, communications contention or unbalanced processor loading. Another common complaint is that even if the parallel algorithm is efficient and reduces processing time it requires a special machine or processor topology which is unlikely to find use outside that particular application. This makes the parallel solution no better than the customised chip alternative.

It was found that the transformations reviewed in this thesis, which are representative of a wide range of commonly used transforms, were all amenable to efficient implementation on a hypercube processor topology. A number of parallel algorithms were developed for this type of architecture with none demonstrating the contention or load unbalancing problems mentioned above. Also the hypercube is a common parallel configuration which can be found in a number of commercial computers. This provides two benefits. Increased processing power, allowing the execution of signal transforms and associated operations such as template matching in real time. Secondly the ability to perform a number of transformations concurrently, enhancing the ability to identify a number of disparate features in a signal or image.

While investigating the parallelisation of transforms a new technique was found for performing tensor or Kronecker product calculations. This consisted of converting the tensor product to a Hadamard product by appropriate mapping of the component matrices onto the processor topology. This was the kernel for a new parallel algorithm for performing any discrete transform which can be expressed as a Kronecker or tensor product.

The performance of the T800 transputer which was used as the "building block" of the processor topologies was found to be inadequate, and provided an example of the short lifetime of microprocessor technology. The age of the T800, and the unusually long development time of a faster replacement have meant that the T800 as a stand alone microprocessor, or in a "coarse-grained" parallel configuration is no longer competitive with conventional microprocessors. This means that if a parallel implementation is to provide better performance than conventional microprocessors increasing numbers of processors have to be employed. This is a limited solution as processor communication and economic overheads increase with larger scale machines.

Given the availability of more powerful processors parallel processing can provide a powerful tool for performance enhancement of computationally intensive tasks. The software development performed in the course of this thesis revealed two drawbacks to parallel processing development. These were the programming complexity associated with larger MIMD parallel applications and the complexity of algorithm design.

A possible area of future work would be to investigate the growth of program complexity in MIMD applications and possible alternatives such as employing SIMD techniques or the development of more sophisticated programming tools and graphic user interfaces to aid MIMD parallel program design.

A parallel algorithm was developed for transforms which can be expressed as Kronecker or tensor products. Further work could be performed to investigate the

Parallel processing techniques can bring performance improvements to computationally intensive tasks. This investigation into parallel algorithms and architectures has provided an insight into an area of great potential which is being used at an increasing rate to meet the processing needs of the computing community.

REFERENCES

[1] N. Ahmed and K.R. Rao, Orthogonal Transforms for Digital Signal Processing, *Springer - Verlag, 1975.*

[2] G. Almasi and A. Gottlieb, Highly Parallel Computing, *Benjamin/Cummings Publishing company, 1989.*

[3] H.C. Andrews and K.L. Caspari, A generalised technique for spectral analysis, *IEEE Transactions on Computing, C-19, 16-25, 1970.*

[4a] K.G. Beauchamp, Transforms for Engineers, A Guide to Signal Processing, *Clarendon Press Oxford, 1987.*

[4b] K.G. Beauchamp, Walsh functions and their applications, *Academic Press, 1975.*

[5] R.E. Blahut, Theory and Practice of Error Control Codes, *Addison-Wesley, 1984.*

[6a] R.N. Bracewell, Numerical Transforms, *Science, Vol. 248, 11 May 1990.*

[6b] R.N. Bracewell, The Fast Hartley Transform, *Proceedings of the IEEE, Vol. 72, No. 8, August 1984.*

[7] J.W. Brewer, Kronecker Products and Matrix Calculus in System Theory, *IEEE Transactions on Circuits and Systems, Vol. cas-25, No.9, September 1978.*

[8] E. Oran Brigham, The fast Fourier transform, *Englewood Cliffs, N.J., Prentice-Hall, 1974.*

[9] L. Carrioli, A Pyramidal Haar-Transform implementation, *Proceedings of the Third International Conference on Image Analysis and Processing, Plenum Press, October 1985.*

[10] A.D. Cenzo, Transform Lengths for Correlation and Convolution, *IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. ASSP-35, No. 5, May 1987.*

[11] Chaitali Chakrabarti and Joseph F. Jaja, A Parallel Algorithm for Template Matching on an SIMD mesh connected computer, *IEEE Proceedings 10th International conference on Pattern Recognition Vol.2, 362-367, 1990.*

[12] W.K. Cham, Y.T. Chan, Integer Discrete Cosine Transforms, *1st IASTED International Symposium on Signal Processing and its Applications, 1987.*

[13] N. Chelemal, K.R. Rao, Fast Computational Algorithms for the Discrete Cosine Transform, *9th Annual Asilomar Conference Circuit, Syst., Comput., Pacific Grove, CA, November 1985).*

[14] W. Chen, C. Smith, S. Fralick, A fast computational algorithm for the discrete cosine transform, *IEEE Transactions on Communications, Com-25, No. 9, 1977.*

[15] Chun-Hsien Chou and Yung-Chang Chen, A VLSI architecture for Real-Time and flexible Image template matching, *IEEE Transactions on Circuits and Systems, Vol. 36, No. 10, October 1989.*

[16a] N.I. Cho, S.U. Lee, DCT Algorithms for VLSI Parallel Implementations, *IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. 38, No. 1, January 1990.*

[16b] N.I. Cho, S.U. Lee, A Fast 4x4 DCT Algorithm for the Recursive 2-D DCT, *IEEE Transactions on Signal Processing, Vol. 40, No. 9, September 1990.*

[17] R.F.W. Coates, Modern Communication Systems, *2nd Ed., 1983, The MacMillan Press Ltd.*

[18] A.D. Culhane, M.C. Peckerbar, A Neural Net Approach to Discrete Hartely and Fourier Transforms, *IEEE Transactions on Circuits and Systems Vol. 36, No. 5, May 1989.*

[19] R.W. Daniel, P.M. Sharkey, Transputer control of a Puma 560 robot via the virtual bus, *IEEE Proceedings, Vol 137, Pt D, No. 4, July 1990.*

[20] I. Daubechies, Orthonormal Bases of Compactly Supported Wavelets, *Communications of Pure and Applied Mathematics, Vol. 41, 909-996, 1988.*

[21] A.R. Davies, R. Wilson, Curve and Corner Extraction using the Multiresolution Fourier Transform, *IEE 4th International Conference on Image Processing and its applications, 1992.*

[22] A. Dembo, Signal Reconstruction from Noisy Partial Information of its Transform, *IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. 37, No. 1, January 1989.*

[23] J. Fransaer, D. Fransaer, Fast Cross-Correlation Algorithm with Application to Spectral Analysis, *IEEE Transactions on Siganl Processing, Vol. 39, No. 9, September 1991.*

[24] I. Gertner, A New Efficient Algorithm to Compute the Two-Dimensional Discrete Fourier Transform, *IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. 36, No. 7, July 1988.*

[25] R. Gonzales and P. Wintz, Digital Image Processing, *Addison-Wesley, 1987.*

[26] J. Granata, M. Conner, R. Tolimieri, Recursive Fast Algorithms and the Role of the Tensor Product, *IEEE Transactions on Signal Processing, Vol. 40, No. 12, December 1992.*

[27] L. Guan, R.K. Ward, Restoration of Randomly Blurred Images by the Wiener Filter, *IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. 37, No. 4, April 1989.*

[28a] A. Gupta, K.R. Rao, A Fast Recursive Algorithm for the Discrete Sine Transform, *IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. 38, No. 3, March 1990.*

[28b] A. Gupta, K.R. Rao, An efficient FFT Algorithm based on the Discrete Sine Transform, *IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. 39, No. 2, Feburary 1991.*

[29] P. Goupillard, A. Grossman, J. Morlet, Cycle-octave and related transforms in seismic signal analysis, *Geoexploration, 23, 85-102, 1985.*

[30] G. Grasseau, A. Arneodo, M. Holschneider, Wavelet Transform of Multifractals, *Physical Review Letters, The American Physical Society, Vol. 61, No. 20, 14 Nov. 1988.*

[31] Glenn Healey, Byron Dom, Pattern Classification Algorithms for Real-time image segmentation, *IEEE Proceedings 10th International conference on Pattern Recognition, Vol 2, 545-550, 1990.*

[32] D. Hein, N. Ahmed, On a Real-Time Walsh-Hadamard/Cosine Transform Image Processor, *IEEE Transactions on Electromagnetic Compatibility, Vol. EMC-20, No. 3, August 1978.*

[33] W. Daniel Hillis, The Connection Machine, *Scientific American, Vol. 256 p108-115, June 1987.*

[34] R.A. Horn, The Hadamard product, Matrix Theory and Applications, *Proceedings of the Symposia in Applied Mathematics, American Mathematical Society, Vol. 40.*

[35] H.S. Hou, A Fast Recursive Algorithm for Computing the Discrete Cosine Transform, *IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. ASSP-35, No. 10, October 1987.*

[36] J. Kevorkian, J.D. Cole, Pertubation Methods in Applied Mathematics, *Springer-Verlag, 1989.*

[37] Khanh Ly, Y. Attikiouzel, Contour Tracing of Biomedical Binary Images, *1st IASTED International Symposium on Signal Processing and its Applications, 1987.*

[38] D. Kothe, J. Baumgardner et al, PAGOSA: A Massively-Parallel Multi-Material Hydrodynamic Model for Three-Dimensional High-Speed Flow and High-Rate Material Deformation, *SCS Proceedings of the 1993 Simulation Multiconference on the High Performance Computing Symposium, 1993.*

[39] W. Kou, J.W. Mark, A New-Look at DCT-Type Transforms, *IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. 37, No. 12, December 1989.*

[40] P. Kraniauskas, Transforms in Signals and Systems, *Addison-Wesley, 1992.*

[41] E. Kreyszig, Advanced Engineering Mathematics, *Wiley, 1988.*

[42] S. Lakshmivarahan, S.K. Dhall, Analysis and Design of Parallel Algorithms, *McGraw-Hill, 1990.*

[43] F. Thomson Leighton, Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes, *Morgan Kaufmann, 1992.*

[44] A.S. Lewis, G. Knowles, VLSI architecture for 2-D Daubechies wavelet transform without multipliers, *Electronics Letters, Vol 27, n2, 171-173, January 17 1991.*

[45] T.D. Lookabaugh, M.G. Perkins, Application of the Princen-Bradley Filter to Speech and Image Compression, *IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. 38, No. 11, November 1990.*

[46] A.G. Marshall and F.R. Verdun, Fourier Transforms in NMR, Optical, and Mass Spectrometry, *Elsevier, 1990.*

[47] T. Murata, Petri Nets: Properties, Analysis and Applications, *Proceedings of the IEEE, Vol. 77, No. 4, April 1989.*

[48] D. Nandagopal, J.S. Packer, J. Singh, Power Spectral Modelling of Heart Rate Variability, *1st IASTED International Symposium on Signal Processing and its Applications, 1987.*

[49] K.N. Ngan, K.S. Leong, H. Singh, Adaptive Cosine Transform Coding of Images in Perceptual Domain, *IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. 37 No. 11, November 1989.*

[50] R.K. Otnes and L. Enochson, Applied Time Series Analysis, *John Wiley and Sons, 1978.*

[51] Parallel C User Guide V2.2.2 *3L Ltd, 1991.*

[52] S.C. Pei, J.L. Wu, Split Vector Radix 2-D Fast Fourier Transform, *IEEE Transactions on Circuits and Systems, Vol. CAS-34, No. 8, August 1987.*

[53] J.W. Ponton, R. McKinnel, Nonlinear process simulation and control using transputers, *IEEE Proceedings, Vol 137, Pt D, No. 4, July 1990.*

[54] W.H. Press, Wavelet Transforms, *Harvard-Smithsonian Center for Astrophysics Preprint No. 3184, 1991.*

[55] M.P. Quirk, M.F. Garyantes, H.C. Wilck, M.J. Grimm, A Wide-Band High-Resolution Spectrum Analyzer, *IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. 36, No. 12, December 1988.*

[56] V. Ransom, R. Krishnamurthy, Nuclear Plant System Simulation and Analysis, *SCS Proceedings of the 1993 Simulation Multiconference on the High Performance Computing Symposium, 1993.*

[57a] K.R. Rao and P.Yip, Discrete Cosine Transform Algorithms, Advantages, Applications, *Academic Press, 1990.*

[57b] K.R. Rao,M. Narasimhan,K. Revuluri, Image Data Processing by Hadamard-Haar Transform, *IEEE Computer Transactions, C-24, pg 888-896, 1975.*

[58] S.M. Rezaul Hasan, A New VLSI Architecture for Image Data Rate Discrete Cosine Transform Processor, *1st IASTED International Symposium on Signal Processing and its Applications, 1987.*

[59] M.A. Richards, On Hardware Implementation of the Split-Radix FFT, *IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. 36, No. 10, October 1988.*

[60] J.A. Roese, W.K. Pratt, Interframe Cosine Transform Image Coding, *IEEE Transactions on Communications, COM-25:1329-1338, 1977.*

[61] A. Satt, D. Malah, Design of Uniform DFT Filter Banks Optimised for Subband Coding of Speech, *IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. 37, No. 11, November 1989.*

[62] H. Schutte, S. Frydrychowicz, J. Schroder, Scene Matching with Translation Invariant Transforms, *Proceedings IEEE 5th International Conference of Pattern Recognition, 1980.*

[63] J.J. Shynk, Adaptive IIR Filtering using Parallel-Form Realizations, *IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. 37, No. 4, April 1989.*

[64] P.K. Sinha, Q.H. Hong, Detection of Vertical Lines and Circles in 3D Space using Hough Transform Techniques, *IEE 4th International Conference on Image Processing and its applications, 1992.*

[65] H.F. Silverman, Programming the WFTA for Two-Dimensional Data, *IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. 37, No. 9, September 1989.*

[66] T. Smit, M.R. Smith, S.T. Nichols, Efficient Sinc Function Interpolation Technique for Center Padded Data, *IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. 38, No. 1, January 1989.*

[67] H.V. Sorenson, D.L. Jones, M.T. Heideman and C.S. Burrus, Real-Valued Fast Fourier Transform Algorithms, *IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. ASSP-35, No. 6, June 1987.*

[68] L. Sousa, J. Barrios, A. Costa, M. Piedade, Parallel Image Processing for Transputer Based Systems, *IEE 4th International Conference on Image Processing and its Applications, 1992.*

[69] G. Strang, Wavelets and Dilation equations: A brief introduction, *SIAM Review, Vol. 31, No. 4, 614-627, December 1989.*

[70] M.T. Sun, T.C. Chen, A.M. Gottlieb, VLSI Implementation of a 16x16 Discrete Cosine Transform, *IEEE Transactions on Circuits and Systems, Vol. 36, No. 4, April 1989.*

[71] Thinking Machines Corporation, Connection Machine Model CM-2 Technical Summary, *Thinking Machines Corporation,1991.*

[72] J.O. Thomas, Digital Imagery Processing, *Issues in Digital Image Processing, 247-90, Noordhoff, Amsterdam, 1980.*

[73] F.B. Tuteur, Wavelet transformations in signal detection, *Wavelets Time-Frequency methods and Phase Space, Proceedings of the International conference, Springer-Verlag, December 1987.*

[74] F. Wang, P. Yip, Cepstrum Analysis using Discrete Trigonometric Transforms, *IEEE Transactions on Signal Processing, Vol. 39, No. 2, February 1991.*

[75] H.C. Webber, Image processing and transputers, *Amsterdam IOS Press, 1992.*

[76] P.J. Whitebread, R.E. Bogner, The Use of Pattern Recognition by Observation Correlation in Image Processing, *1st IASTED International Symposium on Signal Processing and its Applications, 1987.*

[77] J.L. Wu, Block Diagonal Structure in Discrete Transforms, *IEEE Proceedings, Vol. 136, Pt. E, No. 4, July 1989.*

APPENDIX A

TRANSPUTER TRANSFORM SOURCE CODE

Two Transputer Parallel Fourier Transform

Configuration File:

```
!
!     Configuration file for two transputer fourier transform
!
!     fourier.cfg

processor host           ! the PC
processor root           ! transputer 0
processor T1             ! transputer 1

wire jumper  host[0] root[0]
wire jumper1 root[2] T1[1]

task afserver ins=1 outs=1
task filter   ins=2 outs=2 data=10k
task fft2p_p1  ins=3 outs=3
task fft2p_p2  ins=1 outs=1

place afserver host        ! afserver runs on PC
place filter   root        ! filter and wt2p_p1 run on root transputer
place fft2p_p1  root
place fft2p_p2  T1

connect ? afserver[0] filter[0]
connect ? filter[0] afserver[0]
connect ? filter[1] fft2p_p1[1]
connect ? fft2p_p1[1] filter[1]
connect ? fft2p_p1[2] fft2p_p2[0]
connect ? fft2p_p2[0] fft2p_p1[2]
```

Processor 0 Program:

```
/*.................... FFT HOST PROCESSOR PROGRAM .......................*/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <math.h>
#include <chan.h>


#define DATASIZE       512
#define K      6.2831853071796/DATASIZE
#define DATASPLIT      DATASIZE/2
#define BUTTERFLIES    log(DATASIZE)/log(2)

struct datastruct {
                        double real;
                        double imaginary;
                        };


void Butterfly1(int,int,struct datastruct *);
void Dist_FFT_1(struct datastruct *);
void BitReversal(int *);



int main(argc,argv,envp,in_ports,ins,out_ports,outs)

CHAN   *in_ports[],*out_ports[];
int    argc,ins,outs;
char   *argv[],*envp[];

{
        int i,index[DATASIZE],*indexptr;
        char *buffer;
        double temp[DATASIZE];
        struct                 datastruct                   complexresult[DATASIZE],
complextemp[DATASIZE],*complexptr;
        FILE *infile, *outfile1,*outfile2;

/*.................. Memory check, File check  and read ..................*/

        buffer = (char *) calloc(DATASIZE,sizeof(struct datastruct));
        if(buffer == NULL)
        {
                printf("Memory allocation failed.\n");
                exit(0);
        }
        if((infile   =   fopen("e:\\rod\\trnsfrms\\pfourier\\results\\ltf18.dat","rb"))   ==
NULL)
        {
                printf("Unable to open file.\n");
                exit(0);
        }
                fread(temp,sizeof(double),DATASIZE,infile);
        fclose(infile);

        for(i=0;i<DATASIZE;i++)
         {
                complextemp[i].imaginary = 0.0;
                complextemp[i].real = temp[i];
         }

/*................ Data distribution to other processors ................*/

        complexptr = DATASPLIT + complextemp;
        indexptr = DATASPLIT + index;

        chan_out_message(sizeof(struct
datastruct)*DATASIZE,complextemp,out_ports[2]);

/*..................... Perform Processor 1 FFT .........................*/

        Dist_FFT_1(complextemp);

/*...................... Perform Bit Reversal ...........................*/

        BitReversal(index);


/*................ Collect data from other processors ..................*/

        chan_in_message(sizeof(struct
datastruct)*(DATASPLIT),complexptr,in_ports[2]);
        chan_in_message(sizeof(int)*(DATASPLIT),indexptr,in_ports[2]);

        for(i=0;i<DATASIZE;i++)
                complexresult[i] = complextemp[index[i]];

/*..................... Write results to file ...........................*/

        if((outfile1  =  fopen("e:\\rod\\trnsfrms\\pfourier\\results\\rfftrslt.18","wb"))  ==
NULL)
        {
                printf("Unable to open file.\n");
                exit(0);
        }
        for(i=0;i<DATASIZE;i++)
                temp[i] = complexresult[i].real;
        fwrite(temp,sizeof(double),DATASIZE,outfile1);
        fclose(outfile1);

        if((outfile2 = fopen("e:\\rod\\trnsfrms\\pfourier\\results\\ifftrslt.18","wb")) ==
NULL)
        {
                printf("Unable to open file.\n");
                exit(0);
        }
        for(i=0;i<DATASIZE;i++)
                temp[i] = complexresult[i].imaginary;
        fwrite(temp,sizeof(double),DATASIZE,outfile2);
        fclose(outfile2);

        return(0);
}



void Dist_FFT_1(struct datastruct *complextemp)
{
        int i,currentsize,j=0,iter=1;
        struct datastruct ctemp[DATASIZE];

        currentsize = DATASPLIT;
        for(i=0;i<DATASIZE;i++)
                ctemp[i] = complextemp[i];

/*.................... Step 1 for Two Processors ..........................*/

        while(j < currentsize)
        {
                complextemp[j].real = ctemp[j].real + ctemp[j+currentsize].real;
                complextemp[j].imaginary        =        ctemp[j].imaginary        +
ctemp[j+currentsize].imaginary;
                j++;
        }
        currentsize = currentsize / 2;

/*........................ FFT Butterfly ................................*/

        while(currentsize >=1)
        {
                iter++;
                Butterfly1(iter,currentsize,complextemp);
                currentsize /= 2;
        }
}

void Butterfly1(int iter,int currentsize,struct datastruct *complextemp)
{
        int i,j;
        struct datastruct temp[DATASIZE];
        double x,y,r1,r2,imag;

        for(i=0;i<DATASIZE;i++)
                temp[i] = complextemp[i];
        i = 0;
        while(i < DATASPLIT)
        {
                j = 0;
                while(j < currentsize)
                {
                        complextemp[i].real = temp[i].real + temp[i+currentsize].real;
                        complextemp[i].imaginary      =      temp[i].imaginary      +
temp[i+currentsize].imaginary;
                        j++;
                        i++;
                }
                j = 0;
                while(j < currentsize)
                {
                        r1 = temp[i-currentsize].real - temp[i].real;
                        r2 = -r1;
                        imag = temp[i-currentsize].imaginary - temp[i].imaginary;
                        x = cos(K*j*iter);
                        y = sin(K*j*iter);
                        complextemp[i].real = (x*r1) + (y*imag);
                        complextemp[i].imaginary = (x*imag) + (y*r2);
                        j++;
                        i++;
                }
        }
}

void BitReversal(int *index)
{
        int count=0,final_pos=0,init_pos=0,x;

        for(x=0;x<DATASPLIT;x++)
        {
                init_pos = x;
                while(count < BUTTERFLIES)
                {
                        final_pos = final_pos << 1;
                        final_pos = ((init_pos & 1) ? 1:0) + final_pos;
                        init_pos = init_pos >> 1;
                        count++;
                }
                index[x] = final_pos;
                final_pos = count = 0;
        }
}
```

Processor 1 Program:

```
/*...................... FFT PROCESSOR 1 PROGRAM ......................*/
#include <stdlib.h>
#include <math.h>
#include <chan.h>

#define DATASIZE      512
#define K      6.2831853071796/DATASIZE
#define DATASPLIT      DATASIZE/2
#define BUTTERFLIES     log(DATASIZE)/log(2)

struct datastruct {
                        double real;
                        double imaginary;
                        };


void Butterfly2(int,int,struct datastruct *);
void Dist_FFT_2(struct datastruct *);
void BitReversal(int *);


void main(argc,argv,envp,in_ports,ins,out_ports,outs)

CHAN    *in_ports[],*out_ports[];
int     argc,ins,outs;
char    *argv[],*envp[];

{
        int i,index[DATASIZE],*indexptr;
        struct datastruct complextemp[DATASIZE],*complexptr;

/*................ Read Data from Host processor .......................*/

        chan_in_message(sizeof(struct
datastruct)*DATASIZE,complextemp,in_ports[0]);
```

```
/*...................... Perform Processor 2 FFT .........................*/

        Dist_FFT_2(complextemp);

/*...................... Perform Bit Reversal ............................*/

        BitReversal(index);

/*................ Send data to Host processor ..........................*/

        indexptr = DATASPLIT + index;
        complexptr = DATASPLIT + complextemp;
        chan_out_message(sizeof(struct
datastruct)*DATASPLIT,complexptr,out_ports[0]);
        chan_out_message(sizeof(int)*DATASPLIT,indexptr,out_ports[0]);

}



void Dist_FFT_2(struct datastruct *complextemp)
{
        int i,currentsize,j=0,iter=1;
        double x,y,r1,r2,imag;
        struct datastruct ctemp[DATASIZE];

        currentsize = DATASPLIT;
        for(i=0;i<DATASIZE;i++)
                ctemp[i] = complextemp[i];

/*.................... Step 2 for Two Processors .........................*/

        while(j < currentsize)
         {
                r1 = ctemp[j].real - ctemp[j+currentsize].real;
                r2 = -r1;
                imag = - ctemp[j+currentsize].imaginary + ctemp[j].imaginary;
                x = cos(K*j*iter);
                y = sin(K*j*iter);
                complextemp[j+currentsize].real = (x*r1) + (y*imag);
                complextemp[j+currentsize].imaginary = (x*imag) + (y*r2);
                j++;
         }
        currentsize = currentsize / 2;

/*........................ FFT Butterfly ...............................*/

        while(currentsize >=1)
         {
                iter++;
                Butterfly2(iter,currentsize,complextemp);
                currentsize = currentsize / 2;
        }
}

void Butterfly2(int iter,int currentsize,struct datastruct *complextemp)
{
        int i,j;
        struct datastruct temp[DATASIZE];
        double x,y,r1,r2,imag;

        for(i=0;i<DATASIZE;i++)
                temp[i] = complextemp[i];
        i = DATASPLIT;
        while(i < DATASIZE)
        {
                j = 0;
                while(j < currentsize)
                {
                        complextemp[i].real = temp[i].real + temp[i+currentsize].real;
                        complextemp[i].imaginary        =       temp[i].imaginary       +
temp[i+currentsize].imaginary;
                        j++;
                        i++;
                }
                j = 0;
                while(j < currentsize)
                {
                        r1 = -temp[i].real + temp[i-currentsize].real;
                        r2 = -r1;
                        imag = temp[i-currentsize].imaginary - temp[i].imaginary;
                        x = cos(K*j*iter);
                        y = sin(K*j*iter);
                        complextemp[i].real = (x*r1) + (y*imag);
                        complextemp[i].imaginary = (x*imag) + (y*r2);
                        j++;
                        i++;
                }
        }
}

void BitReversal(int *index)
{
        int count=0,final_pos=0,init_pos=0,x;

        for(x=DATASPLIT;x<DATASIZE;x++)
        {
                init_pos = x;
                while(count < BUTTERFLIES)
                {
                        final_pos = final_pos << 1;
                        final_pos = ((init_pos & 1) ? 1:0) + final_pos;
                        init_pos = init_pos >> 1;
                        count++;
                }
                index[x] = final_pos;
                final_pos = count = 0;
        }
}
```

Two Transputer Parallel Cosine Transform

Configuration File:

```
!
!     Configuration file for two transputer cosine transform
!
!     cosine.cfg

processor host              ! the PC
processor root              ! transputer 0
processor T1                ! transputer 1

wire jumper  host[0] root[0]
wire jumper1 root[2] T1[1]

task afserver ins=1 outs=1
task filter   ins=2 outs=2 data=10k
task dctfftp1  ins=3 outs=3
task dctfftp2  ins=1 outs=1

place afserver host          ! afserver runs on PC
place filter   root         ! filter and wt2p_p1 run on root transputer
place dctfftp1  root
place dctfftp2  T1

connect ? afserver[0] filter[0]
connect ? filter[0] afserver[0]
connect ? filter[1] dctfftp1[1]
connect ? dctfftp1[1] filter[1]
connect ? dctfftp1[2] dctfftp2[0]
connect ? dctfftp2[0] dctfftp1[2]
```

Processor 0 Program:

```
/*.................... DCT HOST PROCESSOR PROGRAM .......................*/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <math.h>
#include <chan.h>
#include <time.h>


#define DATASIZE      8
#define K      6.2831853071796/DATASIZE
#define DATASPLIT      DATASIZE/2
#define BUTTERFLIES     log(DATASIZE)/log(2)

struct datastruct {
                    double real;
                    double imaginary;
                    };


void Butterfly1(int,int,struct datastruct *);
void Dist_FFT_1(struct datastruct *);
void BitReversal(int *);
void Sort(double *,struct datastruct *);
void Kmult(struct datastruct *,struct datastruct *,int *);



int main(argc,argv,envp,in_ports,ins,out_ports,outs)

CHAN    *in_ports[],*out_ports[];
int     argc,ins,outs;
char    *argv[],*envp[];

{
        int i,index[DATASIZE],*indexptr,tstart,tend;
        char *buffer;
        double temp[DATASIZE];
        struct               datastruct                complexresult[DATASIZE],
complextemp[DATASIZE],*complexptr;
        FILE *infile, *outfile;

/*................. Memory check, File check  and read .................*/

        buffer = (char *) calloc(DATASIZE,sizeof(struct datastruct));
        if(buffer == NULL)
```

```
	{
		printf("Memory allocation failed.\n");
		exit(0);
	}
	if((infile = fopen("b:\\fdata8.dat","rb"))==NULL)
	{
		printf("Unable to open file.\n");
		exit(0);
	}
		fread(temp,sizeof(double),DATASIZE,infile);
	fclose(infile);

/*............................ Sort Input Data ..........................*/

	tstart = timer_now();
	Sort(temp,complextemp);


/*................ Data distribution to other processors ................*/

	complexptr = DATASPLIT + complextemp;
	indexptr = DATASPLIT + index;
	chan_out_message(sizeof(struct
datastruct)*DATASIZE,complextemp,out_ports[2]);

/*...................... Perform Processor 1 FFT .........................*/

	Dist_FFT_1(complextemp);

/*....................... Perform Bit Reversal ...........................*/

	BitReversal(index);

/*....................... Multiply by exponent ...........................*/

	Kmult(complextemp,complexresult,index);


/*................ Collect data from other processors ...................*/

	chan_in_message(sizeof(struct
datastruct)*(DATASPLIT),complexptr,in_ports[2]);
	chan_in_message(sizeof(int)*(DATASPLIT),indexptr,in_ports[2]);

	for(i=0;i<DATASIZE;i++)
		complexresult[i] = complextemp[index[i]];

	tend = timer_now();
	printf("Execution time was %d\n",tend-tstart);
```

```
/*...................... Write results to file ...........................*/

        if((outfile = fopen("b:\\cosine.rlt","w"))==NULL)
        {
                printf("Unable to open file.\n");
                exit(0);
        }
        for(i=0;i<DATASIZE/2;i++)
                fprintf(outfile,"%lf\n",complexresult[i].real);
        for(i=0;i<DATASIZE/2;i++)
                fprintf(outfile,"%lf\n",complexresult[i].imaginary);

        fclose(outfile);

        return(0);
}



void Dist_FFT_1(struct datastruct *complextemp)
{
        int i,currentsize,j=0,iter=1;
        struct datastruct ctemp[DATASIZE];

        currentsize = DATASPLIT;
        for(i=0;i<DATASIZE;i++)
                ctemp[i] = complextemp[i];

/*.................... Step 1 for Two Processors .........................*/

        while(j < currentsize)
        {
                complextemp[j].real = ctemp[j].real + ctemp[j+currentsize].real;
                complextemp[j].imaginary        =        ctemp[j].imaginary        +
ctemp[j+currentsize].imaginary;
                j++;
        }
        currentsize = currentsize / 2;

/*........................ FFT Butterfly ................................*/

        while(currentsize >=1)
        {
                iter++;
                Butterfly1(iter,currentsize,complextemp);
                currentsize /= 2;
        }
}
```

```
void Butterfly1(int iter,int currentsize,struct datastruct *complextemp)
{
        int i,j;
        struct datastruct temp[DATASIZE];
        double x,y,r1,r2,imag;

        for(i=0;i<DATASIZE;i++)
                temp[i] = complextemp[i];
        i = 0;
        while(i < DATASPLIT)
        {
                j = 0;
                while(j < currentsize)
                {
                        complextemp[i].real = temp[i].real + temp[i+currentsize].real;
                        complextemp[i].imaginary      =      temp[i].imaginary      +
temp[i+currentsize].imaginary;
                        j++;
                        i++;
                }
                j = 0;
                while(j < currentsize)
                {
                        r1 = temp[i-currentsize].real - temp[i].real;
                        r2 = -r1;
                        imag = temp[i-currentsize].imaginary - temp[i].imaginary;
                        x = cos(K*j*iter);
                        y = sin(K*j*iter);
                        complextemp[i].real = (x*r1) + (y*imag);
                        complextemp[i].imaginary = (x*imag) + (y*r2);
                        j++;
                        i++;
                }
        }
}

void BitReversal(int *index)
{
        int count=0,final_pos=0,init_pos=0,x;

        for(x=0;x<DATASPLIT;x++)
        {
                init_pos = x;
                while(count < BUTTERFLIES)
                {
                        final_pos = final_pos << 1;
                        final_pos = ((init_pos & 1) ? 1:0) + final_pos;
                        init_pos = init_pos >> 1;
```

```
				count++;
			}
			index[x] = final_pos;
			final_pos = count = 0;
		}
}

void Sort(double *temp,struct datastruct *complextemp)
{
		int i;

		for(i=0;i<DATASIZE/2;i++)
		{
			complextemp[i].real = temp[2*i];
			complextemp[i].imaginary = 0.0;
			complextemp[DATASIZE-1-i].real = temp[2*i+1];
			complextemp[DATASIZE-1-i].imaginary = 0.0;
		}
}



void  Kmult(struct  datastruct  *complextemp,struct  datastruct  *complexresult,int
*index)
{
		int i;
		double c,s;

		for(i=0;i<DATASIZE/4;i++)
		{
			c = cos(i * K * 0.25);
			s = sin(i * K * 0.25);
			complexresult[i].real     =     complextemp[index[i]].real     *     c     +
complextemp[index[i]].imaginary * s;
			complexresult[i].imaginary = -complextemp[index[i]].imaginary * c +
complextemp[index[i]].real * s;
		}
}
```

Processor 1 Program:

```
/*....................... DCT PROCESSOR 1 PROGRAM .......................*/
#include <stdlib.h>
#include <math.h>
#include <chan.h>

#define DATASIZE      8
#define K      6.2831853071796/DATASIZE
#define DATASPLIT      DATASIZE/2
#define BUTTERFLIES    log(DATASIZE)/log(2)

struct datastruct {
                        double real;
                        double imaginary;
                        };


void Butterfly2(int,int,struct datastruct *);
void Dist_FFT_2(struct datastruct *);
void BitReversal(int *);
void Kmult(struct datastruct *,struct datastruct *,int *);


void main(argc,argv,envp,in_ports,ins,out_ports,outs)

CHAN   *in_ports[],*out_ports[];
int    argc,ins,outs;
char   *argv[],*envp[];

{
        int i,index[DATASIZE],*indexptr;
        struct                                                          datastruct
complextemp[DATASIZE],*complexptr,complexresult[DATASIZE];


/*................ Read Data from Host processor ........................*/

        chan_in_message(sizeof(struct
datastruct)*DATASIZE,complextemp,in_ports[0]);

/*..................... Perform Processor 2 FFT .........................*/

        Dist_FFT_2(complextemp);

/*..................... Perform Bit Reversal ............................*/

        BitReversal(index);
```

```
/*...................... Multiply by exponent ............................*/

        Kmult(complextemp,complexresult,index);

/*................ Send data to Host processor .........................*/

        indexptr = DATASPLIT + index;
        complexptr = DATASPLIT + complextemp;
        chan_out_message(sizeof(struct
datastruct)*DATASPLIT,complexptr,out_ports[0]);
        chan_out_message(sizeof(int)*DATASPLIT,indexptr,out_ports[0]);

}



void Dist_FFT_2(struct datastruct *complextemp)
{
        int i,currentsize,j=0,iter=1;
        double x,y,r1,r2,imag;
        struct datastruct ctemp[DATASIZE];

        currentsize = DATASPLIT;
        for(i=0;i<DATASIZE;i++)
                ctemp[i] = complextemp[i];

/*.................... Step 2 for Two Processors .........................*/

        while(j < currentsize)
        {
                r1 = ctemp[j].real - ctemp[j+currentsize].real;
                r2 = -r1;
                imag = - ctemp[j+currentsize].imaginary + ctemp[j].imaginary;
                x = cos(K*j*iter);
                y = sin(K*j*iter);
                complextemp[j+currentsize].real = (x*r1) + (y*imag);
                complextemp[j+currentsize].imaginary = (x*imag) + (y*r2);
                j++;
        }
        currentsize = currentsize / 2;

/*........................ FFT Butterfly ................................*/

        while(currentsize >=1)
        {
                iter++;
                Butterfly2(iter,currentsize,complextemp);
                currentsize = currentsize / 2;
        }
```

```
}

void Butterfly2(int iter,int currentsize,struct datastruct *complextemp)
{
        int i,j;
        struct datastruct temp[DATASIZE];
        double x,y,r1,r2,imag;

        for(i=0;i<DATASIZE;i++)
                temp[i] = complextemp[i];
        i = DATASPLIT;
        while(i < DATASIZE)
        {
                j = 0;
                while(j < currentsize)
                {
                        complextemp[i].real = temp[i].real + temp[i+currentsize].real;
                        complextemp[i].imaginary        =       temp[i].imaginary       +
temp[i+currentsize].imaginary;
                        j++;
                        i++;
                }
                j = 0;
                while(j < currentsize)
                {
                        r1 = -temp[i].real + temp[i-currentsize].real;
                        r2 = -r1;
                        imag = temp[i-currentsize].imaginary - temp[i].imaginary;
                        x = cos(K*j*iter);
                        y = sin(K*j*iter);
                        complextemp[i].real = (x*r1) + (y*imag);
                        complextemp[i].imaginary = (x*imag) + (y*r2);
                        j++;
                        i++;
                }
        }
}

void BitReversal(int *index)
{
        int count=0,final_pos=0,init_pos=0,x;

        for(x=DATASPLIT;x<DATASIZE;x++)
        {
                init_pos = x;
                while(count < BUTTERFLIES)
                {
                        final_pos = final_pos << 1;
                        final_pos = ((init_pos & 1) ? 1:0) + final_pos;
                init_pos = init_pos >> 1;
                count++;
            }
            index[x] = final_pos;
            final_pos = count = 0;
        }
}

void Kmult(struct datastruct *complextemp,struct datastruct *complexresult,int
*index)
{
        int i;
        double c,s;

        for(i=DATASPLIT;i<DATASPLIT+DATASIZE/4;i++)
        {
                c = cos(i * K * 0.25);
                s = sin(i * K * 0.25);
                complexresult[i].real = complextemp[index[i]].real * c +
complextemp[index[i]].imaginary * s;
                complexresult[i].imaginary = -complextemp[index[i]].imaginary * c +
complextemp[index[i]].real * s;
        }
}
```

Two Transputer Parallel Walsh Transform

Configuration File:

```
!
!     Configuration file for two transputer walsh transform
!
!     walsh.cfg

processor host              ! the PC
processor root              ! transputer 0
processor T1                ! transputer 1

wire jumper  host[0] root[0]
wire jumper1 root[2] T1[1]

task afserver ins=1 outs=1
task filter   ins=2 outs=2 data=10k
task wt2p_p1  ins=3 outs=3
task wt2p_p2  ins=1 outs=1

place afserver host          ! afserver runs on PC
place filter   root         ! filter and wt2p_p1 run on root transputer
place wt2p_p1  root
place wt2p_p2  T1

connect ? afserver[0] filter[0]
connect ? filter[0] afserver[0]
connect ? filter[1] wt2p_p1[1]
connect ? wt2p_p1[1] filter[1]
connect ? wt2p_p1[2] wt2p_p2[0]
connect ? wt2p_p2[0] wt2p_p1[2]
```

Processor 0 Program:

```
/*................. WALSH TRANSFORM HOST PROCESSOR PROGRAM ..............*/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <math.h>
#include <chan.h>


#define DATASIZE       2048
#define DATASPLIT      DATASIZE/2
#define BUTTERFLIES    log(DATASIZE)/log(2)

void Butterfly1(int,double *);
void Dist_FWT_1(double *);
void BitReversal(int *);



int main(argc,argv,envp,in_ports,ins,out_ports,outs)

CHAN *in_ports[],*out_ports[];
int argc,ins,outs;
char *argv[],*envp[];

{
        int i,index[DATASIZE],*indexptr;
        char *buffer;
        double walshresult[DATASIZE], walshtemp[DATASIZE],*walshptr;
        FILE *infile, *outfile;


/*.................. Memory check, File check  and read ..................*/

        buffer = (char *) calloc(DATASIZE,sizeof(double));
        if(buffer == NULL)
        {
                printf("Memory allocation failed.\n");
                exit(0);
        }
        if((infile = fopen("a:\\wdat2048.dat","r")) == NULL)
        {
                printf("Unable to open file.\n");
                exit(0);
        }
        fread(walshtemp,sizeof(double),DATASIZE,infile);
        fclose(infile);
```

```
/*................ Data distribution to other processors ................*/

        walshptr = DATASPLIT + walshtemp;
        indexptr = DATASPLIT + index;
        chan_out_message(sizeof(double)*DATASIZE,walshtemp,out_ports[2]);

/*..................... Perform Processor 1 FWT .........................*/

        Dist_FWT_1(walshtemp);

/*...................... Perform Bit Reversal ...........................*/

        BitReversal(index);

/*................ Collect data from other processors ...................*/

        chan_in_message(sizeof(double)*DATASPLIT,walshptr,in_ports[2]);
        chan_in_message(sizeof(int)*DATASPLIT,indexptr,in_ports[2]);

/*................ Combine data from all procrssors ....................*/

        for(i=0;i<DATASIZE;i++)
                walshresult[i] = walshtemp[index[i]];

/*..................... Write results to file ...........................*/

        if((outfile = fopen("a:\\fresults.dat","w")) == NULL)
        {
                printf("Unable to open file.\n");
                exit(0);
        }
        fwrite(walshresult,sizeof(double),DATASIZE,outfile);
        fclose(outfile);
        return(0);
}


void Dist_FWT_1(double *walshtemp)
{
        int i,currentsize,j=0;
        double wtemp[DATASIZE];

        currentsize = DATASPLIT;
        for(i=0;i<DATASIZE;i++)
                wtemp[i] = walshtemp[i];

/*................... Step 1 for Two Processors .........................*/
```

```
        while(j < currentsize)
         {
                walshtemp[j] = wtemp[j] + wtemp[j+currentsize];
                j++;
         }
        currentsize = currentsize / 2;

/*........................ FWT Butterfly ................................*/

        while(currentsize >=1)
         {
                Butterfly1(currentsize,walshtemp);
                currentsize = currentsize / 2;
         }
}

void Butterfly1(int currentsize,double *walshtemp)
{
        int i,j;
        double wtemp[DATASIZE];

        for(i=0;i<DATASIZE;i++)          /*  for(i=0;i<currentsize;i++)  */
                wtemp[i] = walshtemp[i];
        i = 0;
        while(i < (DATASPLIT))
        {
                j = 0;
                while(j < currentsize)
                {
                        walshtemp[i] = wtemp[i] + wtemp[i+currentsize];
                        j++;
                        i++;
                }
                j = 0;
                while(j < currentsize)
                {
                        walshtemp[i] = wtemp[i-currentsize] - wtemp[i];
                        j++;
                        i++;
                }
        }
}

void BitReversal(int *index)
{
        int count=0,final_pos=0,init_pos=0,x;

        for(x=0;x<DATASPLIT;x++)
        {
                init_pos = x;
                while(count < BUTTERFLIES)
                {
                        final_pos = final_pos << 1;
                        final_pos = ((init_pos & 1) ? 1:0) + final_pos;
                        init_pos = init_pos >> 1;
                        count++;
                }
                index[x] = final_pos;
                final_pos = count = 0;
        }
}
```

Processor 1 Program:

```
/*.................. WALSH TRANSFORM PROCESSOR 1 PROGRAM .................*/

#include <stdlib.h>
#include <math.h>
#include <chan.h>

#define DATASIZE        2048
#define DATASPLIT       DATASIZE/2
#define BUTTERFLIES     log(DATASIZE)/log(2)


void Butterfly2(int,double *);
void Dist_FWT_2(double *);
void BitReversal(int *);


void main(argc,argv,envp,in_ports,ins,out_ports,outs)
CHAN *in_ports[],*out_ports[];
int argc,ins,outs;
char *argv[],*envp[];

{
        int i,index[DATASIZE],*indexptr;
        double walshtemp[DATASIZE],*walshptr;

/*................ Read Data from Host processor ........................*/

        chan_in_message(sizeof(double)*DATASIZE,walshtemp,in_ports[0]);

/*...................... Perform Processor 2 FWT .........................*/

        Dist_FWT_2(walshtemp);
        BitReversal(index);

/*................ Send data to Host processor .........................*/
        indexptr = DATASPLIT + index;
        walshptr = DATASPLIT + walshtemp;
        chan_out_message(sizeof(double)*DATASPLIT,walshptr,out_ports[0]);
        chan_out_message(sizeof(int)*DATASPLIT,indexptr,out_ports[0]);
}



void Dist_FWT_2(double *walshtemp)
{
        int i,currentsize,j=0;
        double wtemp[DATASIZE];

        currentsize = DATASPLIT;
        for(i=0;i<DATASIZE;i++)
                wtemp[i] = walshtemp[i];

/*.................... Step 2 for Two Processors .........................*/

        while(j < currentsize)
         {
                walshtemp[j+currentsize] = wtemp[j] - wtemp[j+currentsize];
                j++;
         }
        currentsize = currentsize / 2;

/*........................ FWT Butterfly ...............................*/

        while(currentsize >=1)
         {
                Butterfly2(currentsize,walshtemp);
                currentsize = currentsize / 2;
         }
}

void Butterfly2(int currentsize,double *walshtemp)
{
        int i,j;
        double wtemp[DATASIZE];

        for(i=0;i<DATASIZE;i++)
                wtemp[i] = walshtemp[i];
        i = DATASPLIT;
        while(i < (DATASIZE))
        {
                j = 0;
```

```
		while(j < currentsize)
		{
			walshtemp[i] = wtemp[i] + wtemp[i+currentsize];
			j++;
			i++;
		}
		j = 0;
		while(j < currentsize)
		{
			walshtemp[i] = wtemp[i-currentsize] - wtemp[i];
			j++;
			i++;
		}
	}
}

void BitReversal(int *index)
{
	int count=0,final_pos=0,init_pos=0,x;

	for(x=DATASPLIT;x<DATASIZE;x++)
	{
		init_pos = x;
		while(count < BUTTERFLIES)
		{
			final_pos = final_pos << 1;
			final_pos = ((init_pos & 1) ? 1:0) + final_pos;
			init_pos = init_pos >> 1;
			count++;
		}
		index[x] = final_pos;
		final_pos = count = 0;
	}
}
```

Two Transputer Parallel Haar Transform

Configuration File:

```
!
!       Configuration file for two transputer haar transform
!
!       haar.cfg

processor host          ! the PC
processor root          ! transputer 0
processor T1            ! transputer 1

wire jumper  host[0] root[0]
wire jumper1 root[2] T1[1]

task afserver ins=1 outs=1
task filter   ins=2 outs=2 data=10k
task phaarp1  ins=3 outs=3
task phaarp2  ins=1 outs=1

place afserver host      ! afserver runs on PC
place filter   root     ! filter and phaarp1 run on root transputer
place phaarp1  root
place phaarp2  T1

connect ? afserver[0] filter[0]
connect ? filter[0] afserver[0]
connect ? filter[1] phaarp1[1]
connect ? phaarp1[1] filter[1]
connect ? phaarp1[2] phaarp2[0]
connect ? phaarp2[0] phaarp1[2]
```

Processor 0 Program:

```
/*            TWO   PROCESSOR   HOST   TRANSPUTER   PARALLEL   HAAR
TRANSFORM         */


#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <chan.h>


#define DATASIZE        16
#define DATASPLIT       DATASIZE/2
#define RMAX            log(DATASIZE)/log(2)


void Dist_HT_1(double *);
void Multipliers(double *);


int main(argc,argv,envp,in_ports,ins,out_ports,outs)

CHAN    *in_ports[],*out_ports[];
char    *argv[],*envp[];
int     argc,ins,outs;

{
        int i,j,k=1;
        double Haarresult1[DATASIZE],Haarresult2[DATASPLIT];
        double *Hptr;
        double Haarfinal[DATASIZE];
        char *buffer;
        FILE *infile,*outfile;

                /*      Memory check, File check and read               */

        buffer = (char *)calloc(DATASIZE,sizeof(double));

        if(buffer==NULL)
        {
                printf("Memory allocation failed.\n");
                exit(0);
        }

        if((infile = fopen("a:\\Hdata16.dat","rb"))==NULL)
        {
                printf("Unable to open data file.\n");
        exit(0);
}

fread(Haarresult1,sizeof(double),DATASIZE,infile);
fclose(infile);

                /*      Data distribution       */

Hptr = Haarresult1 + DATASPLIT;
chan_out_message(sizeof(double)*DATASPLIT,Hptr,out_ports[2]);

        /*      Perform processor 1 Haar Transform      */

Dist_HT_1(Haarresult1);

Multipliers(Haarresult1);


        /*      Collect data from other processors      */


chan_in_message(sizeof(double)*DATASPLIT,Haarresult2,in_ports[2]);

Haarfinal[0] = Haarresult1[0] + Haarresult2[0];
Haarfinal[1] = Haarresult1[0] - Haarresult2[0];

for(i=2;i<=DATASPLIT;i+=i)
{
        for(j=i/2;j<i;j++)
        {
                k++;
                Haarfinal[k] = Haarresult1[j];
        }
        for(j=i/2;j<i;j++)
        {
                k++;
                Haarfinal[k] = Haarresult2[j];
        }
}




        /*      Write results to file   */

if((outfile = fopen("a:\\Hrslt16.dat","wb")) == NULL)
{
			printf("Unable to open result file.\n");
			exit(0);
		}
		fwrite(Haarfinal,sizeof(double),DATASIZE,outfile);
		fclose(outfile);
		return(0);
}


void Dist_HT_1(double *Haarresult1)
{
		int i,j,currentsize;
		double Htemp[DATASPLIT];

		for(i=0;i<DATASPLIT;i++)
			Htemp[i] = Haarresult1[i];

		currentsize = DATASPLIT/2;

		while(currentsize >= 1)
		{
			i = 0;
			for(j=0;j<currentsize;j++)
			{
				Haarresult1[j] = Htemp[i] + Htemp[i+1];
				Haarresult1[j+currentsize] = Htemp[i] - Htemp[i+1];
				i+=2;
			}
			for(i=0;i<DATASPLIT;i++)
				Htemp[i] = Haarresult1[i];
			currentsize/=2;
		}

}


void Multipliers(double *Haarresult1)
{
		int m,i=1;
		double Haarmult[DATASPLIT],r;


		for(r=1;r<RMAX;r++)
		{
			for(m=1;m<=(pow(2,r)/2);m++)
			{
				Haarmult[i] = pow(2,r/2);
				i++;
			}
```

```
        }
        for(i=1;i<DATASPLIT;i++)
                Haarresult1[i] = Haarmult[i] * Haarresult1[i];
}
```

Processor 1 Program:

```
/*      TWO PROCESSOR TRANSPUTER 1 PARALLEL HAAR TRANSFORM
*/


#include <stdlib.h>
#include <math.h>
#include <chan.h>


#define DATASIZE        16
#define DATASPLIT       DATASIZE/2
#define RMAX            log(DATASIZE)/log(2)


void Dist_HT_2(double *);
void Multipliers(double *);


int main(argc,argv,envp,in_ports,ins,out_ports,outs)

CHAN    *in_ports[],*out_ports[];
char    *argv[],*envp[];
int     argc,ins,outs;

{
        int i;
        double Haarresult[DATASPLIT];
        char *buffer;


                        /*      Collect Data            */

        chan_in_message(sizeof(double)*DATASPLIT,Haarresult,in_ports[0]);

                /*      Perform processor 2 Haar Transform      */

        Dist_HT_2(Haarresult);

        Multipliers(Haarresult);
```

```
		/*	Distribute data		*/

	chan_out_message(sizeof(double)*DATASPLIT,Haarresult,out_ports[0]);
	return(0);
}


void Dist_HT_2(double *Haarresult)
{
	int i,j,currentsize;
	double Htemp[DATASPLIT];

	for(i=0;i<DATASPLIT;i++)
		Htemp[i] = Haarresult[i];

	currentsize = DATASPLIT/2;

	while(currentsize >= 1)
	{
		i = 0;
		for(j=0;j<currentsize;j++)
		{
			Haarresult[j] = Htemp[i] + Htemp[i+1];
			Haarresult[j+currentsize] = Htemp[i] - Htemp[i+1];
			i+=2;
		}
		for(i=0;i<DATASPLIT;i++)
			Htemp[i] = Haarresult[i];
		currentsize/=2;
	}

}


void Multipliers(double *Haarresult)
{
	int m,i=1;
	double Haarmult[DATASPLIT],r;

	for(r=1;r<RMAX;r++)
		for(m=1;m<=(pow(2,r)/2);m++)
		{
			Haarmult[i] = pow(2,r/2);
			i++;
		}

	for(i=1;i<DATASPLIT;i++)
		Haarresult[i] = Haarmult[i] * Haarresult[i];
}
```

Two Transputer Parallel D4 Wavelet Transform

Configuration File:

```
!
!     Configuration file for two transputer D4 Wavelet transform
!
!     pwav.cfg

processor host              ! the PC
processor root              ! transputer 0
processor T1                ! transputer 1

wire jumper  host[0] root[0]
wire jumper1 root[2] T1[1]

task afserver ins=1 outs=1
task filter   ins=2 outs=2 data=10k
task pwavp0  ins=3 outs=3
task pwavp1  ins=1 outs=1

place afserver host          ! afserver runs on PC
place filter   root         ! filter and pwavp0 run on root transputer
place pwavp0  root
place pwavp1  T1

connect ? afserver[0] filter[0]
connect ? filter[0] afserver[0]
connect ? filter[1] pwavp0[1]
connect ? pwavp0[1] filter[1]
connect ? pwavp0[2] pwavp1[0]
connect ? pwavp1[0] pwavp0[2]
```

Processor 0 Program:

```
/*...............     D4 Wavelet Transform Host Processor Program     ................*/




#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <chan.h>


#define DATASIZE        8
#define DATASPLIT       DATASIZE/2
#define C0              (1+sqrt(3))/(4*sqrt(2))
#define C1              (3+sqrt(3))/(4*sqrt(2))
#define C2              (3-sqrt(3))/(4*sqrt(2))
#define C3              (1-sqrt(3))/(4*sqrt(2))



int main(argc,argv,envp,in_ports,ins,out_ports,outs)

CHAN   *in_ports[],*out_ports[];
int    argc,ins,outs;
char   *argv[],*envp[];

{
        int i,n,currentsize;
        double temp[DATASPLIT],result[DATASIZE],x[DATASIZE];
        char *buffer;
        FILE *infile,*outfile;

                /*    Memory check, File check and read            */

        buffer = (char *)calloc(DATASIZE,sizeof(double));

        if(buffer==NULL)
        {
                printf("Memory allocation failed.\n");
                exit(0);
        }

        if((infile = fopen("c:\\trnsfrms\\data\\fdata.dat","rb"))==NULL)
        {
                printf("Unable to open data file.\n");
                exit(0);
        }
```

```
	fread(x,sizeof(double),DATASIZE,infile);
	fclose(infile);

		/*	Perform transform			*/

	chan_out_message(sizeof(double)*DATASIZE,x,out_ports[2]);

	currentsize = DATASIZE;

	while(currentsize >= 4)
	{
		for(n=0;n<currentsize/2;n++)
		{
			if(n != (currentsize/2)-1)
				temp[n]  =  (C0*x[2*n])  +  (C1*x[(2*n)+1])  +
(C2*x[(2*n)+2]) + (C3*x[(2*n)+3]);
			else
				temp[n] = (C2*x[0]) + (C3*x[1]) + (C0*x[currentsize-
2]) + (C1*x[currentsize-1]);
		}

		currentsize /= 2;

		chan_out_message(sizeof(double)*currentsize,temp,out_ports[2]);

		for(n=0;n<currentsize;n++)
			x[n] = temp[n];

	}


	chan_in_message(sizeof(double)*DATASIZE,result,in_ports[2]);



		/*	Write results to file	*/

	if((outfile = fopen("c:\\trnsfrms\\pwavelet\\results\\frslt.dat","wb")) == NULL)
	{
		printf("Unable to open result file.\n");
		exit(0);
	}
	fwrite(result,sizeof(double),DATASIZE,outfile);
	fclose(outfile);
	return(0);
}
```

Processor 1 Program:

```
/*.....................        D4    Wavelet    Transform    Processor    1    Program
............................*/


#include <stdlib.h>
#include <math.h>
#include <chan.h>

#define         DATASIZE    8
#define DATASPLIT DATASIZE/2
#define C0              (1+sqrt(3))/(4*sqrt(2))
#define C1          (3+sqrt(3))/(4*sqrt(2))
#define C2          (3-sqrt(3))/(4*sqrt(2))
#define C3          (1-sqrt(3))/(4*sqrt(2))


void main(argc,argv,envp,in_ports,ins,out_ports,outs)

CHAN *in_ports[],*out_ports[];
int     argc,ins,outs;
char    *argv[],*envp[];

{
        int i,n,currentsize;
        double temp[DATASPLIT],result[DATASIZE],x[DATASIZE];


        chan_in_message(sizeof(double)*DATASIZE,x,in_ports[0]);

        currentsize = DATASIZE;

        while(currentsize >= 4)
        {
                for(n=0;n<currentsize/2;n++)
                {
                        if(n != (currentsize/2)-1)
                                temp[n]   =   (C3*x[2*n])   -   (C2*x[(2*n)+1])   +
(C1*x[(2*n)+2]) - (C0*x[(2*n)+3]);
                        else
                                temp[n] = (C1*x[0]) - (C0*x[1]) + (C3*x[currentsize-
2]) - (C2*x[currentsize-1]);
                }

                currentsize /= 2;

                for(n=0;n<currentsize;n++)
                        result[n+currentsize] = temp[n];
        chan_in_message(sizeof(double)*currentsize,x,in_ports[0]);
    }

    for(n=0;n<2;n++)
        result[n] = x[n];

    chan_out_message(sizeof(double)*DATASIZE,result,out_ports[0]);
}
```

APPENDIX B

TABLES OF EXPERIMENTAL RESULTS

Note: Any data omitted from execution time tables has been excluded on the grounds of either being superflous to requirements or being unable to be measured with the software compilers available.

1. Processor farm implementation of the Walsh-Hadamard matrix form of the Walsh transform.

Dataset size double precision floating point numbers	Execution time	
	64uS Clock ticks	mS
8	33	2.112
16	75	4.8
32	207	13.25
64	673	43.07
128	2583	165.31

2. Serial implementations of the fast Walsh transform.

Processor: 80386SX

Dataset size double precision floating point numbers	Execution time mS
8	4.779
16	11.986
32	28.728
64	67.267
128	153.172
256	344.897
512	767.104
1024	1688.445
2048	3686.403

Processor: 80386

Dataset size double precision floating point numbers	Execution time mS
8	2.032
16	5.112
32	12.285
64	28.677
128	65.663
256	147.983
512	329.781
1024	726.636
2048	1587.159

Processor: 80486

Dataset size double precision floating point numbers	Execution time mS
8	0.846
16	2.114
32	5.071
64	11.826
128	27.067
256	61.032
512	135.944
1024	300.744
2048	656.776

One transputer

Dataset size double precision floating point numbers	Execution time 64uS Clock ticks	 mS
8	75	4.8
16	186	11.904
32	488	31.232
64	1043	66.752
128	2394	153.216
256	5381	344.384
512	11961	765.504
1024	26396	1687.616
2048	57635	3688.640

3. Two transputer hypercube implementation of the fast Walsh transform.

Dataset size double precision floating point numbers	Execution time 64uS Clock ticks	mS
8	40	2.56
16	98	6.72
32	233	14.91
64	543	34.75
128	1244	79.62
256	2794	178.82
512	6194	396.42
1024	13623	871.87
2048	29744	1903.62

4. Two transputer fast Walsh transform computing resource demands.

Dataset size double precision floating point numbers	Execution time 64 uS clock ticks communications time calc.time	butterfly calc. time	bit reversal
128	16	104	1120
256	32	242	2517
512	63	533	5603

5. Two transputer fast Fourier transform.

Dataset size double precision floating point numbers	Execution time 64uS Clock ticks	mS
8	49	3.136
16	128	8.192
32	313	20.032
64	745	47.680
128	1728	110.592
256	3935	251.840
512	8831	565.184
1024	19560	1251.840
2048	42948	2748.672

6. Two transputer fast Haar transform.

Dataset size double precision floating point numbers	Execution time 64uS Clock ticks	mS
8	31	1.984
16	77	4.928
32	164	10.496
64	331	21.184
128	659	42.176
256	1312	83.968
512	2631	168.384
1024	5277	337.728
2048	10576	676.864

7. Two transputer D4 wavelet transform.

Dataset size double precision floating point numbers	Execution time 64uS Clock ticks	mS
8	21	1.344
16	48	3.072
32	103	6.592
64	214	13.696
128	432	27.648
256	871	55.744
512	1744	111.616

8. Serial D4 Wavelet transform 80286

Dataset size double precision floating point numbers	Execution time 64uS Clock ticks	mS
8		54.945
16		27.4725
32		49.4505
64		104.3956
128		208.7912
256		423.0769

9. Serial D4 Wavelet transform 80386SX-16Mhz

Dataset size double precision floating point numbers	Execution time 64uS Clock ticks	mS
8		-
16		-
32		-
64		54.945
128		109.890
256		219.780

10. Serial D4 Wavelet transform 80486-33Mhz

Dataset size double precision floating point numbers	Execution time 64uS Clock ticks	mS
8		-
16		-
32		-
64		-
128		-
256		-
512		54.945

11. Two transputer D4 Wavelet transform computing resource demands.

Dataset size double precision floating point numbers	Execution time 64 uS clock ticks comms time	calc. time
256	36	835
512	73	1675

12. Serial implementation of the Haar Transform

80286

Dataset size Double precision floating point numbers	Execution time mS
8	54.945
16	109.890
32	164.835
64	329.670

80386SX-16 Mhz

Dataset size Double precision floating point numbers	Execution time mS
128	54.945
256	109.890
512	164.835

80486-33 Mhz

Dataset size Double precision floating point numbers	Execution time mS
512	54.945

13. Serial Implementation of the Cosine Transform via the FFT.

80486-33 Mhz

Dataset size Double precision floating point numbers	Execution time mS
256	60
512	110
1024	280

80386-33 Mhz

Dataset size Double precision floating point numbers	Execution time mS
64	50
128	110
256	170
512	330

80386SX

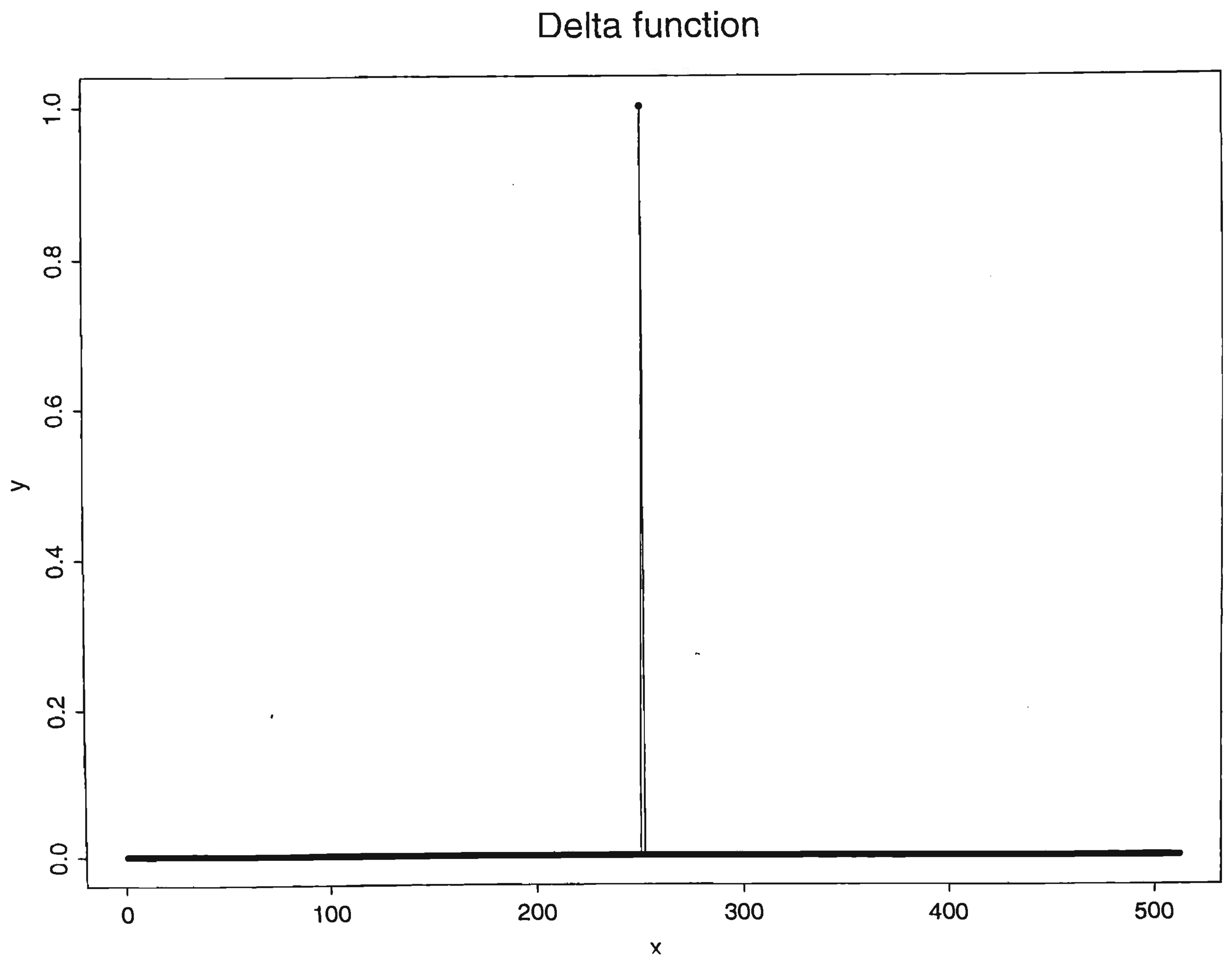
Dataset size Double precision floating point numbers	Execution time mS
32	60
64	110
128	270
256	610
512	1270

14. Two transputer cosine transform.

Dataset size double precision floating point numbers	Execution time	
	64uS Clock ticks	mS
8	53	3.4
16	136	8.7
32	332	21.2
64	781	50
128	1801	115
256	4083	261
512	9104	583
1024	20173	1290

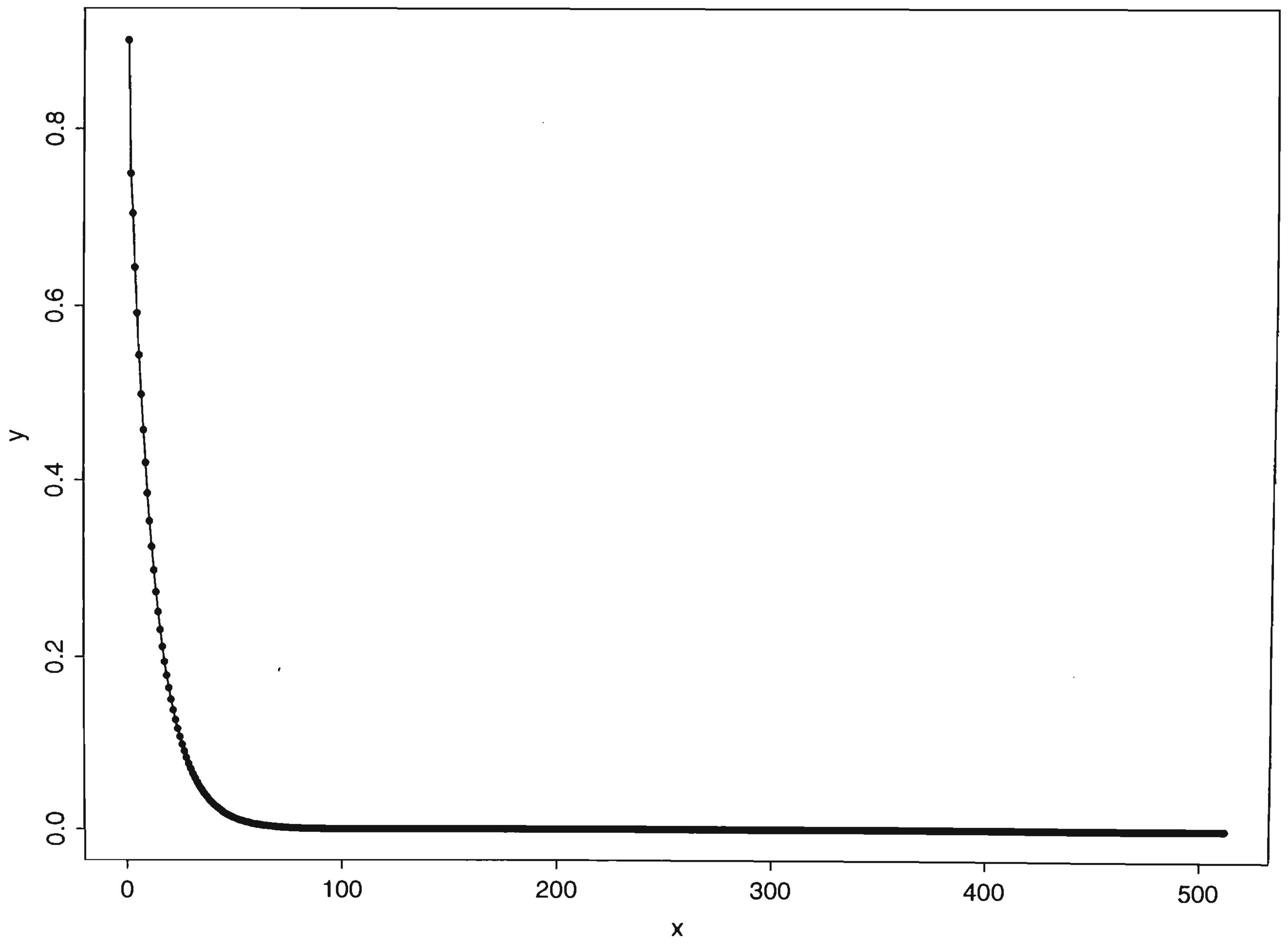
APPENDIX C

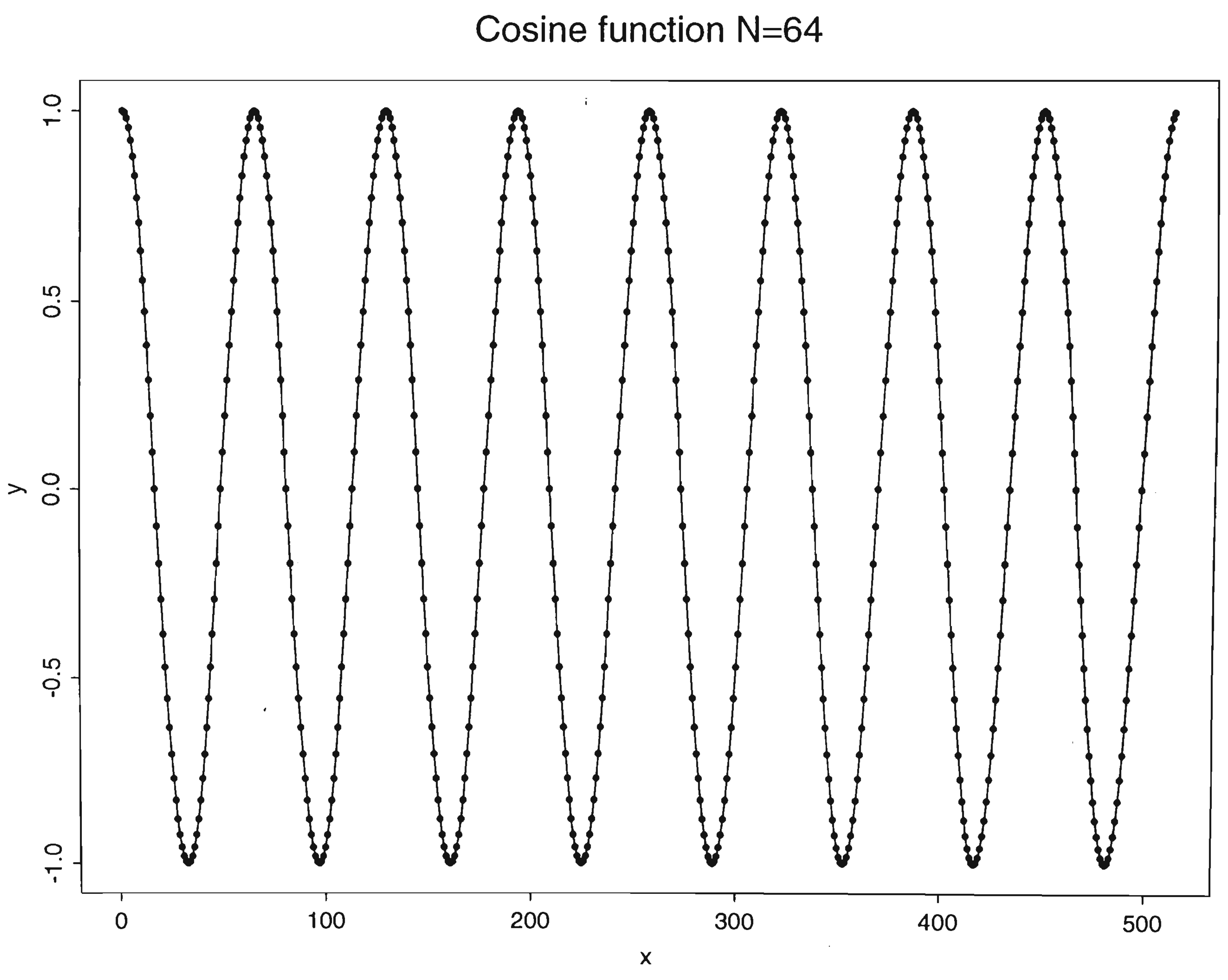
TEST FUNCTIONS AND TRANSFORMS

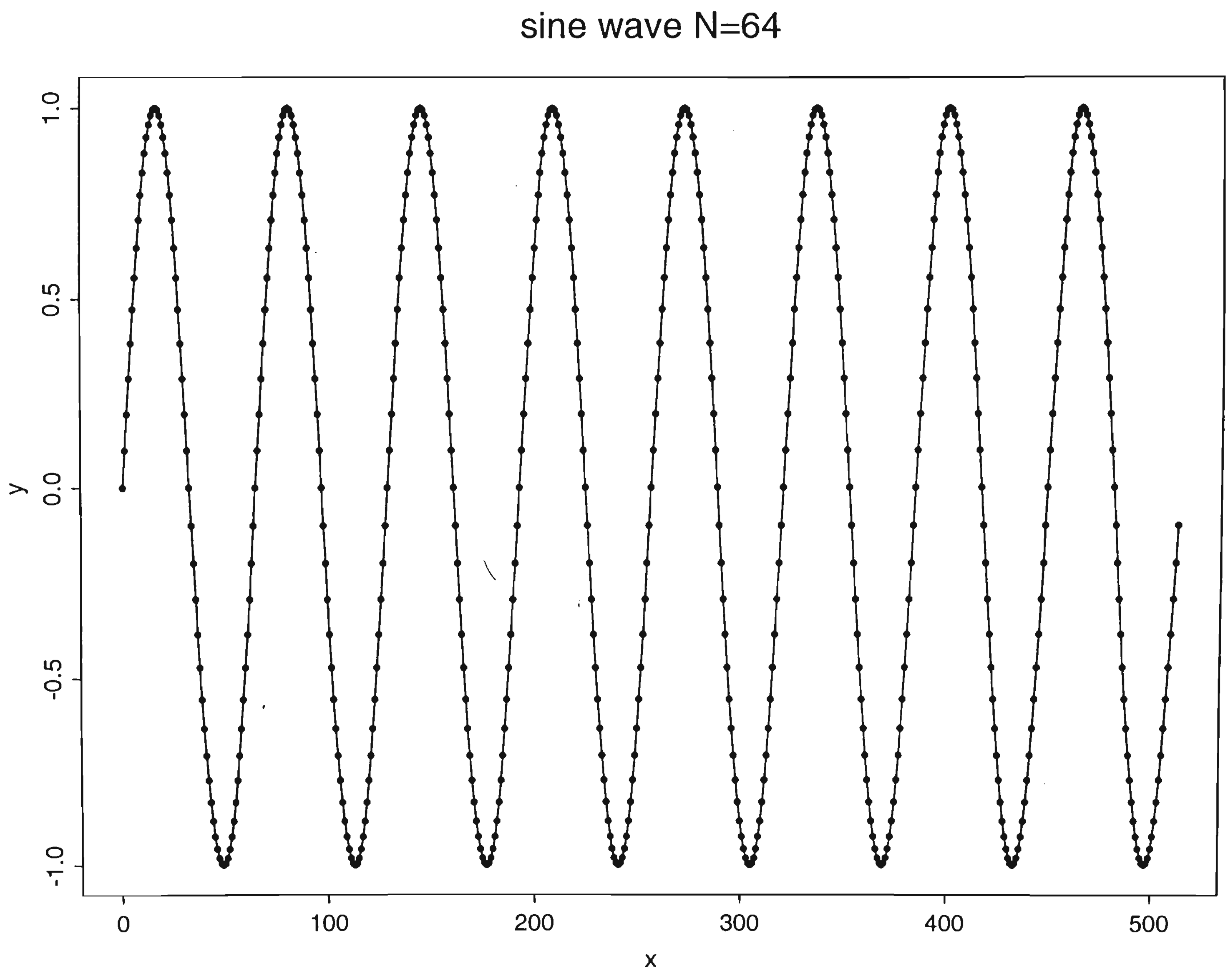


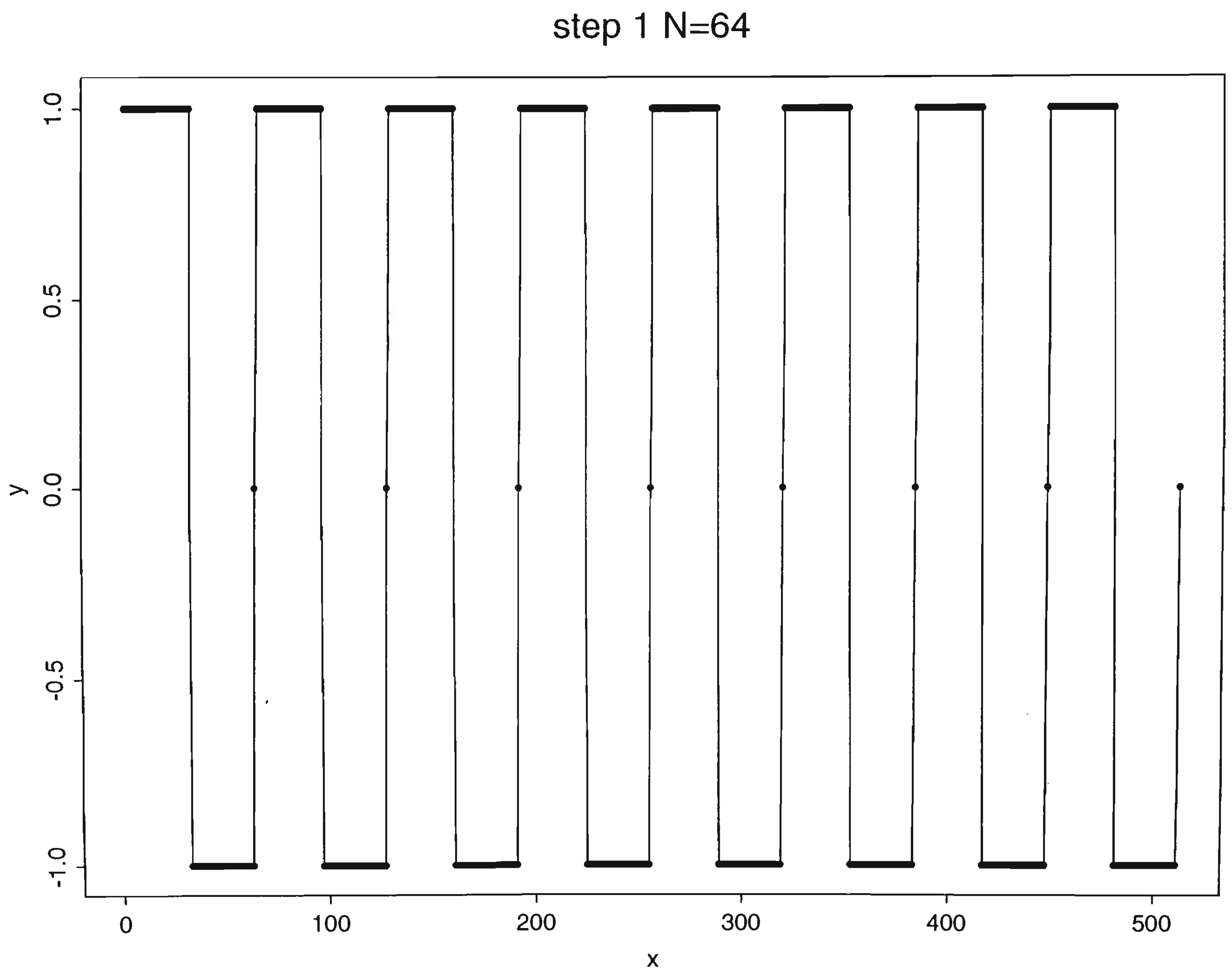
Delta function
y
1.0
0.8
0.6
0.4
0.2
0.0
x
0
100
200
300
400
500

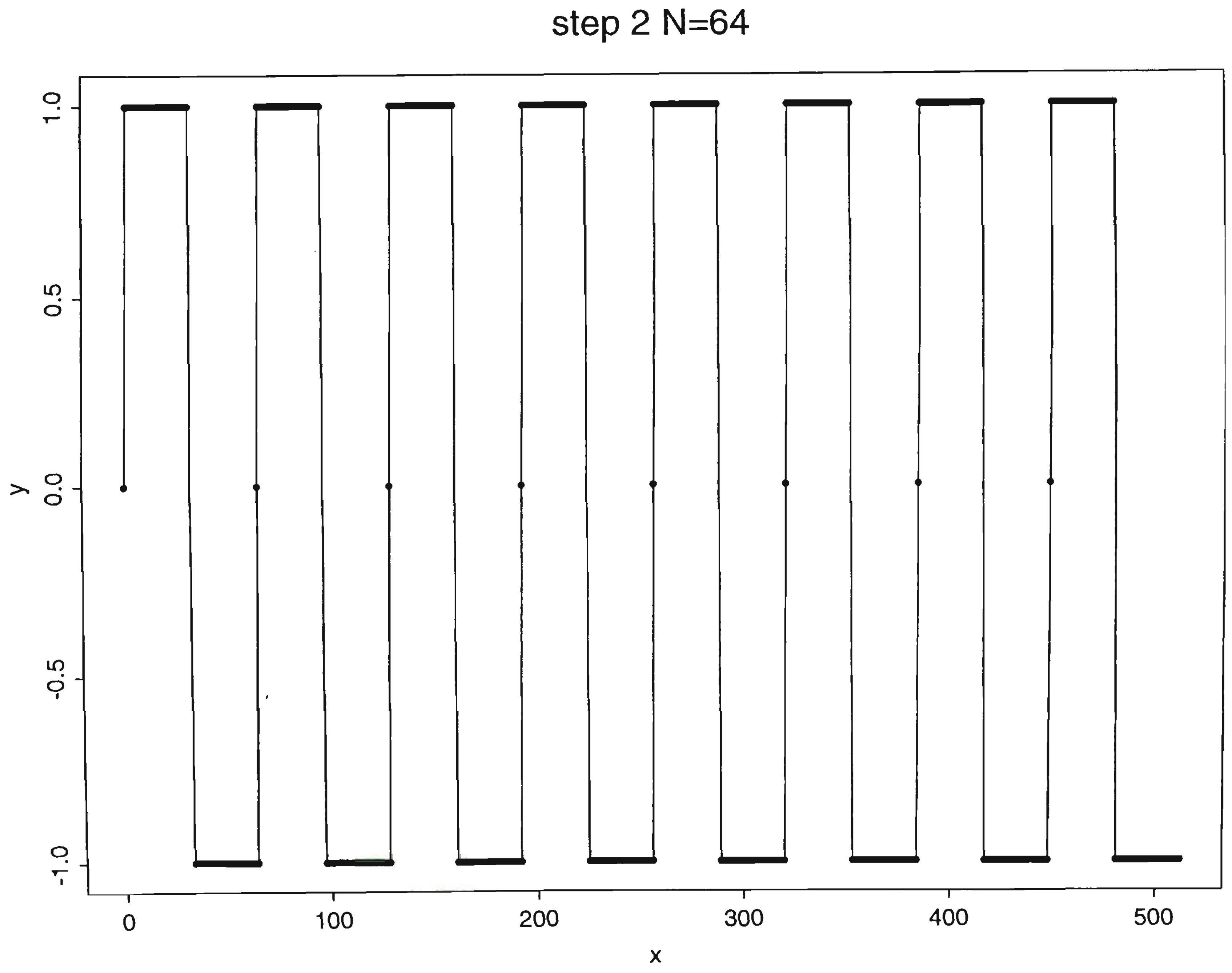
Iterative function x(n+1)=ax(n)+bx(n-1), x(0)=0.9,x(1)=0.75,a=0.7,b=0.2

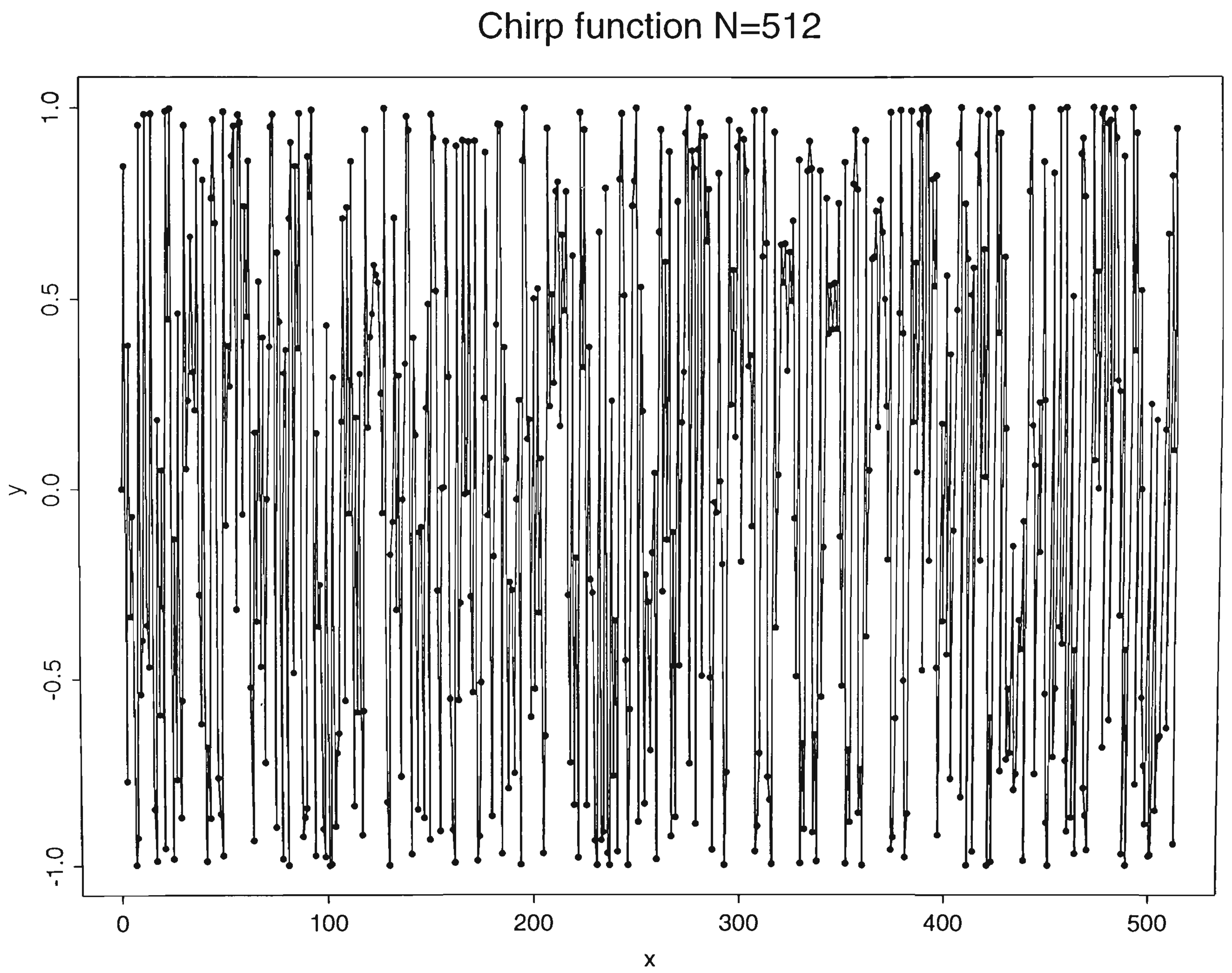


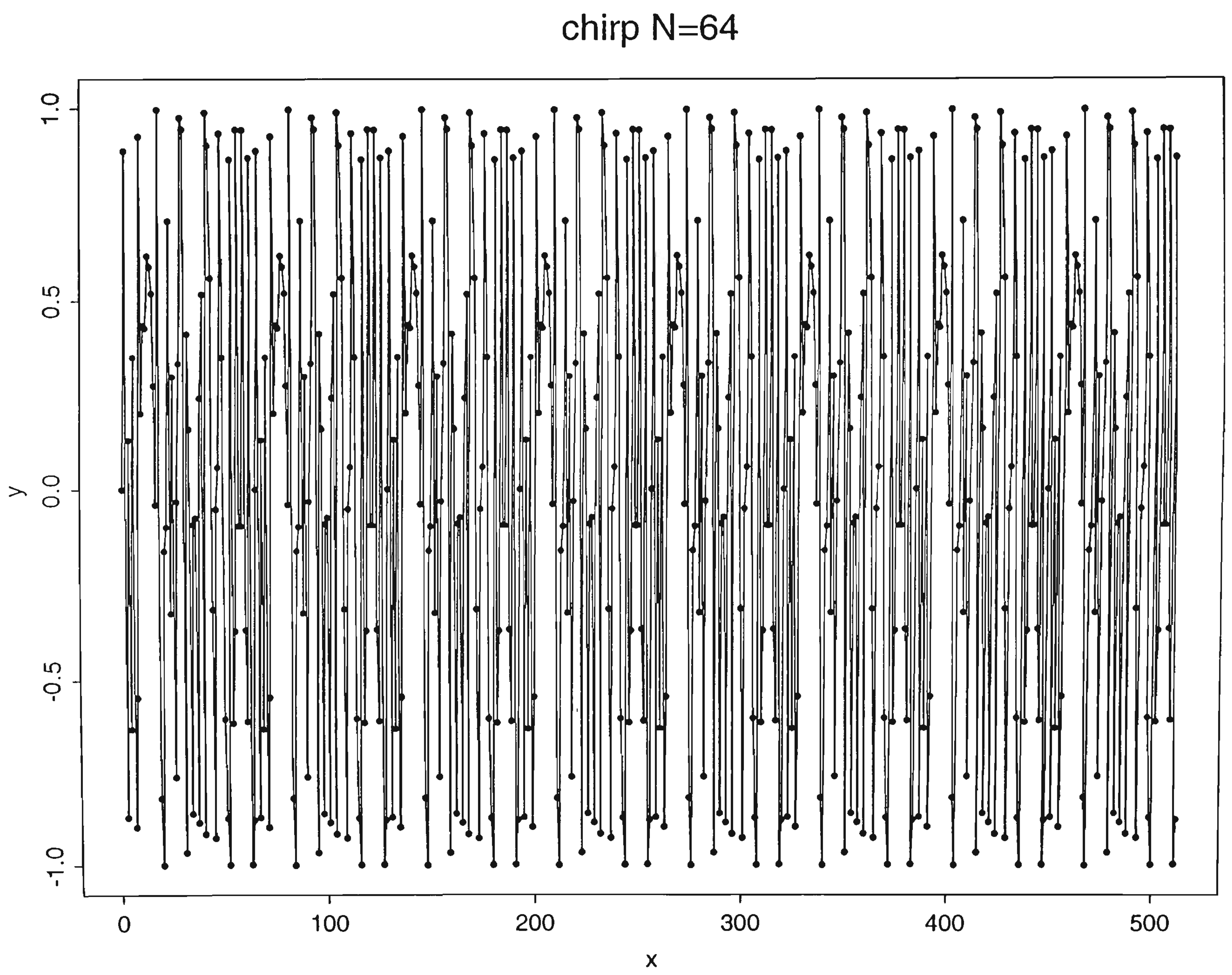

Cosine function N=64
1.0
0.5
0.0
-0.5
-1.0
y
0
100
200
300
400
500
x

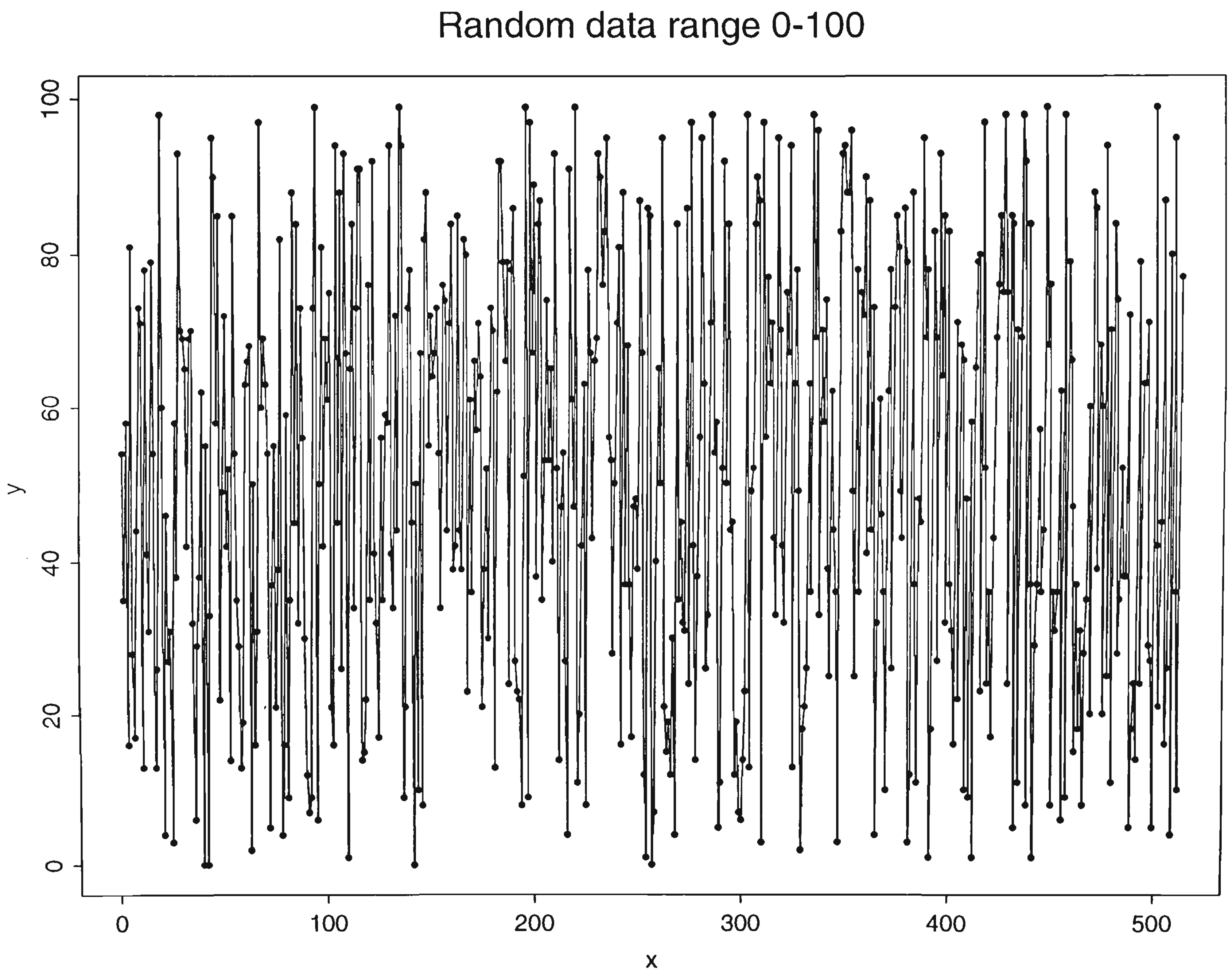

sine wave N=64
y
1.0
0.5
0.0
-0.5
-1.0
0
100
200
300
400
500
x

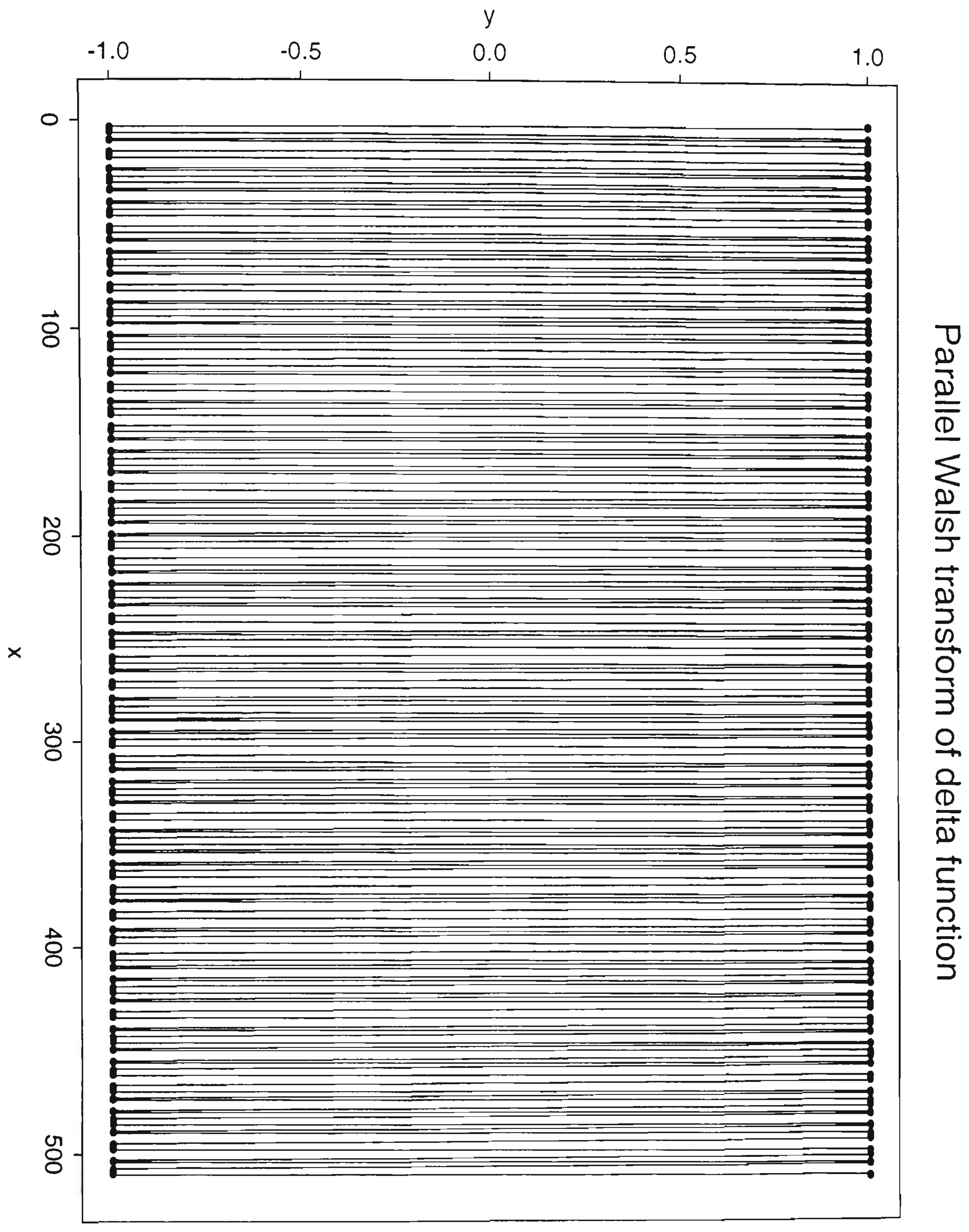

step 1 N=64
1.0
0.5
0.0
-0.5
-1.0
y
0
100
200
300
400
500
x


step 2 N=64
y
1.0
0.5
0.0
-0.5
-1.0
x
0
100
200
300
400
500

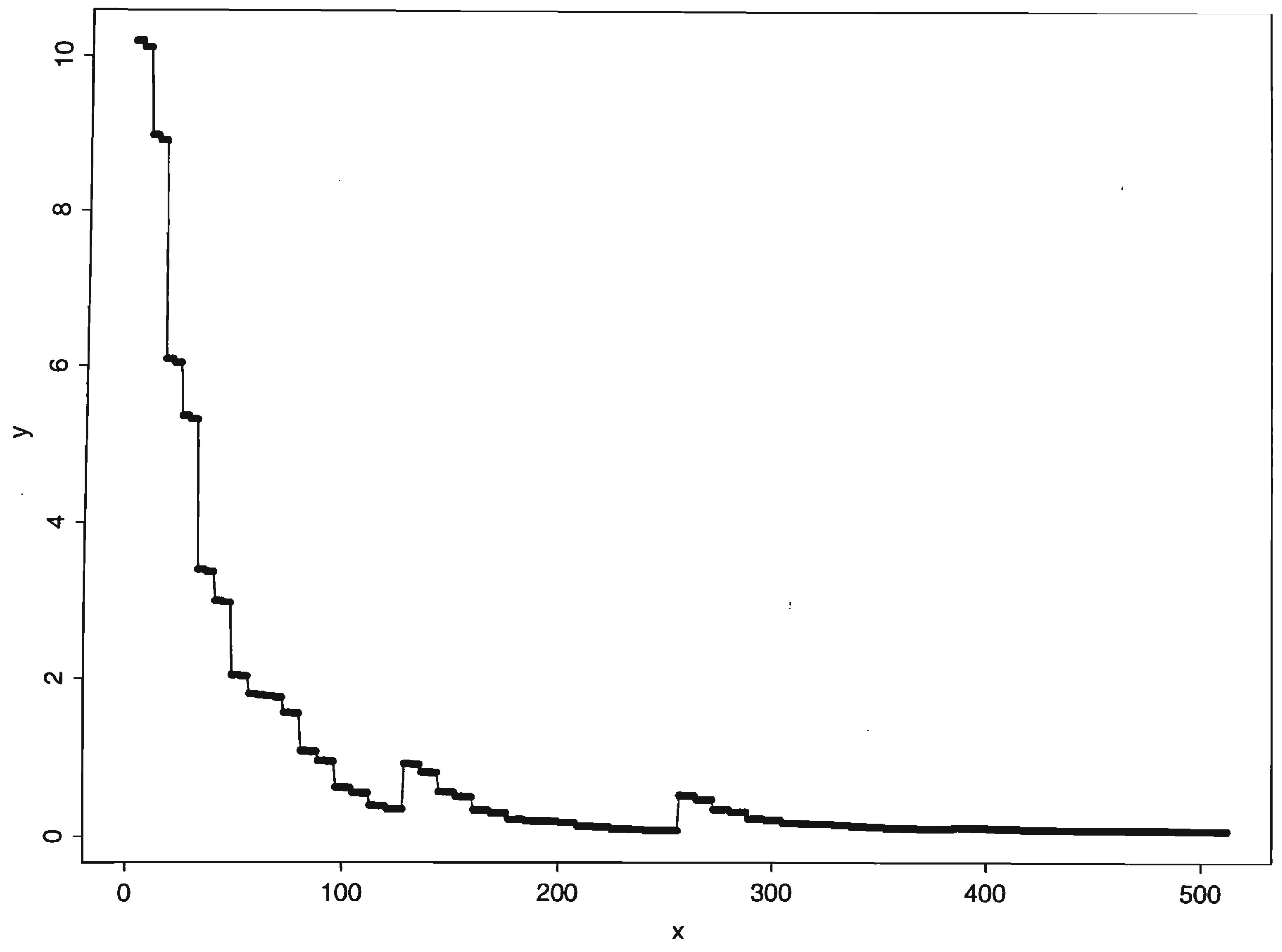

Chirp function N=512
y
x
1.0
0.5
0.0
-0.5
-1.0
0
100
200
300
400
500

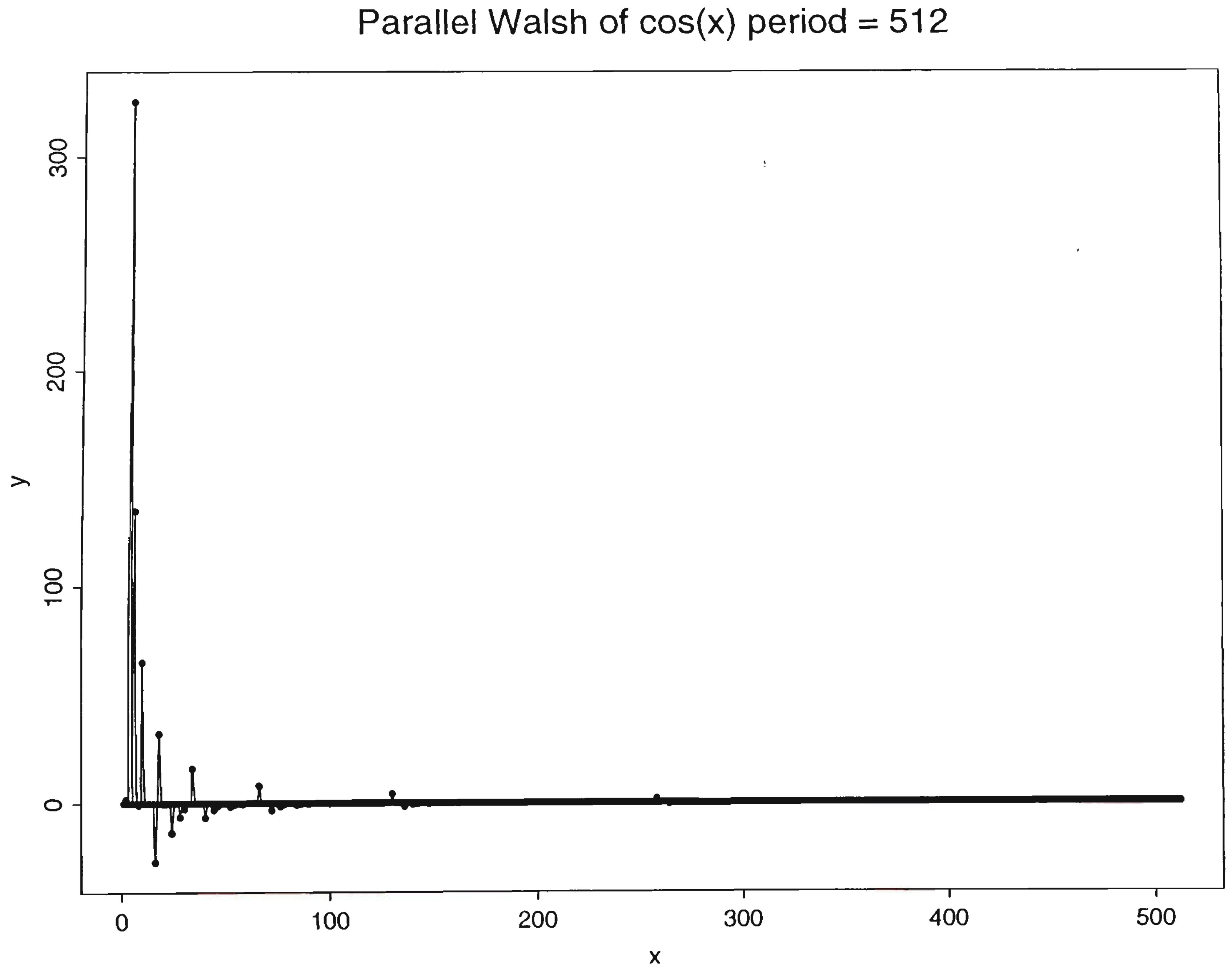

chirp N=64
y
1.0
0.5
0.0
-0.5
-1.0
x
0
100
200
300
400
500


Random data range 0-100
y
x
0
20
40
60
80
100
0
100
200
300
400
500

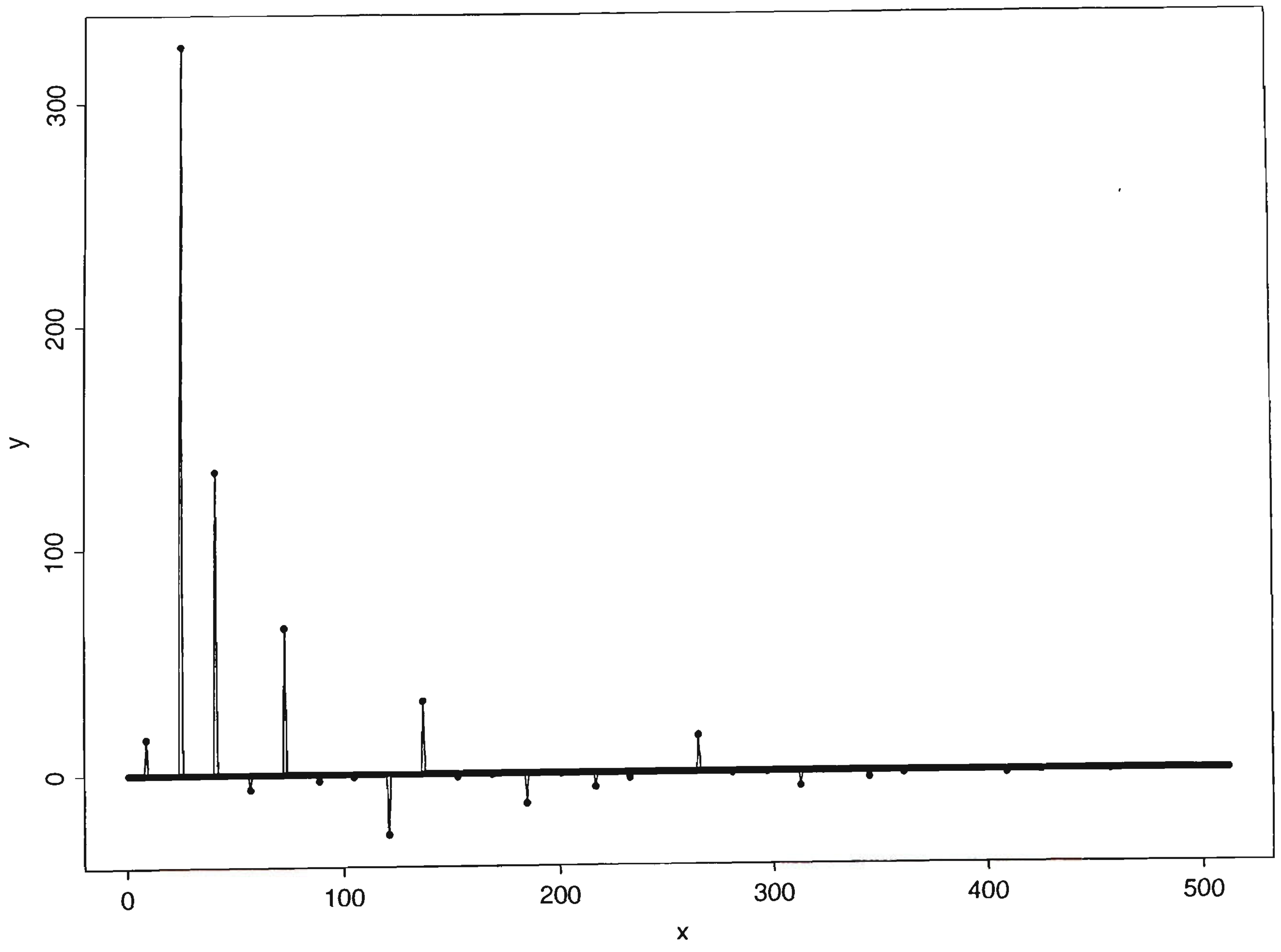

Parallel Walsh transform of delta function
y
-1.0
-0.5
0.0
0.5
1.0
x
0
100
200
300
400
500

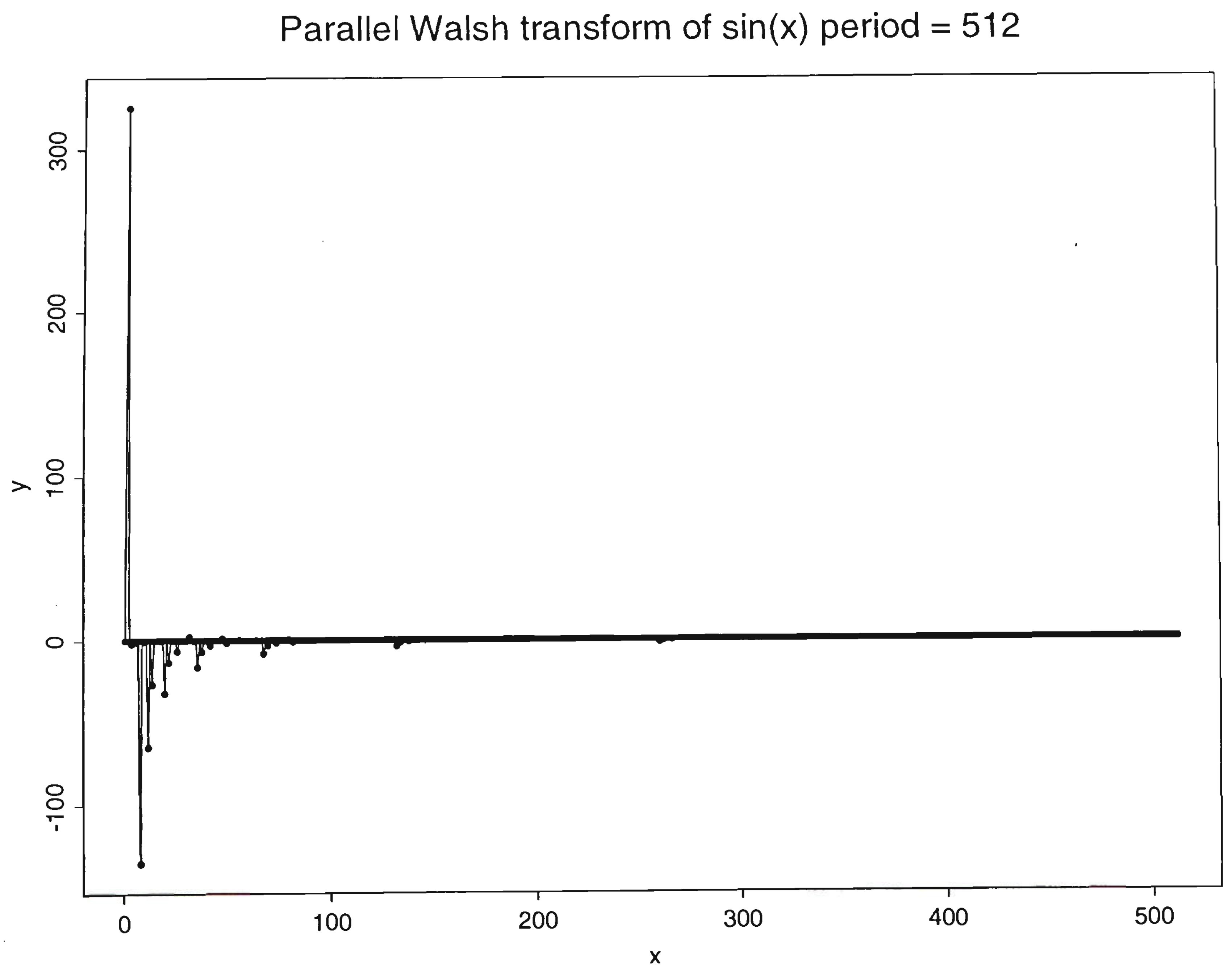
Parallel Walsh transform of iteration function



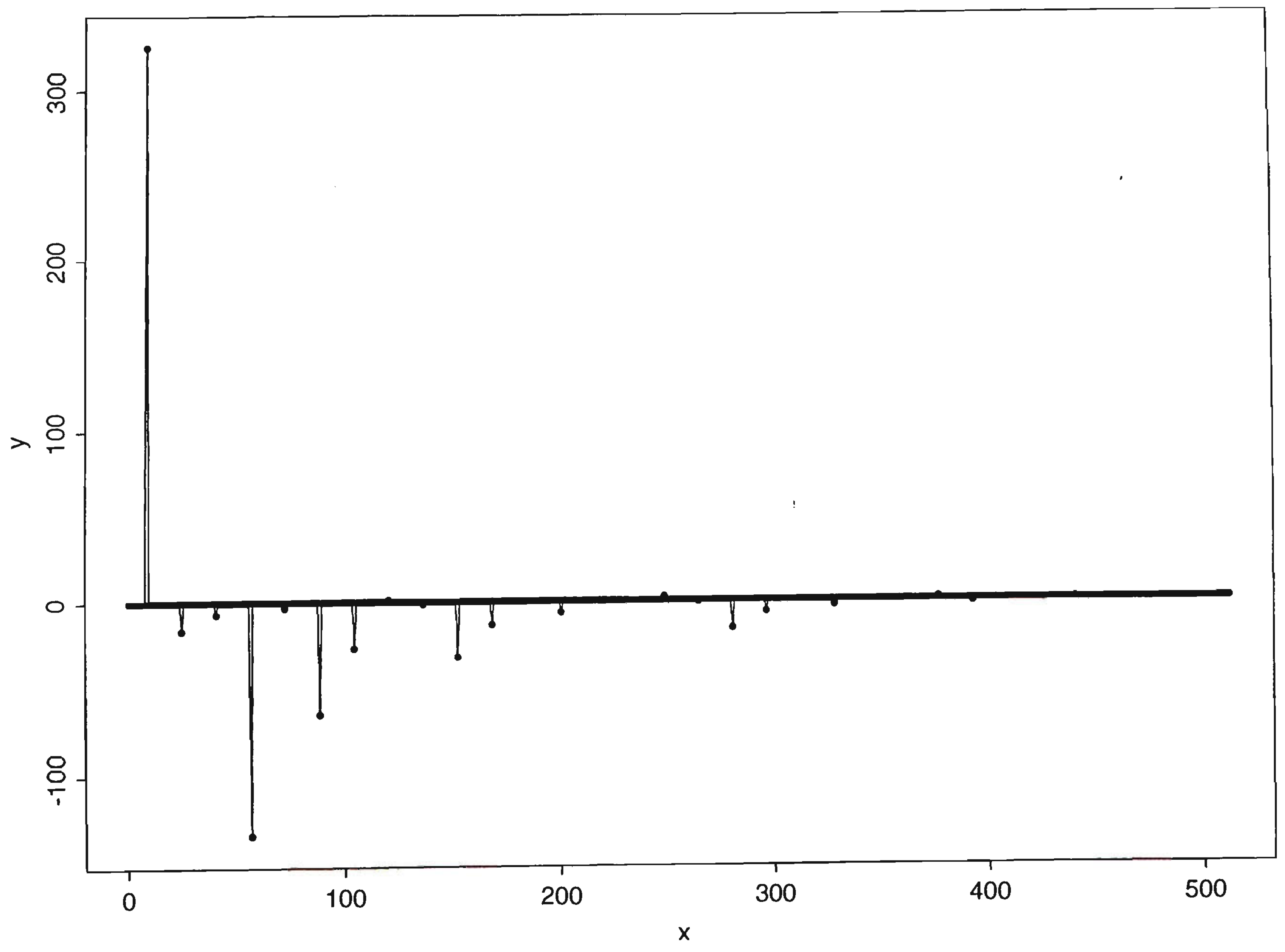

Parallel Walsh of cos(x) period = 512
y
0
100
200
300
x
0
100
200
300
400
500

Parallel Walsh transform of cos(x) period = 64

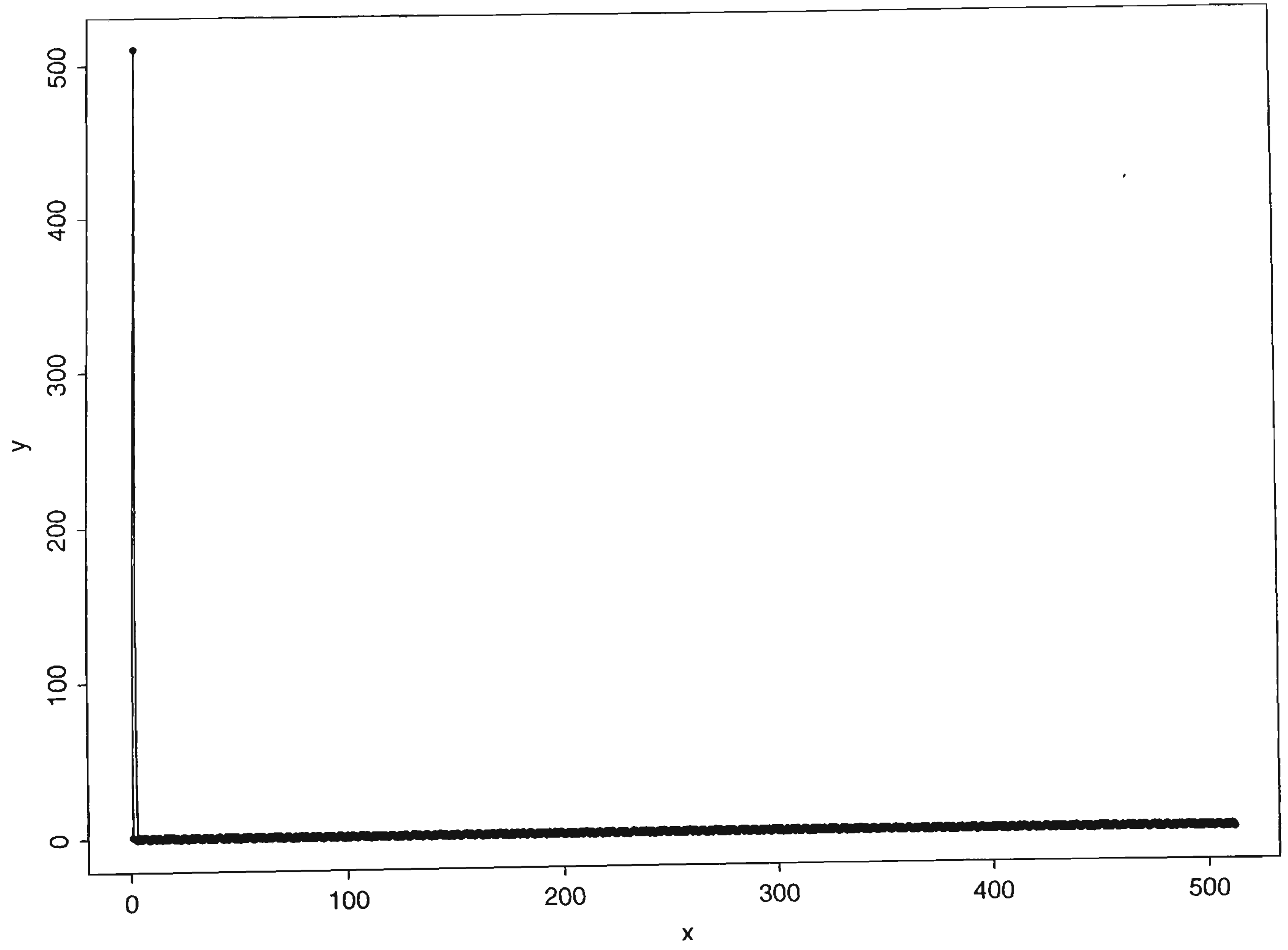


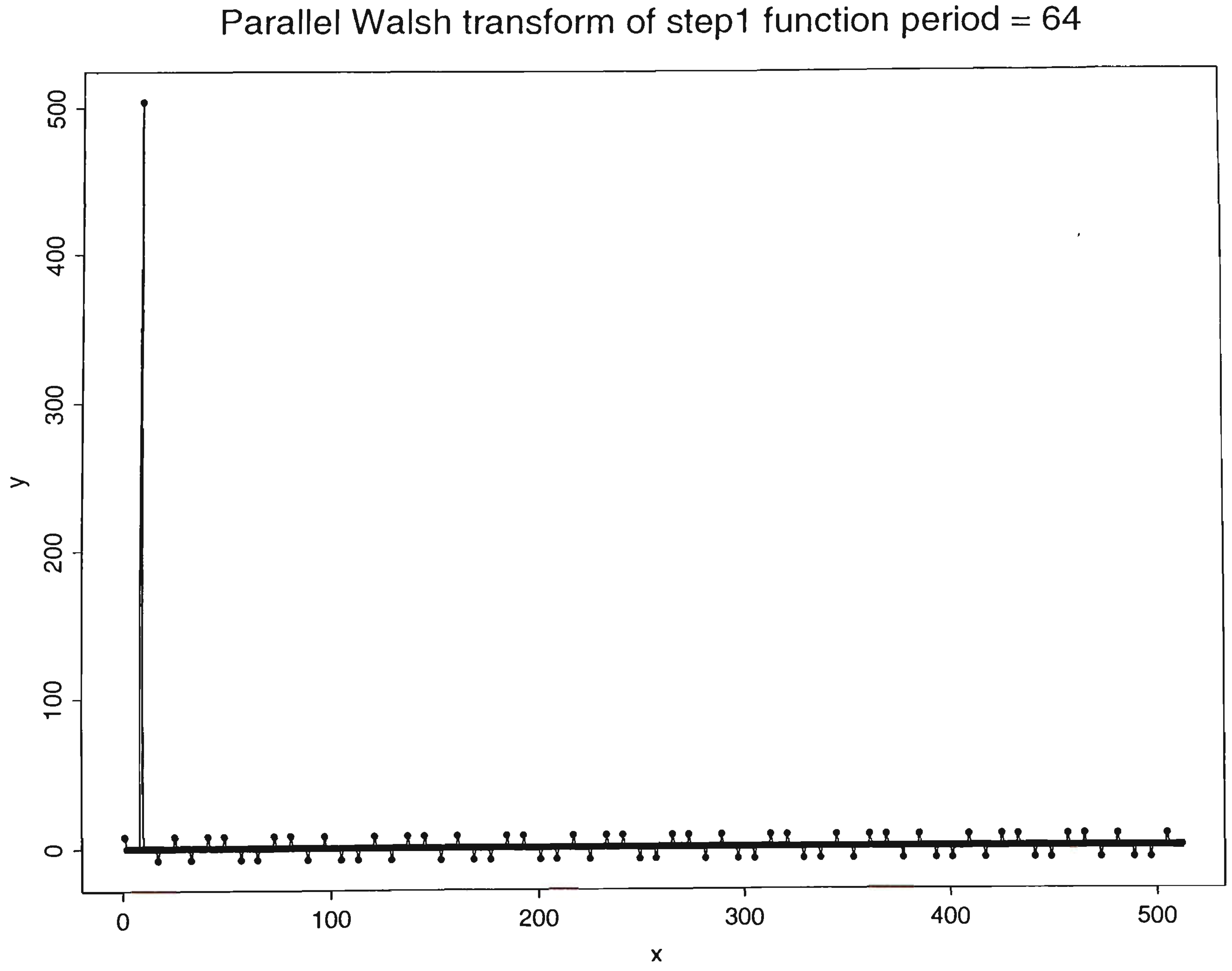

Parallel Walsh transform of sin(x) period = 512
y
300
200
100
0
-100
x
0
100
200
300
400
500

Parallel Walsh transform of sin(x) period = 64

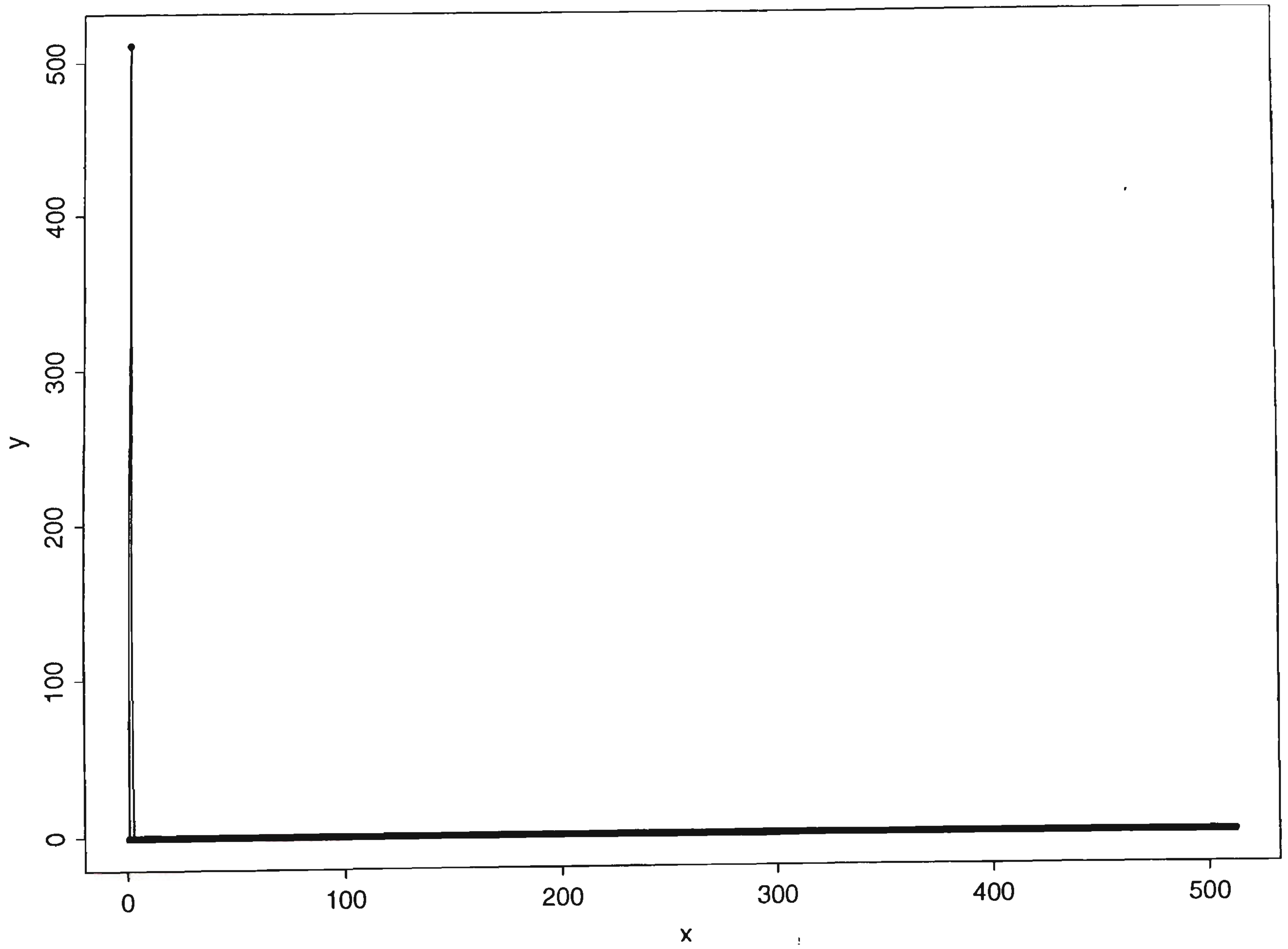


Parallel Walsh transform of step1 period = 512

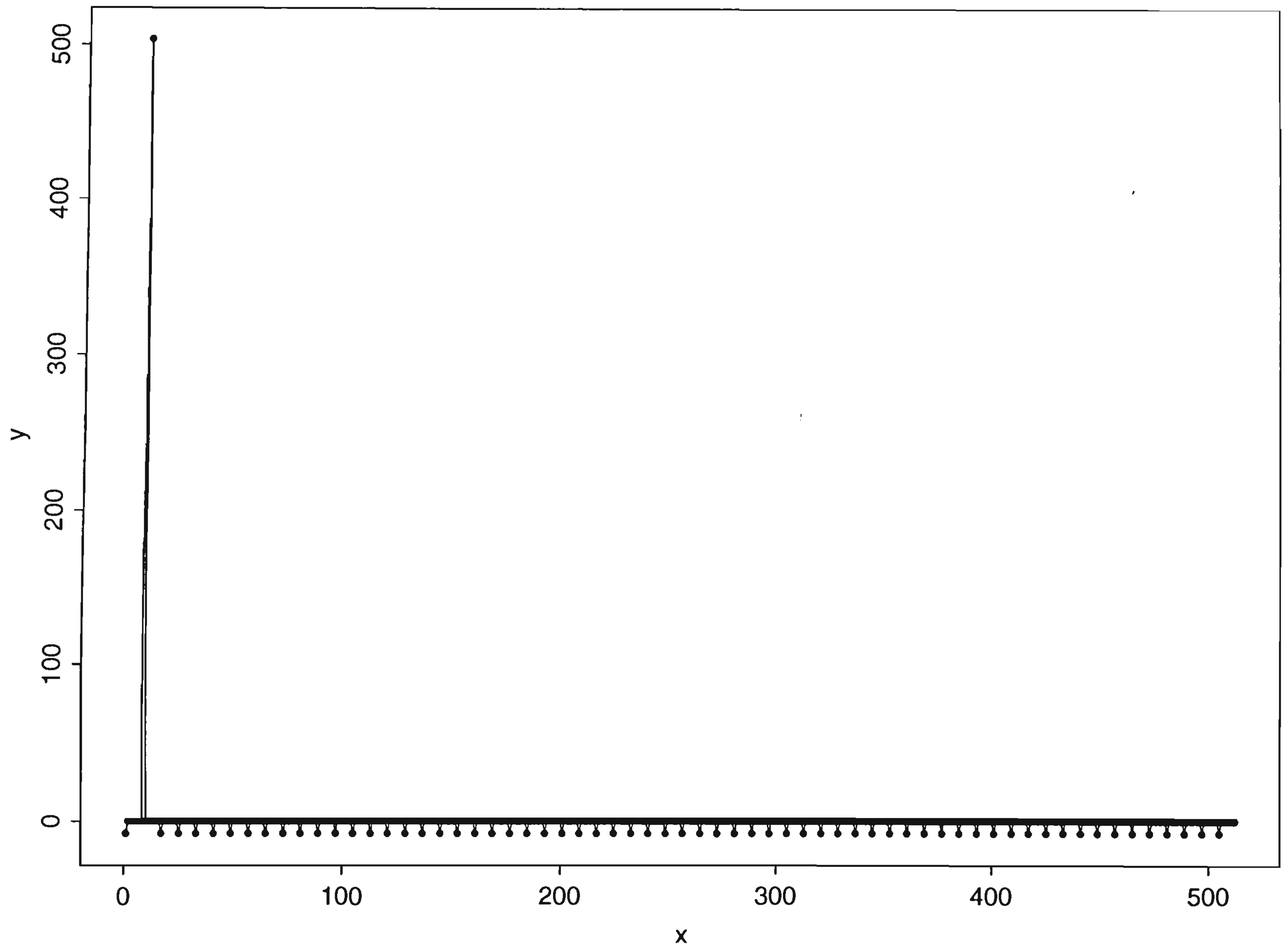



Parallel Walsh transform of step1 function period = 64
y
0
100
200
300
400
500
x
0
100
200
300
400
500

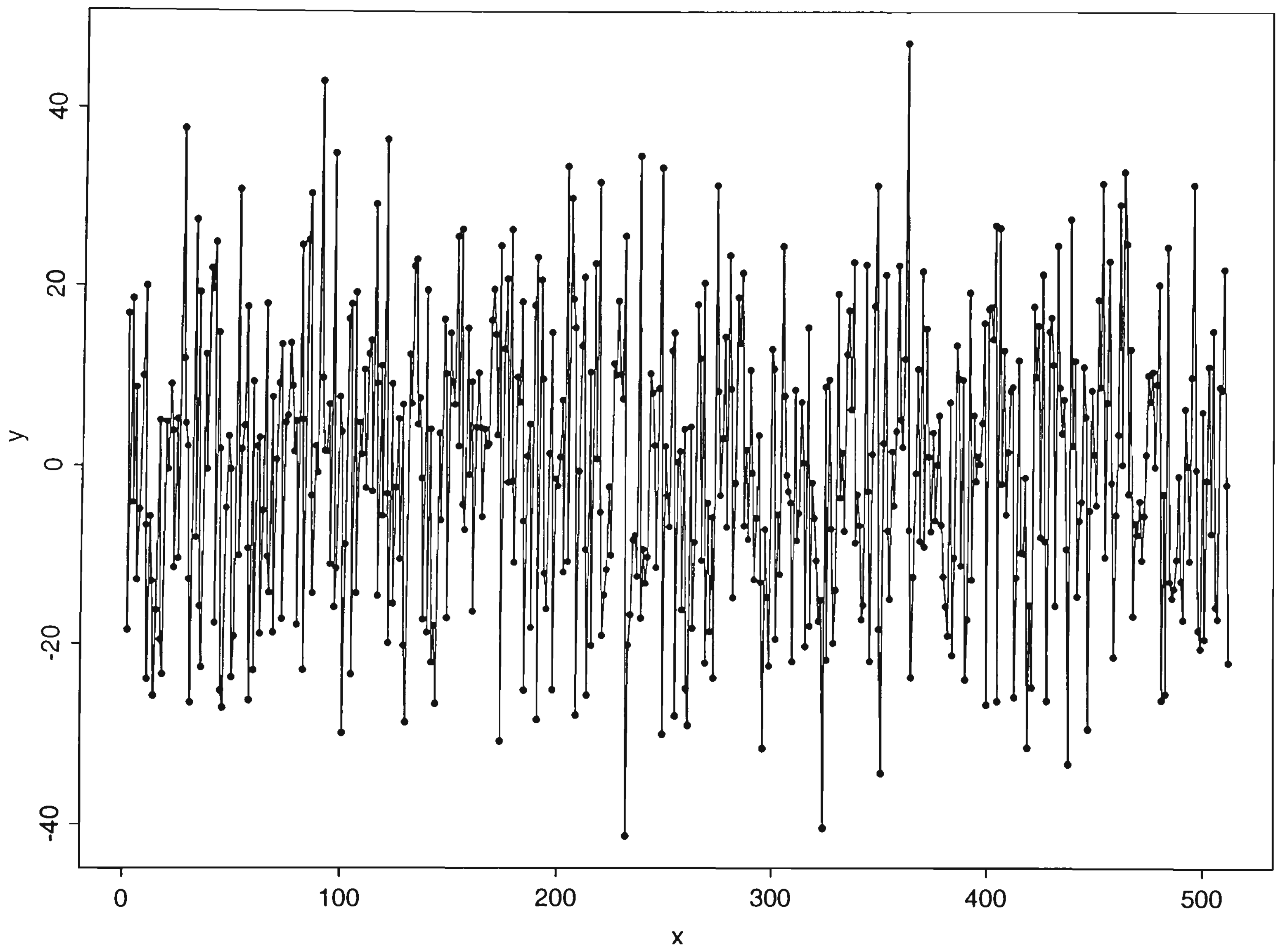
Parallel Walsh transform of step2 period = 512



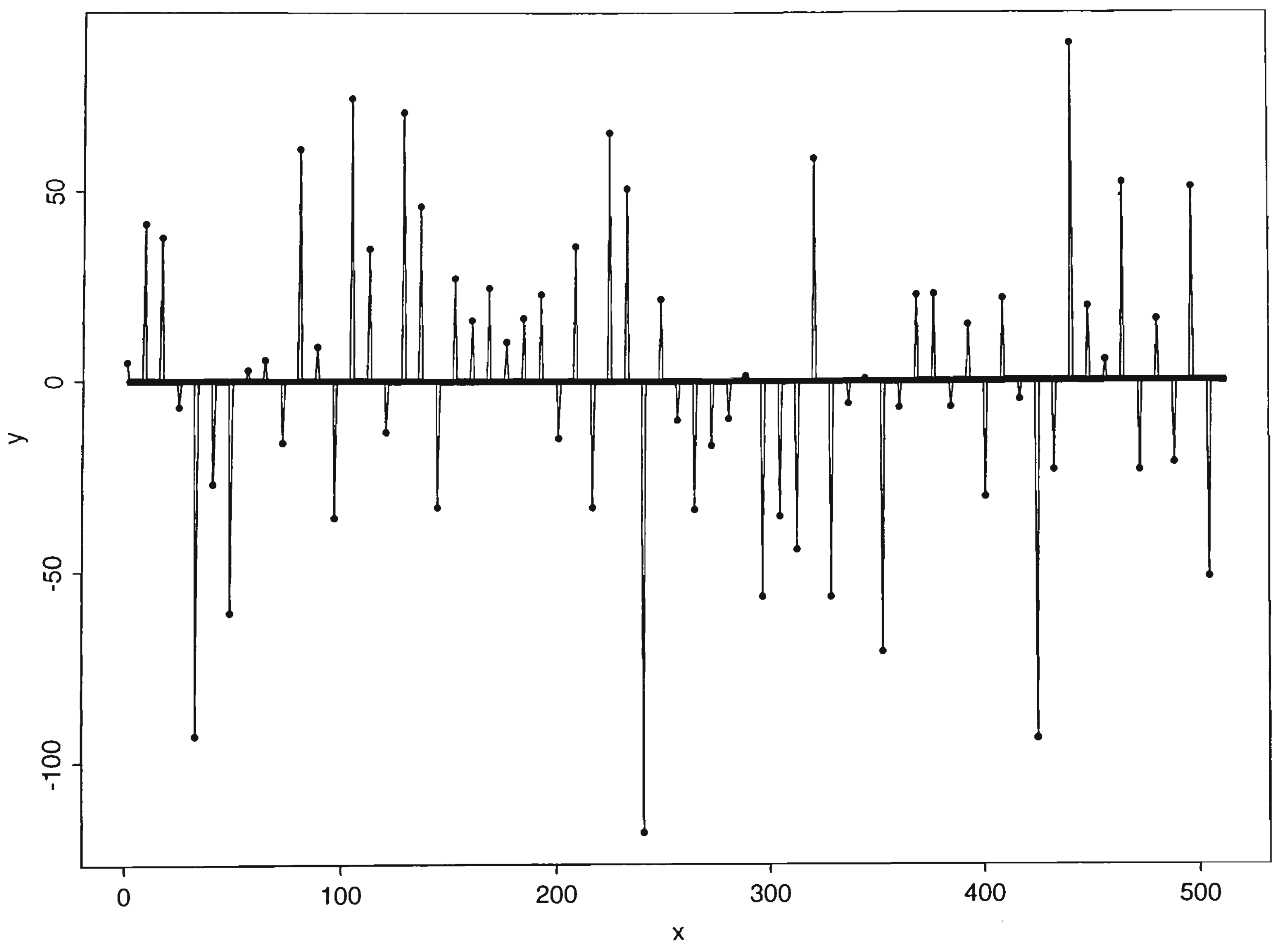
Parallel Walsh transform of step2 function period = 64

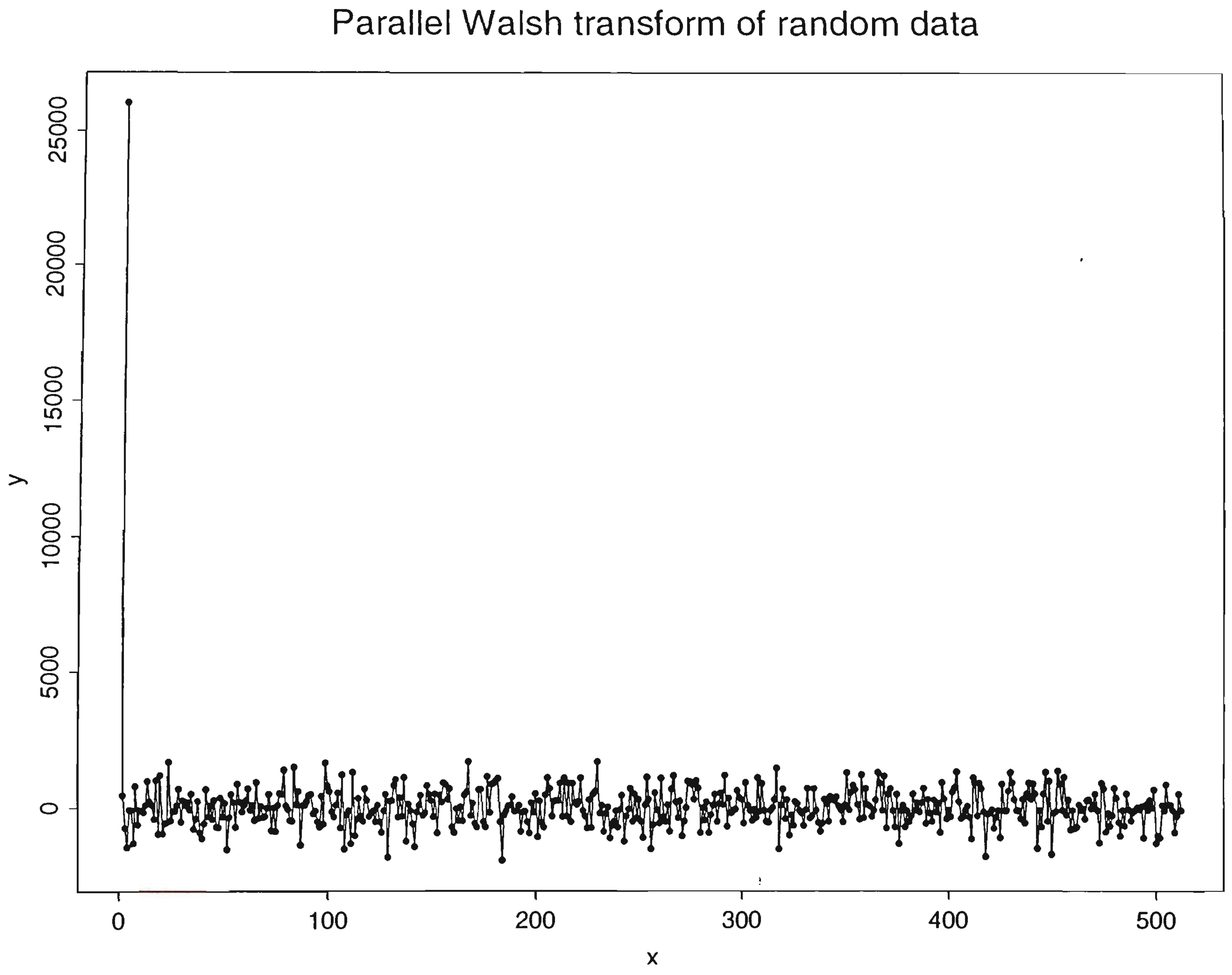


Parallel Walsh transform of chirp function period = 512

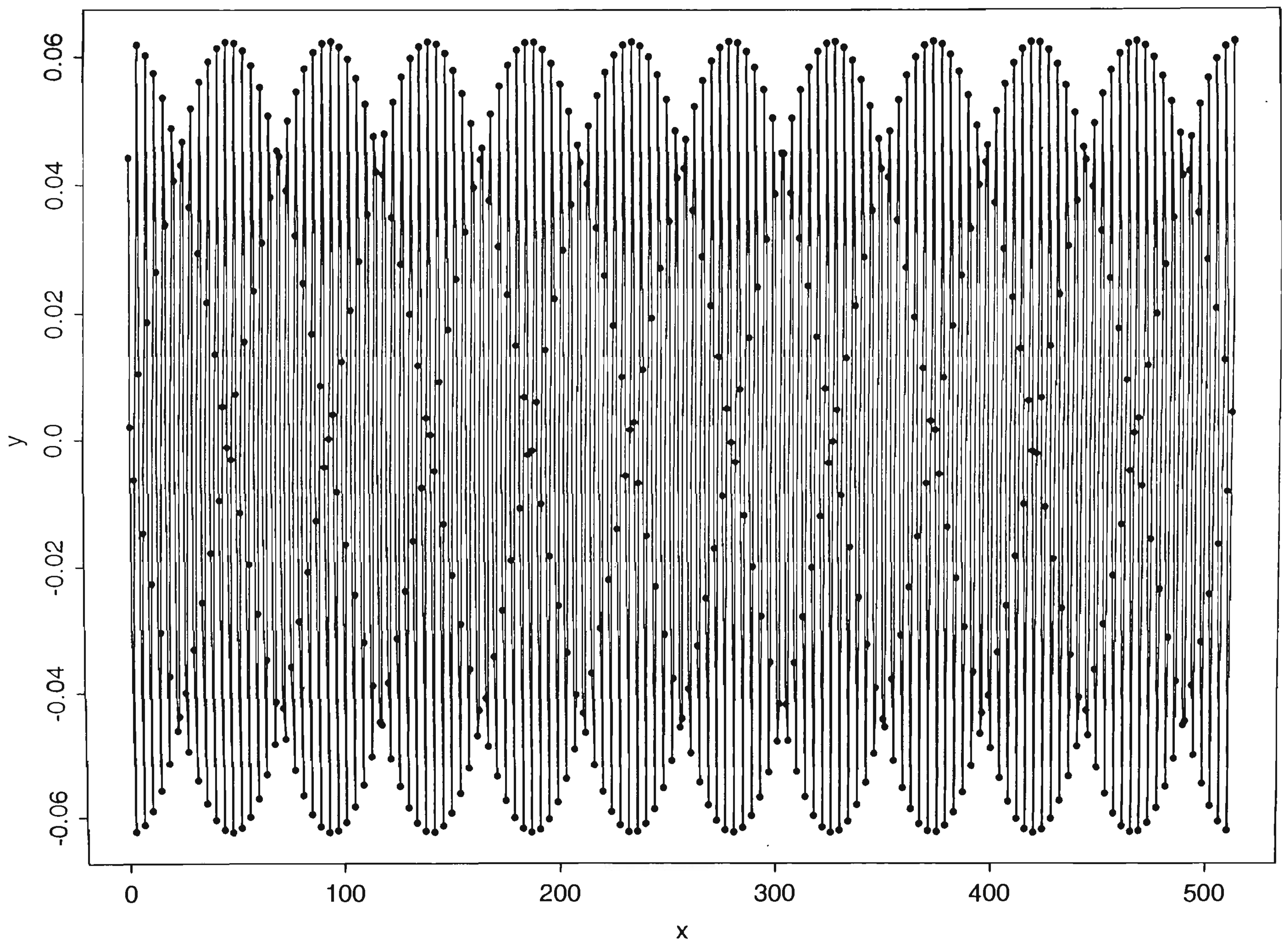


Parallel Walsh transform of chirp function period = 64

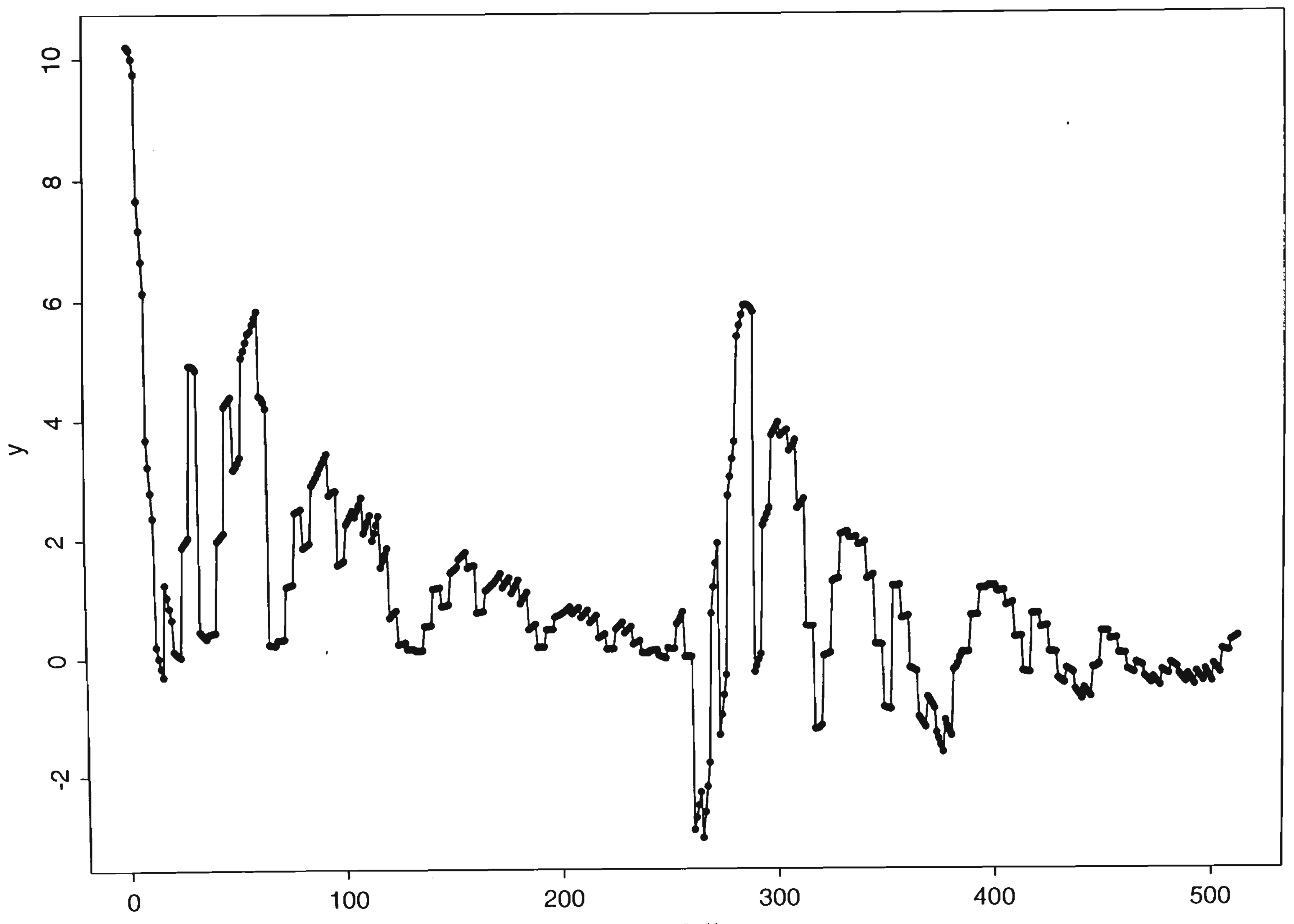



Parallel Walsh transform of random data
y
25000
20000
15000
10000
5000
0
0
100
200
300
400
500
x

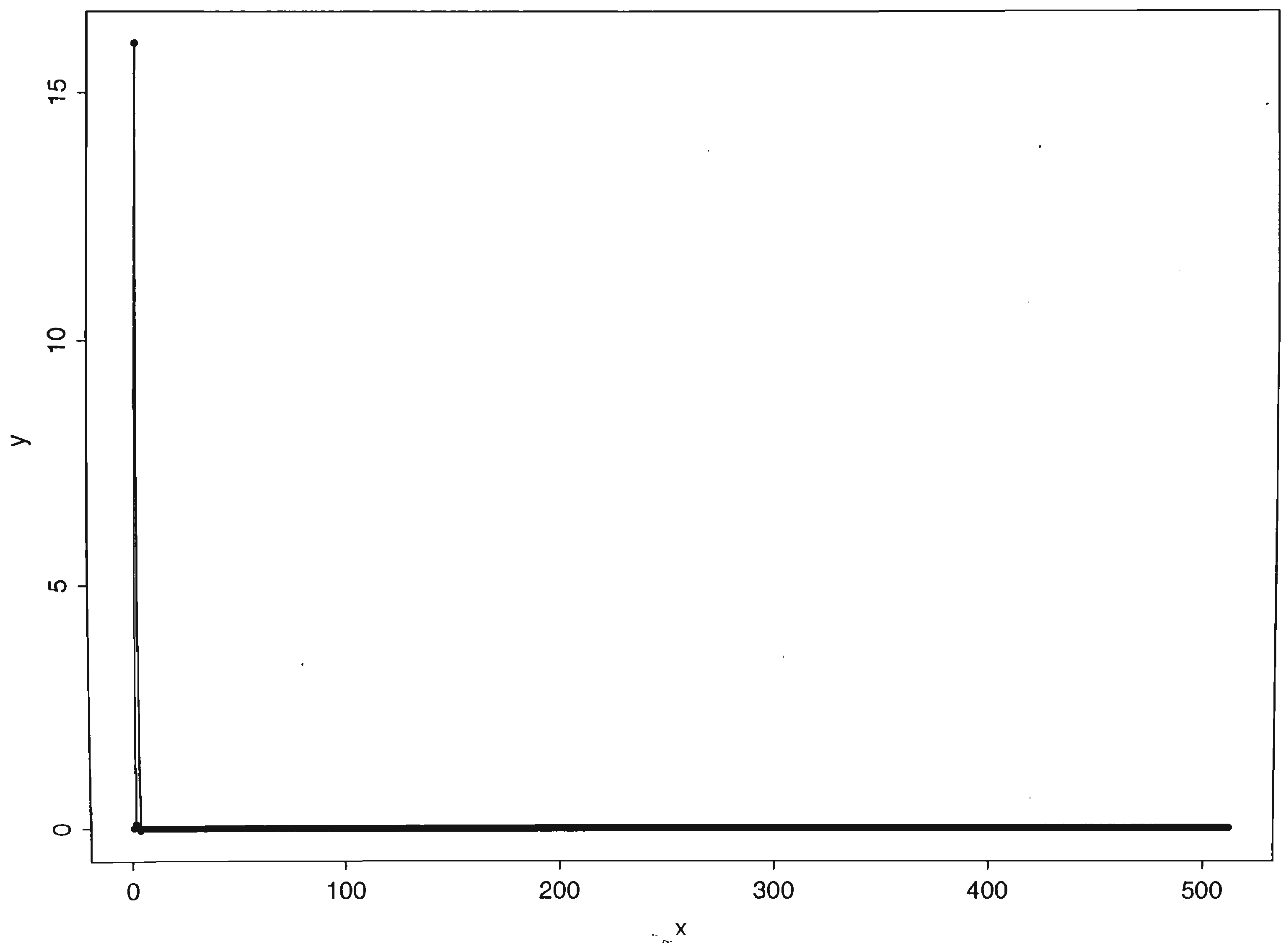
Parallel cosine transform of delta function



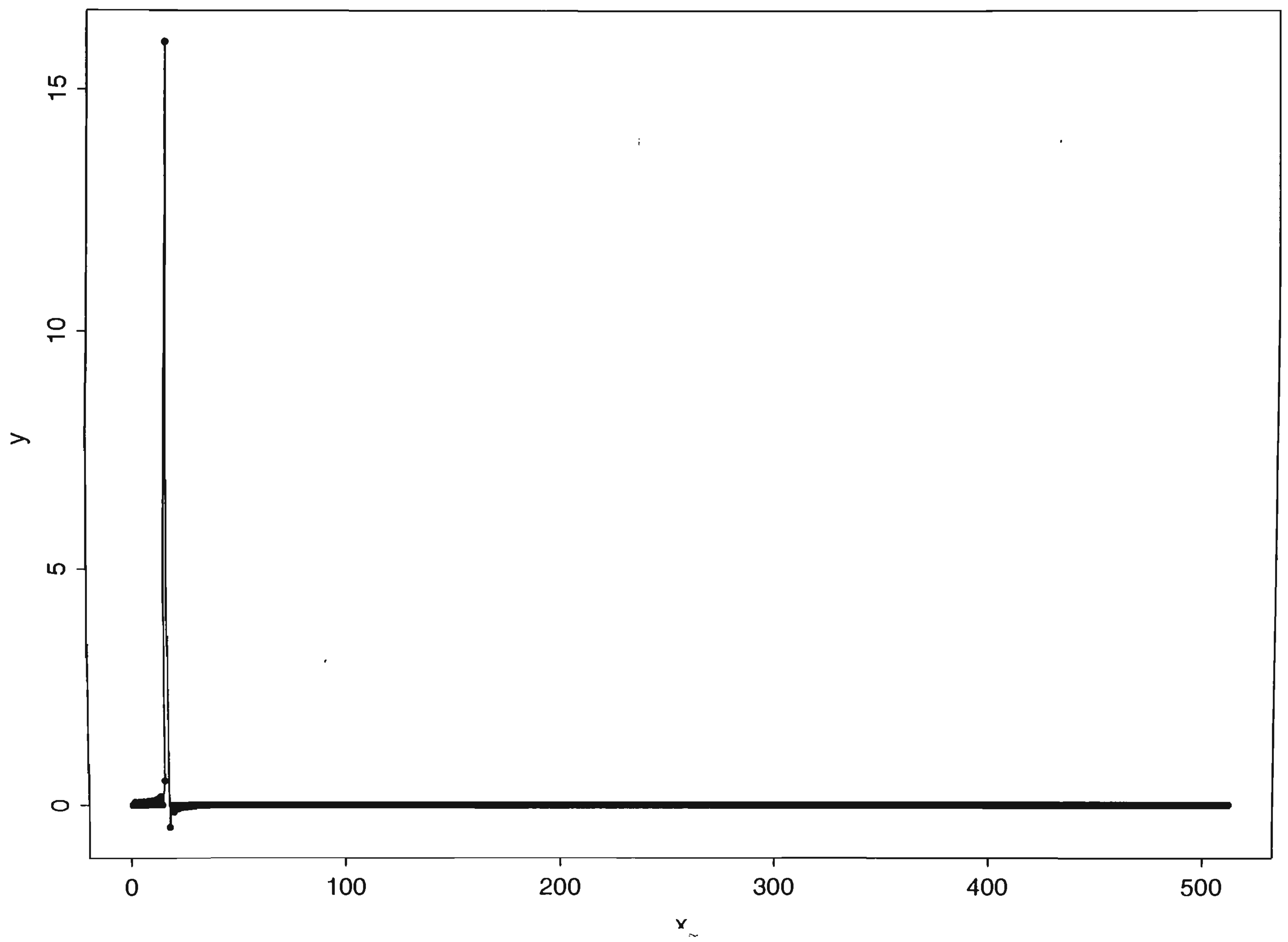
Parallel cosine transform of iteration function

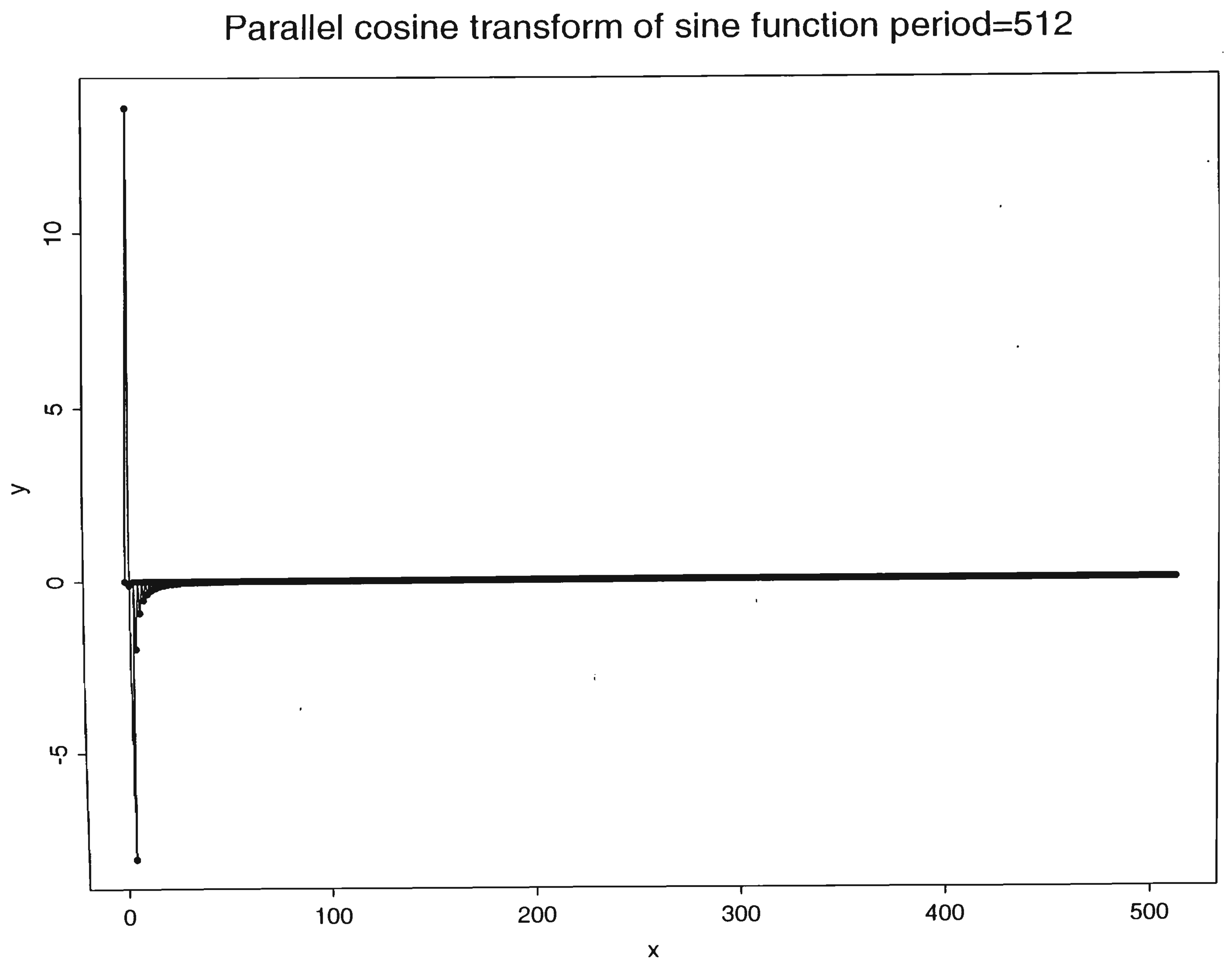


Parallel cosine transform of cosine function period=512

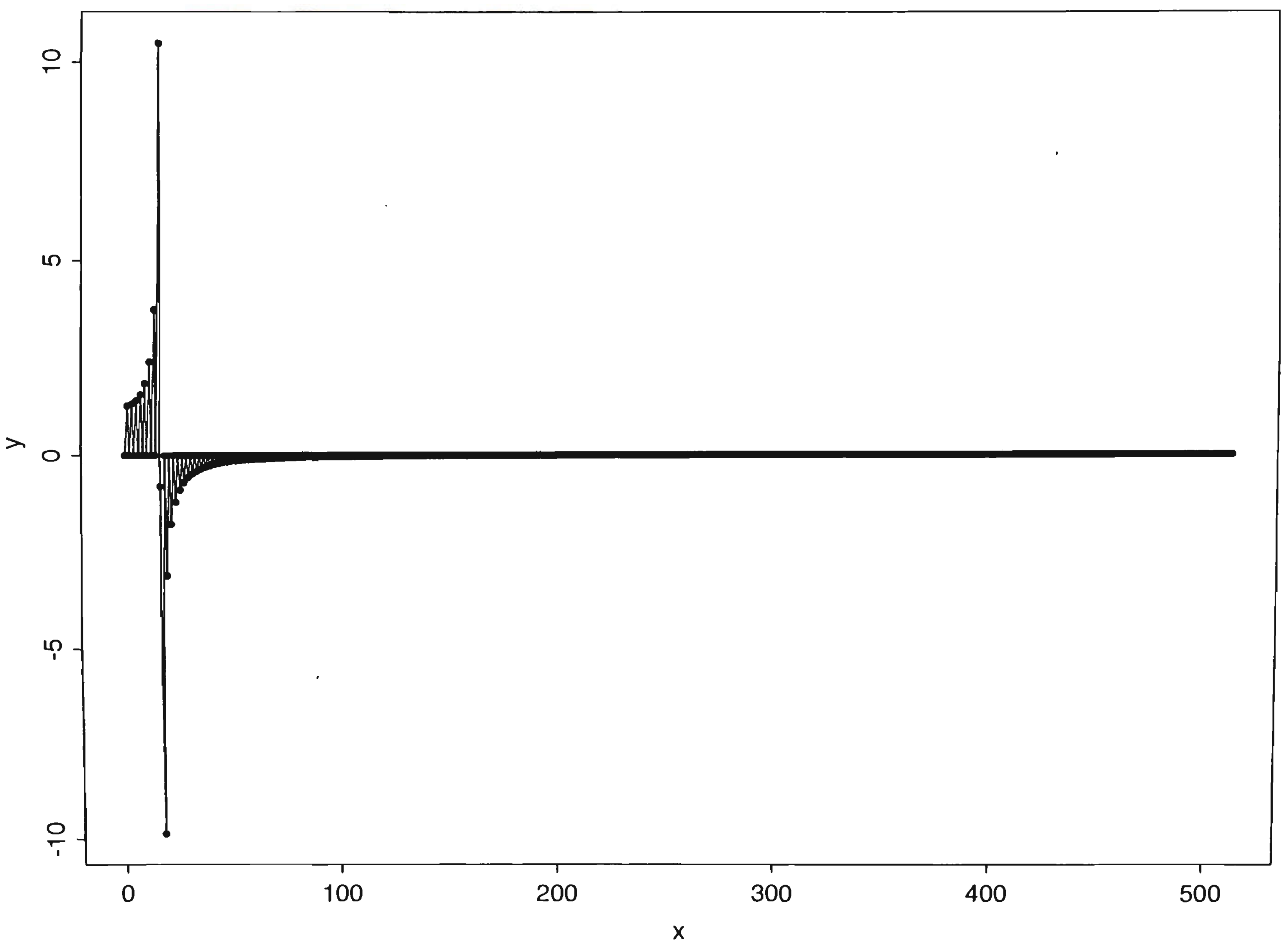


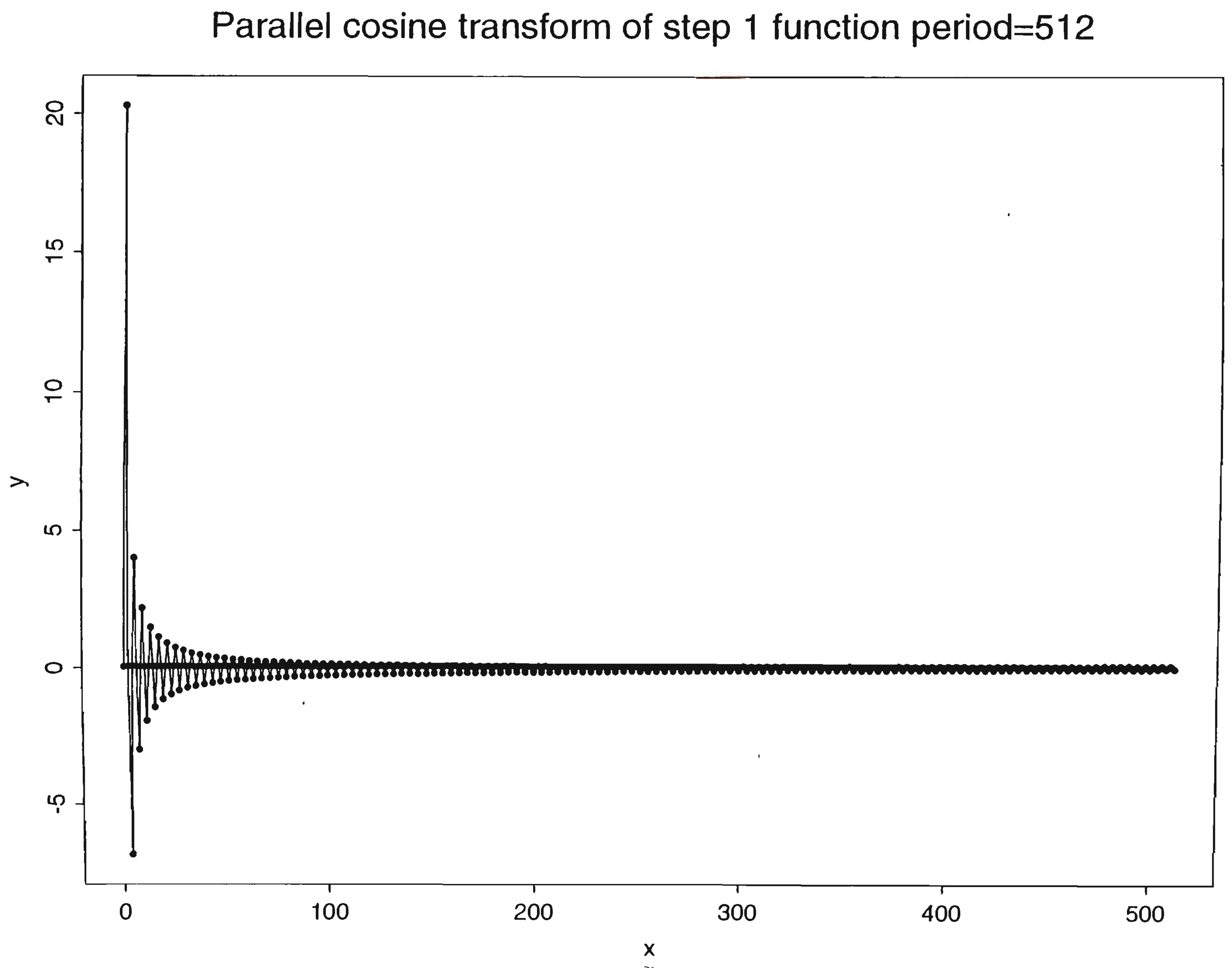
Parallel cosine transform of cosine function period=64



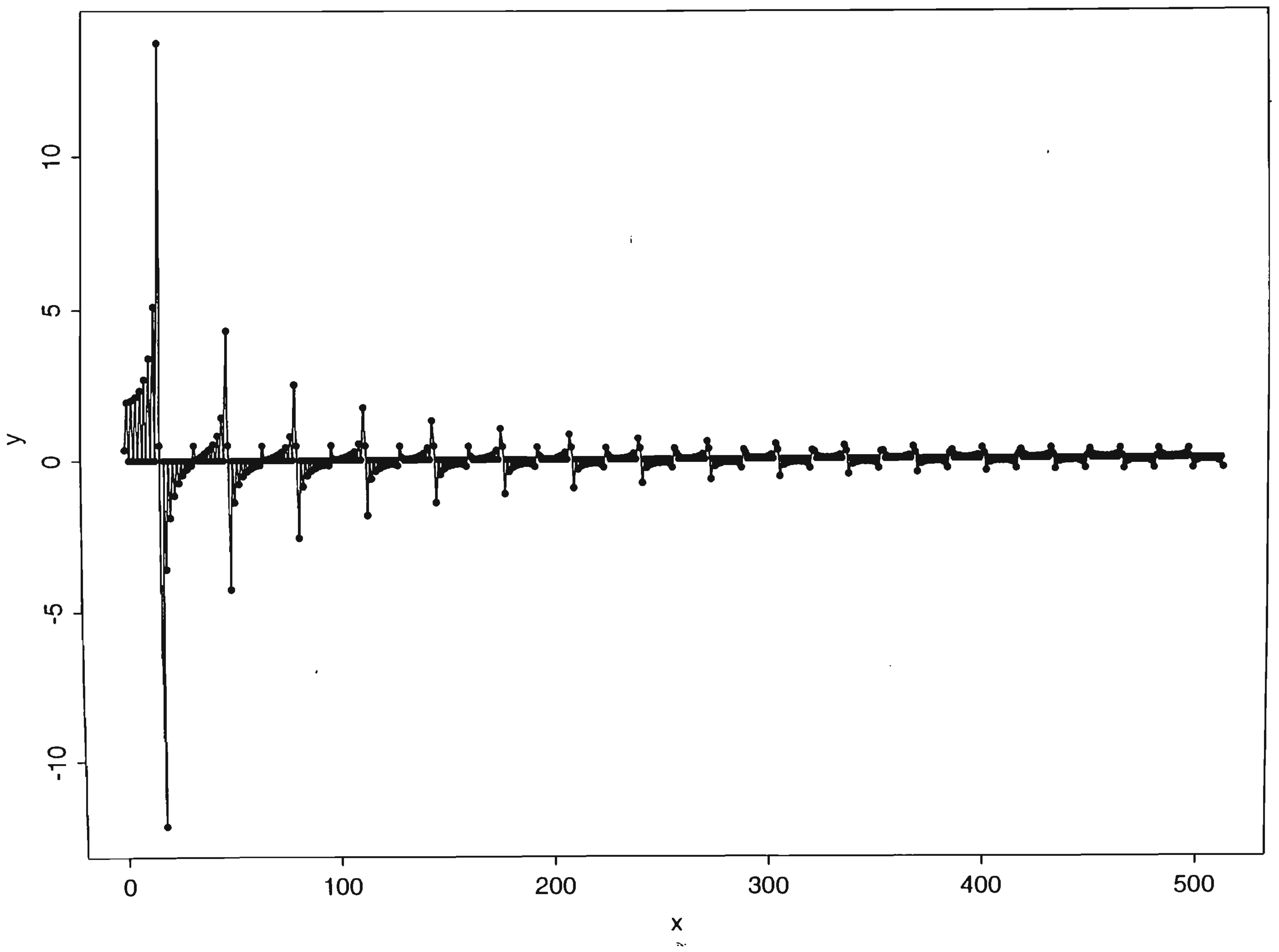

Parallel cosine transform of sine function period=512
y
10
5
0
-5
x
0
100
200
300
400
500

Parallel cosine transform of sine function period=64

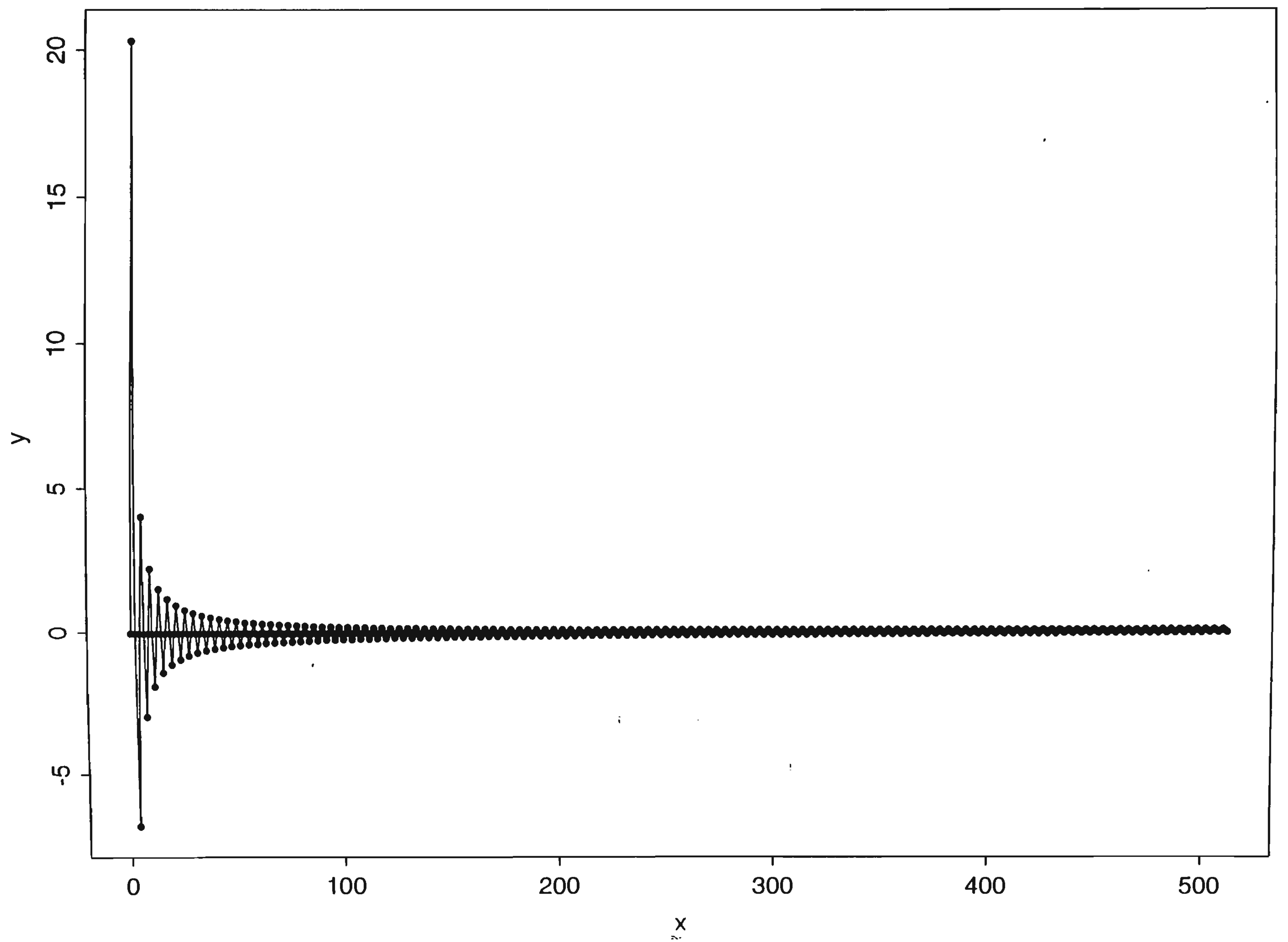



Parallel cosine transform of step 1 function period=512
y
x
20
15
10
5
0
-5
0
100
200
300
400
500

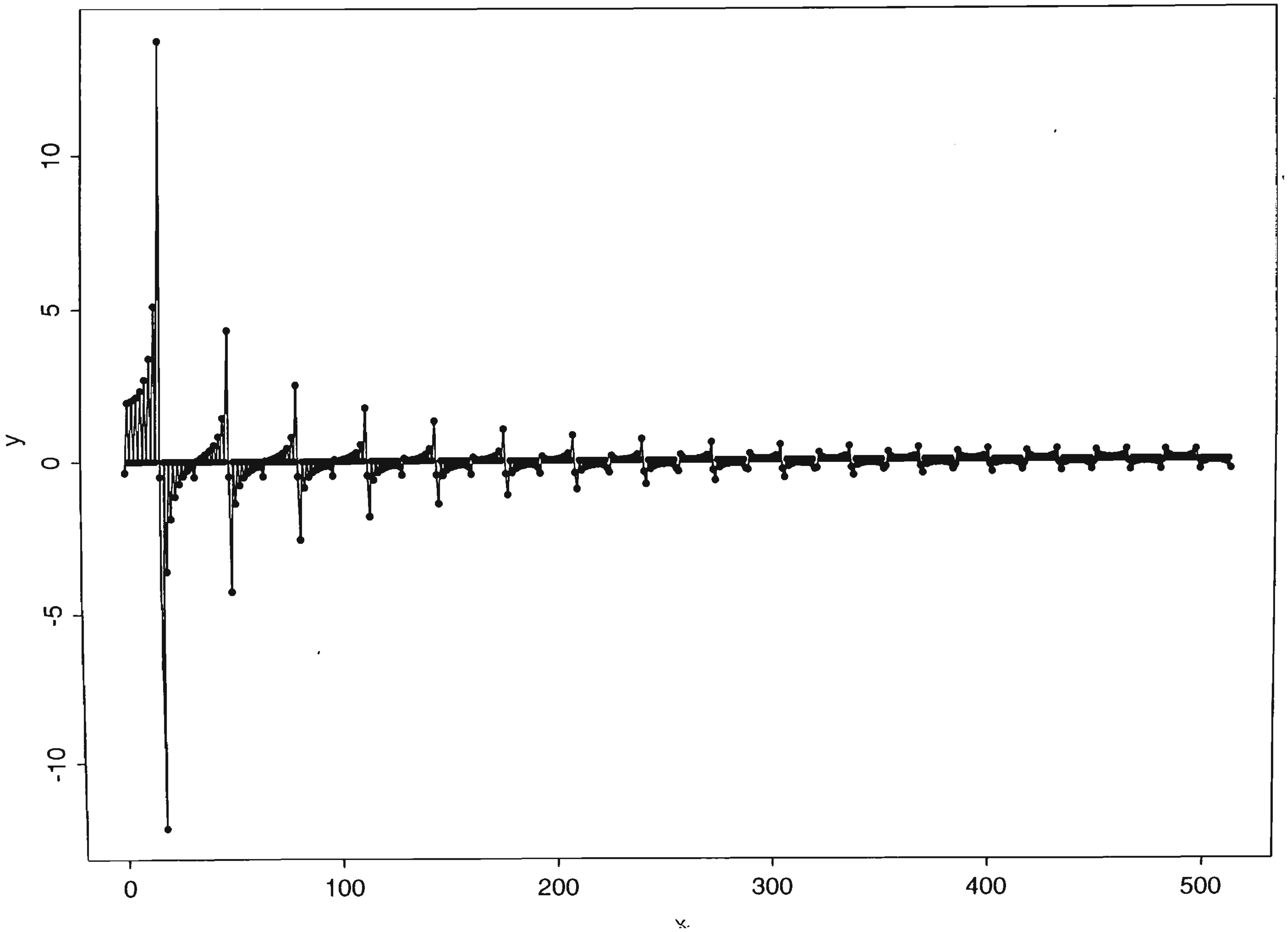
Parallel cosine transform of step 1 function period=64



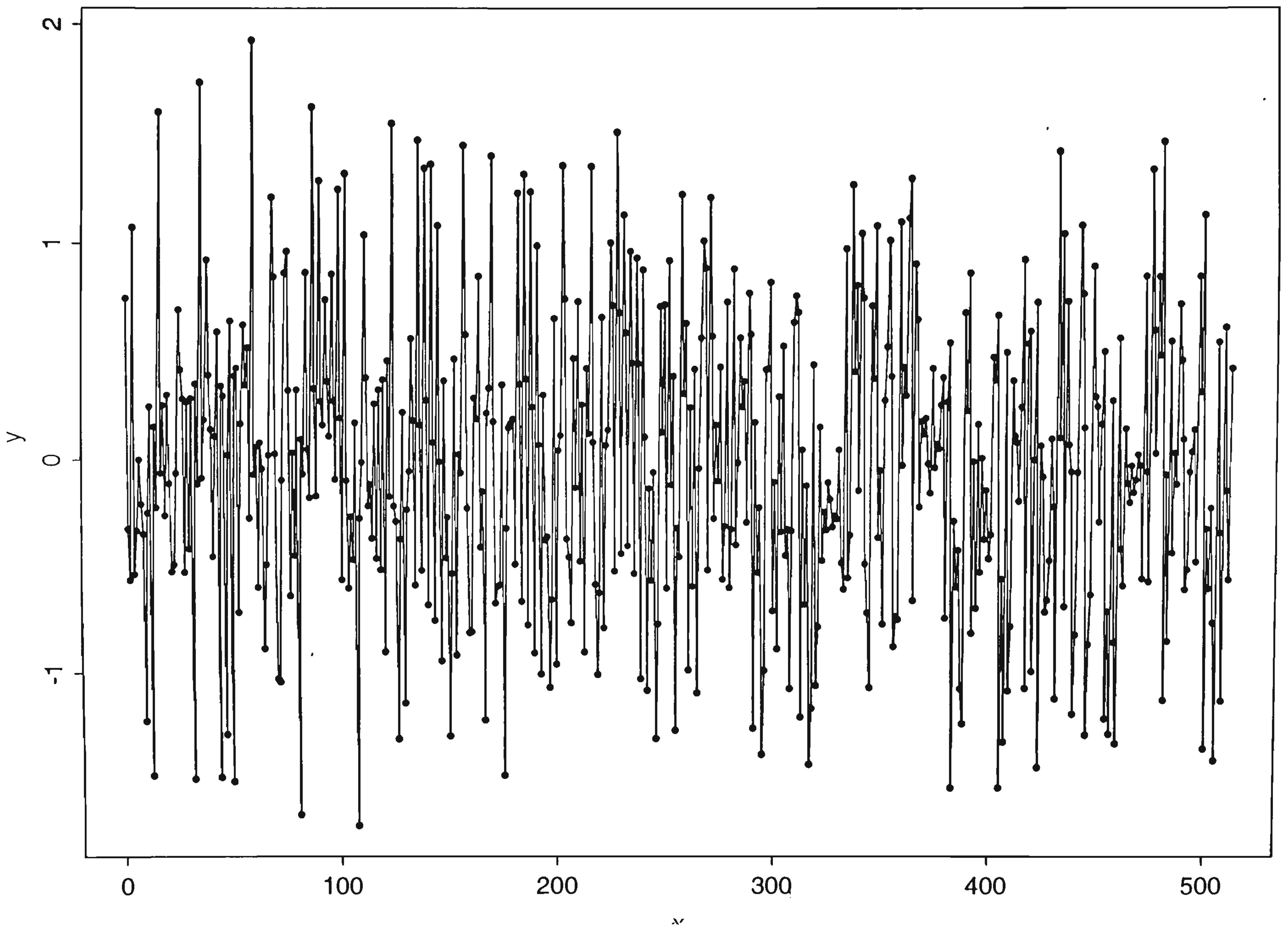
Parallel cosine transform of step 2 function period=512



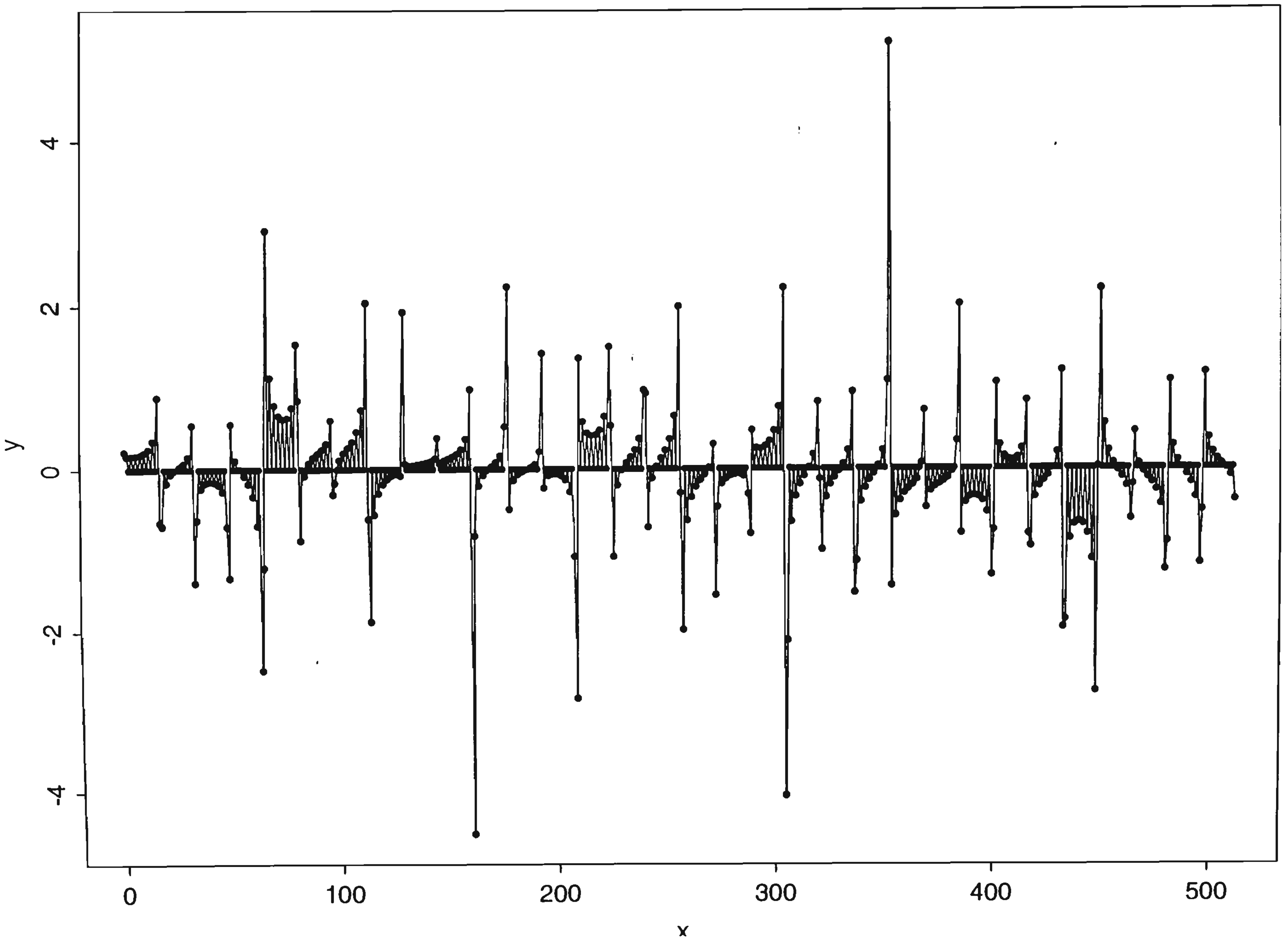
Parallel cosine transform of step 2 function period=64

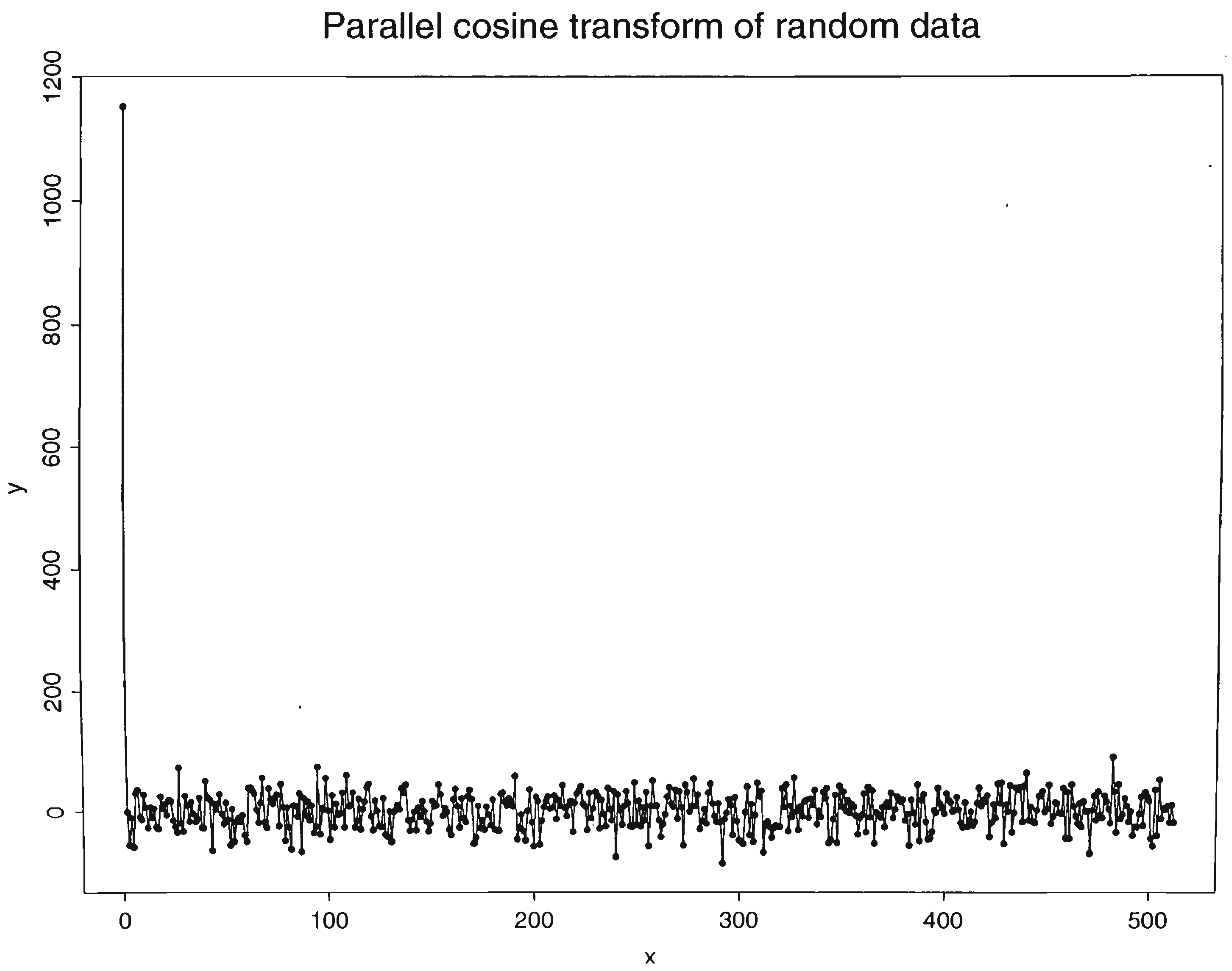


Parallel cosine transform of chirp function period=512



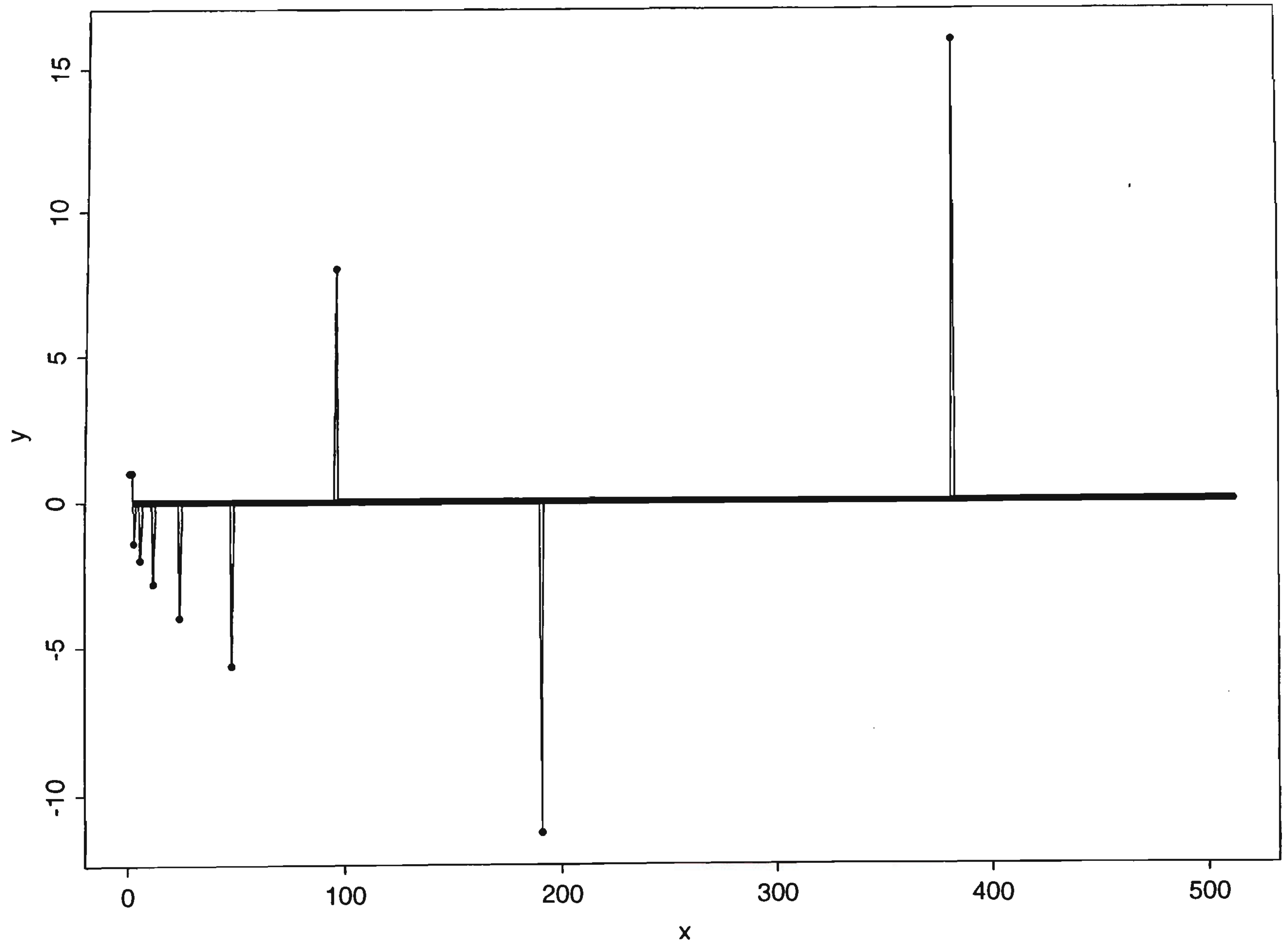
Parallel cosine transform of chirp function period=64

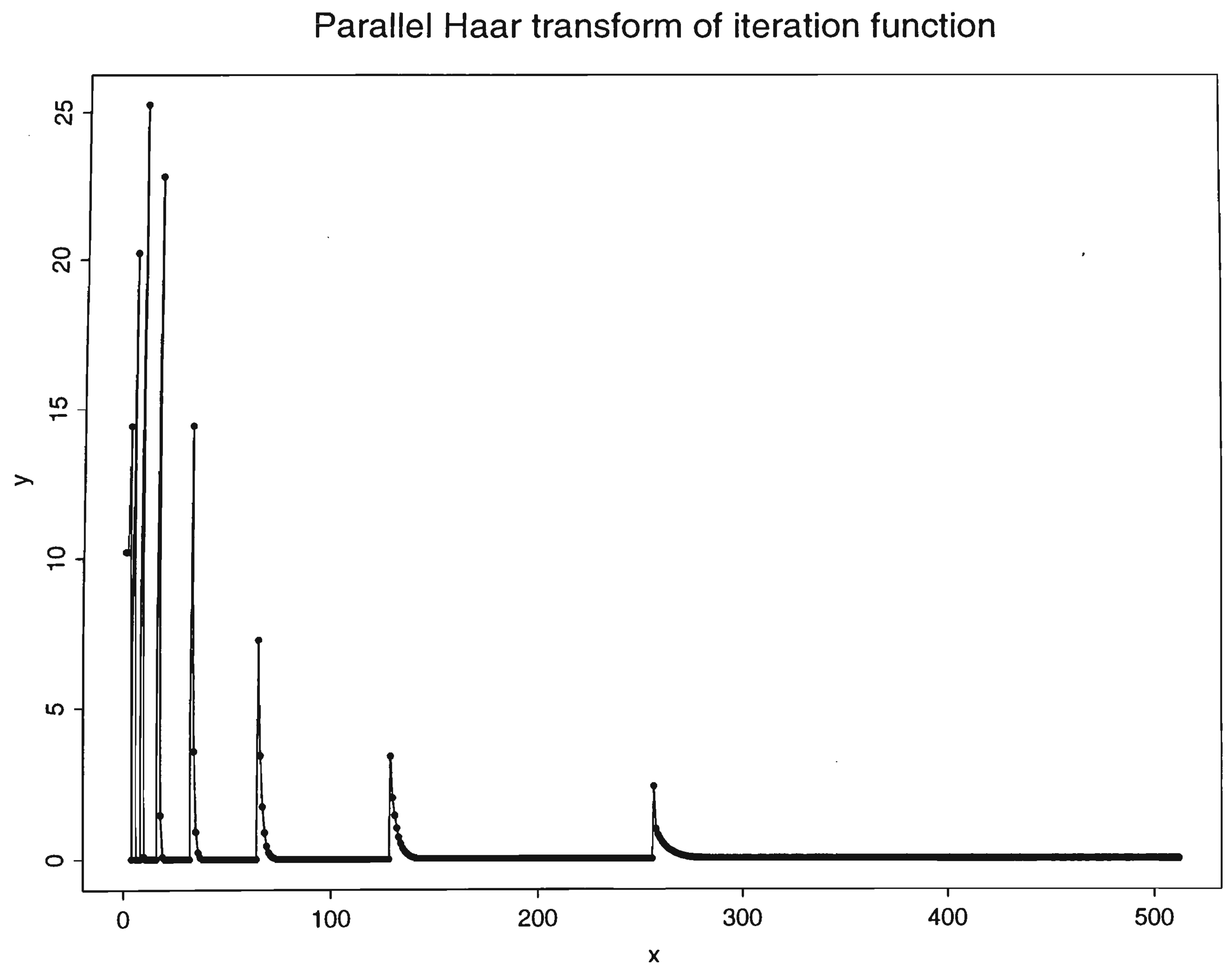




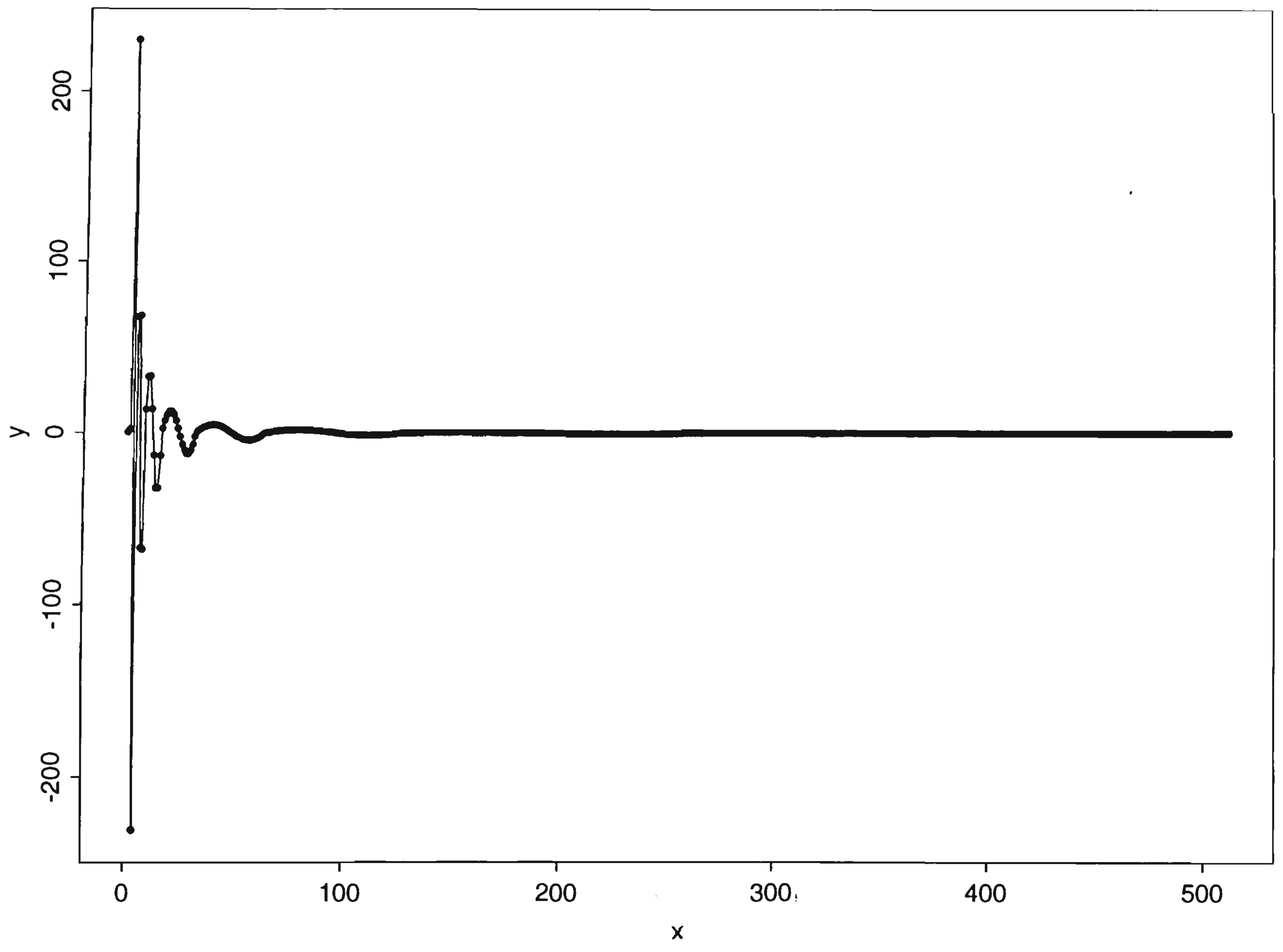
Parallel cosine transform of random data
1200
1000
800
600
400
200
0
y
0
100
200
300
400
500
x

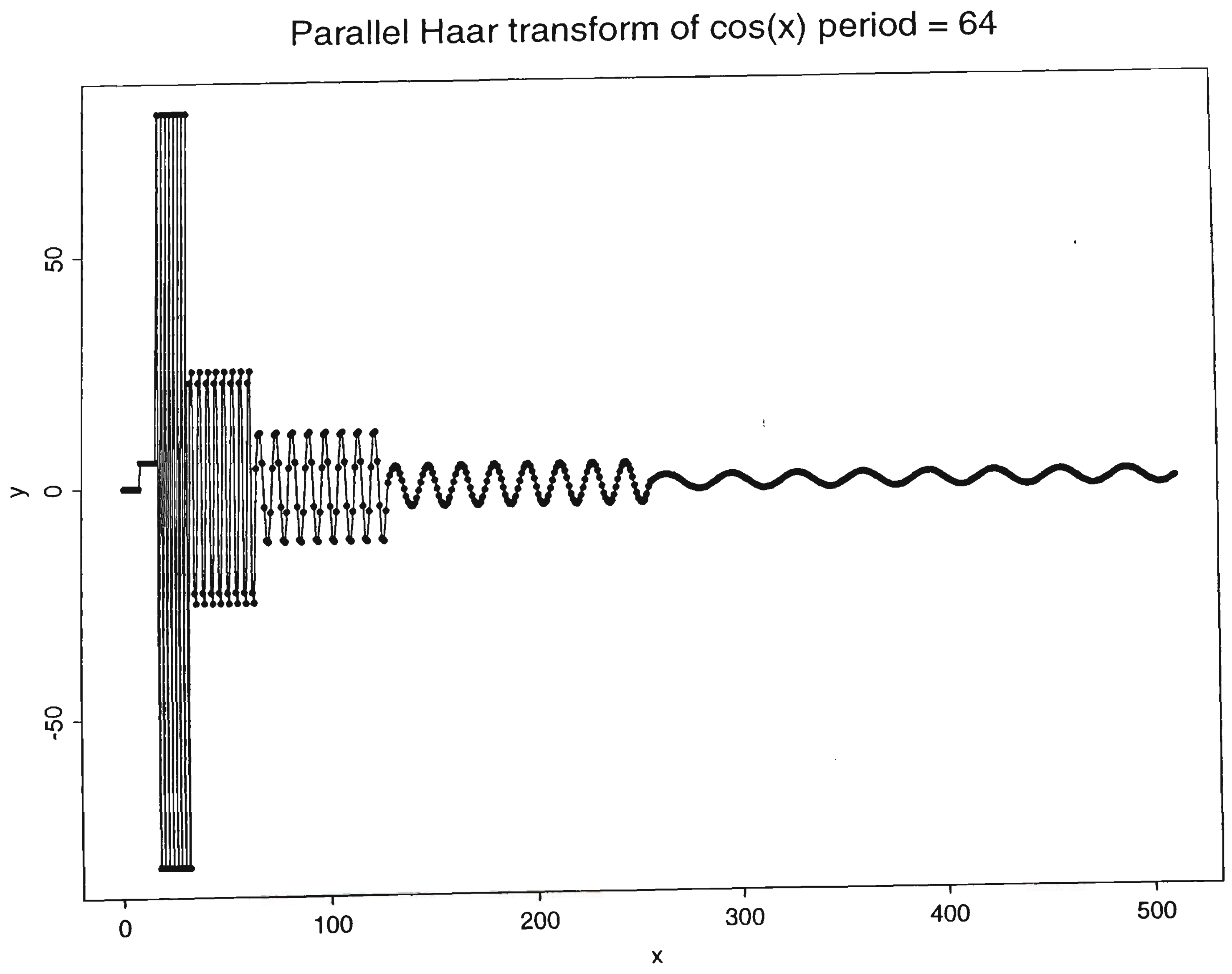
Parallel Haar transform of delta function



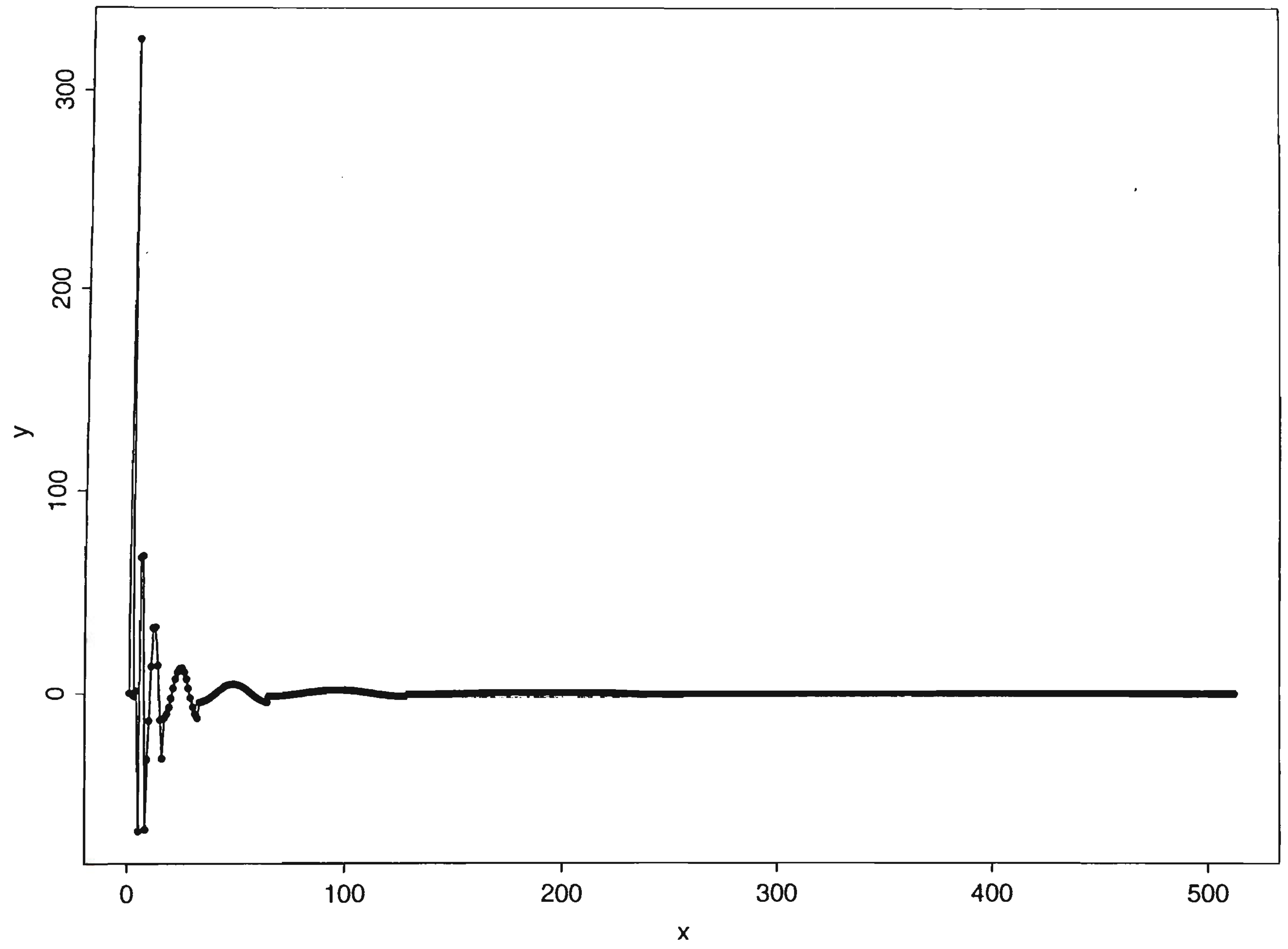

Parallel Haar transform of iteration function
y
x
0
5
10
15
20
25
0
100
200
300
400
500

Parallel Haar transform of cos(x) period = 512

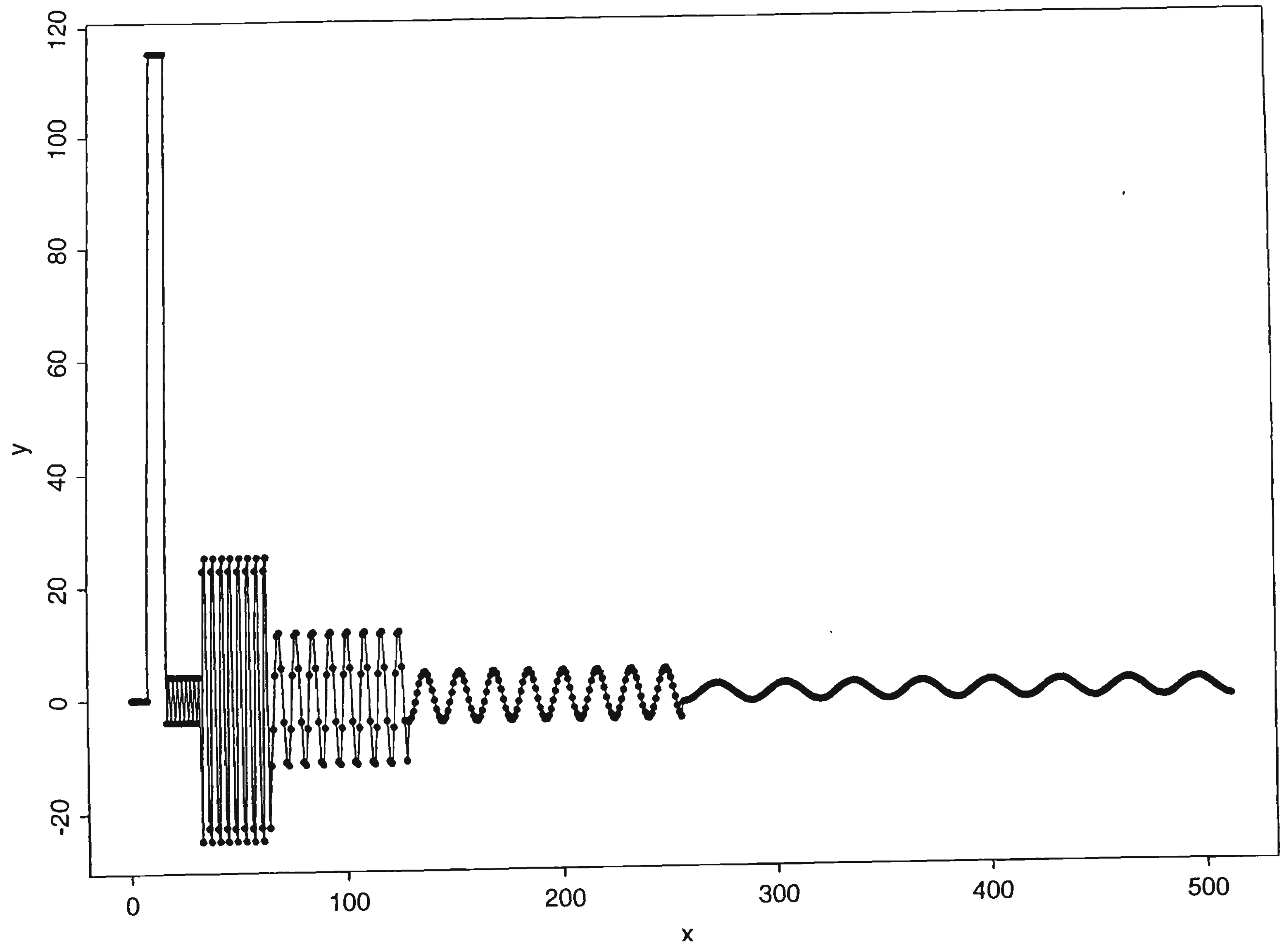


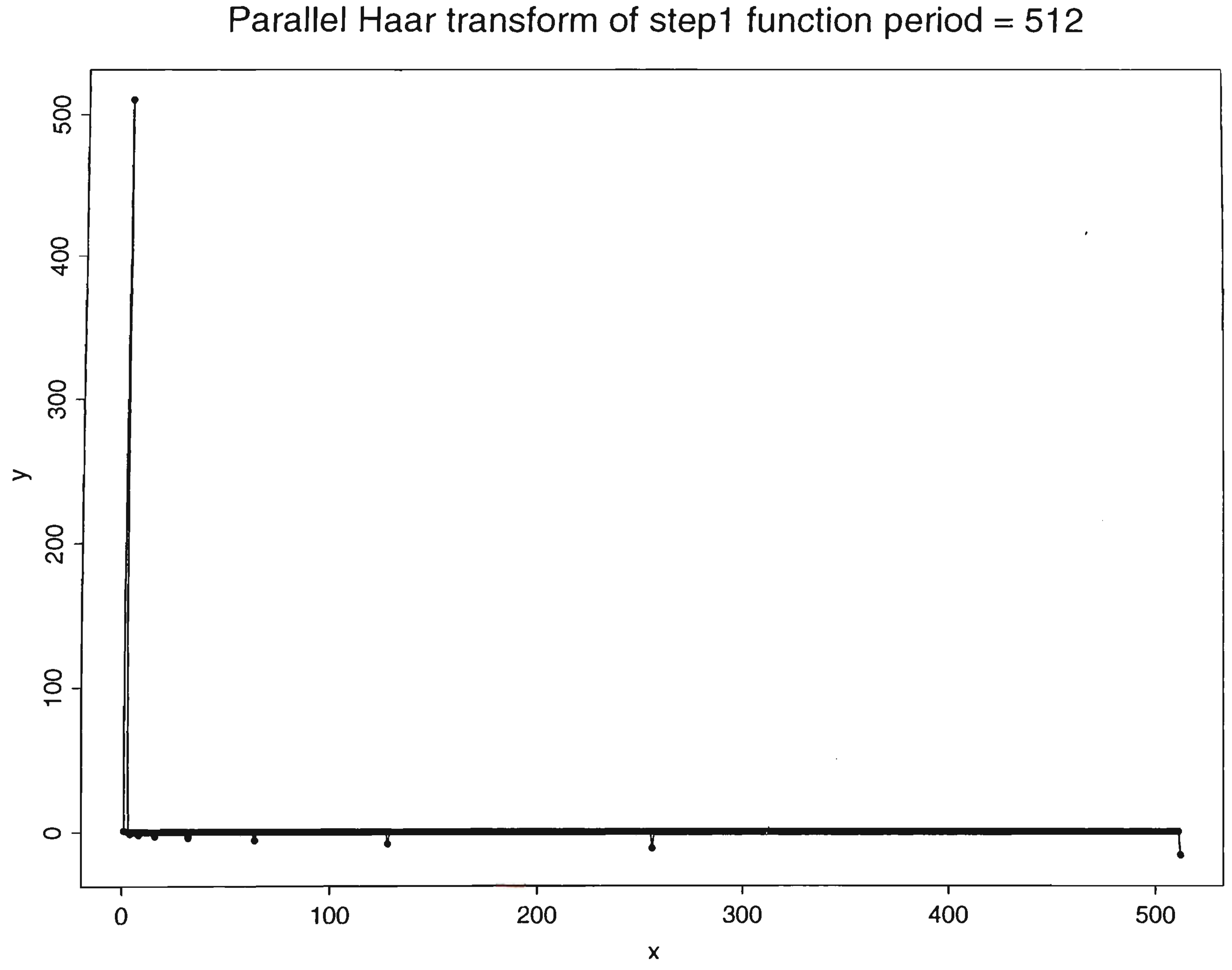

Parallel Haar transform of cos(x) period = 64
y
50
0
-50
x
0
100
200
300
400
500

Parallel Haar transform of sin(x) period = 512



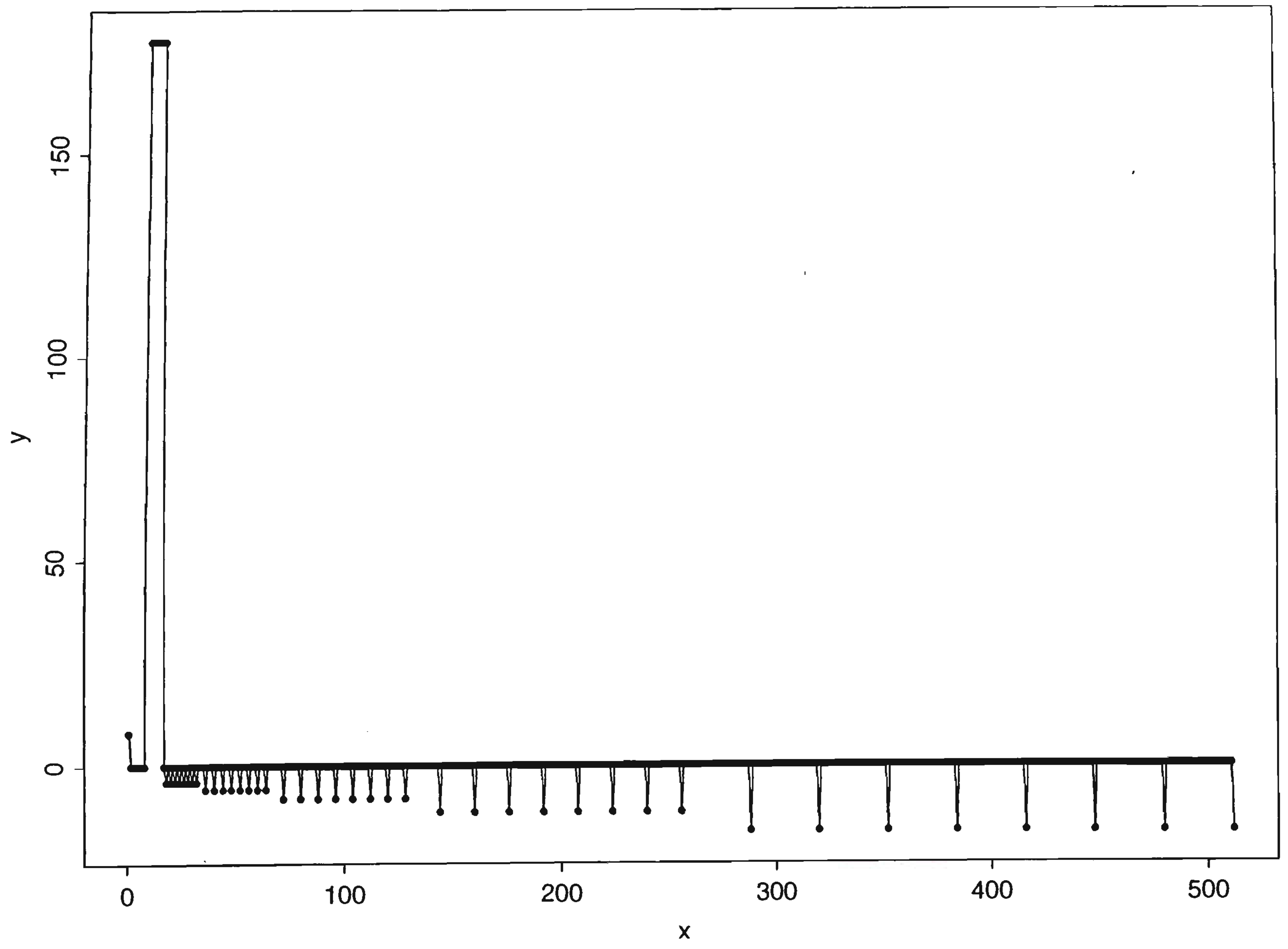
Parallel Haar transform of sin(x) period = 64

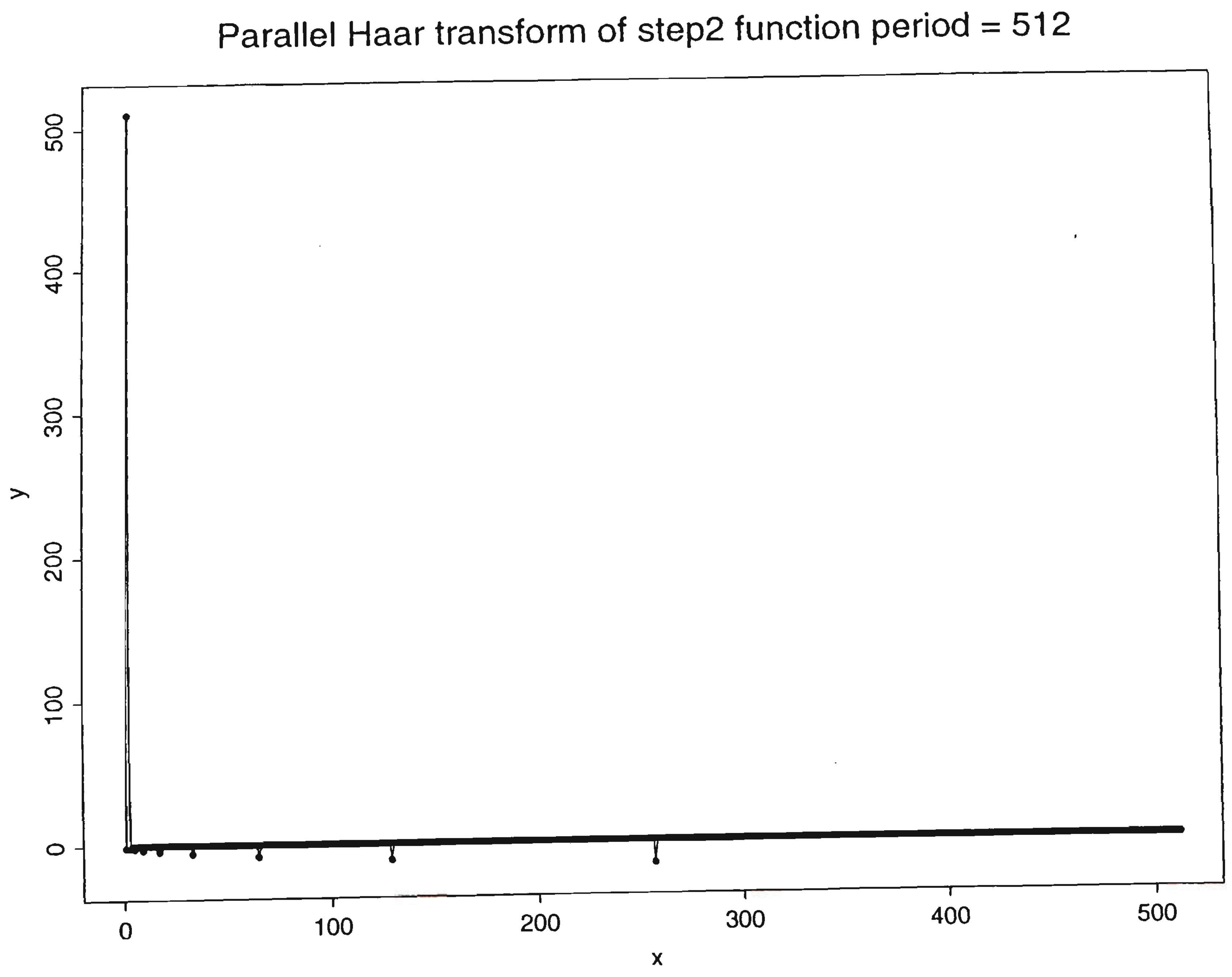




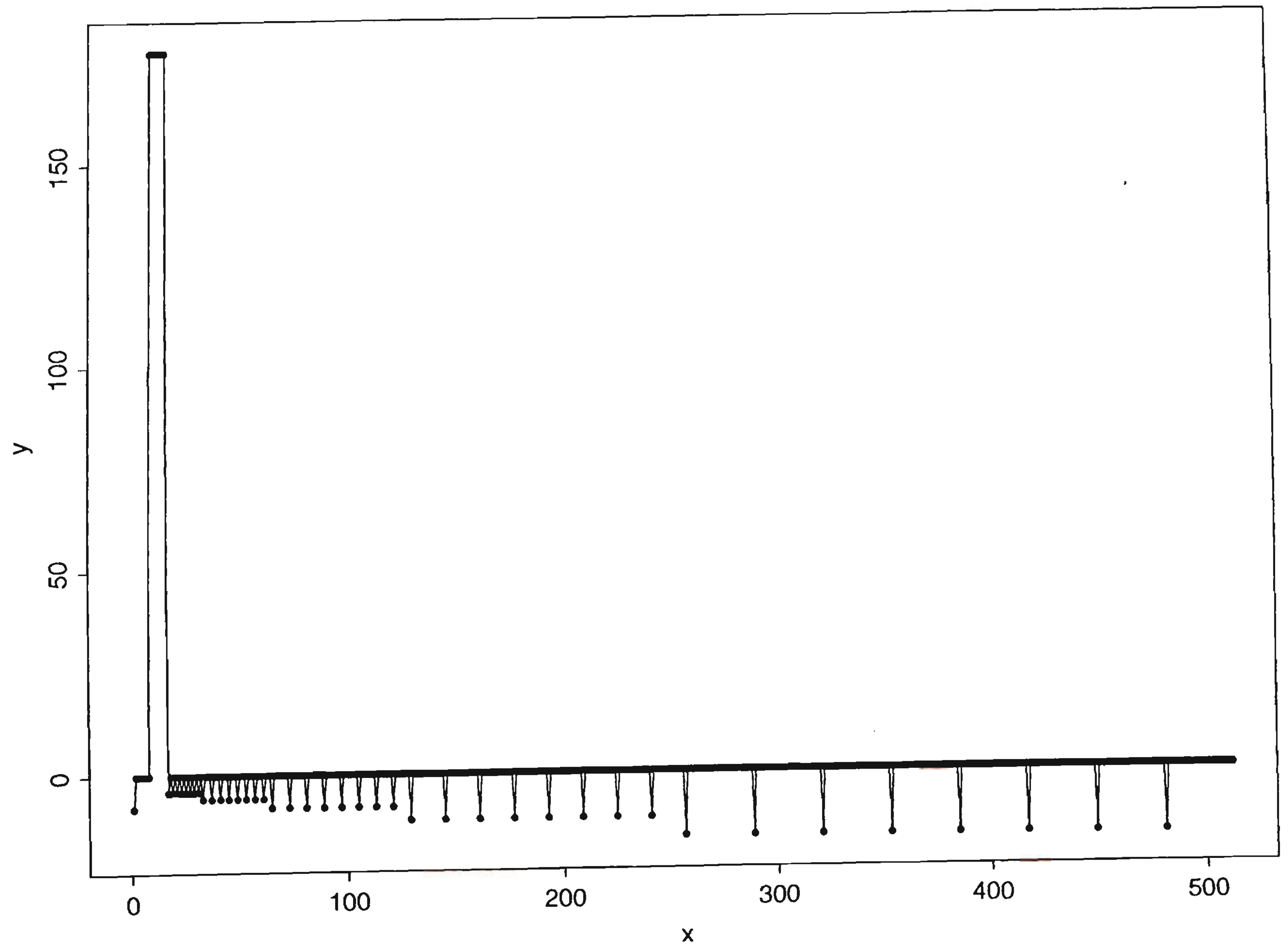
Parallel Haar transform of step1 function period = 512
y
x
0
100
200
300
400
500

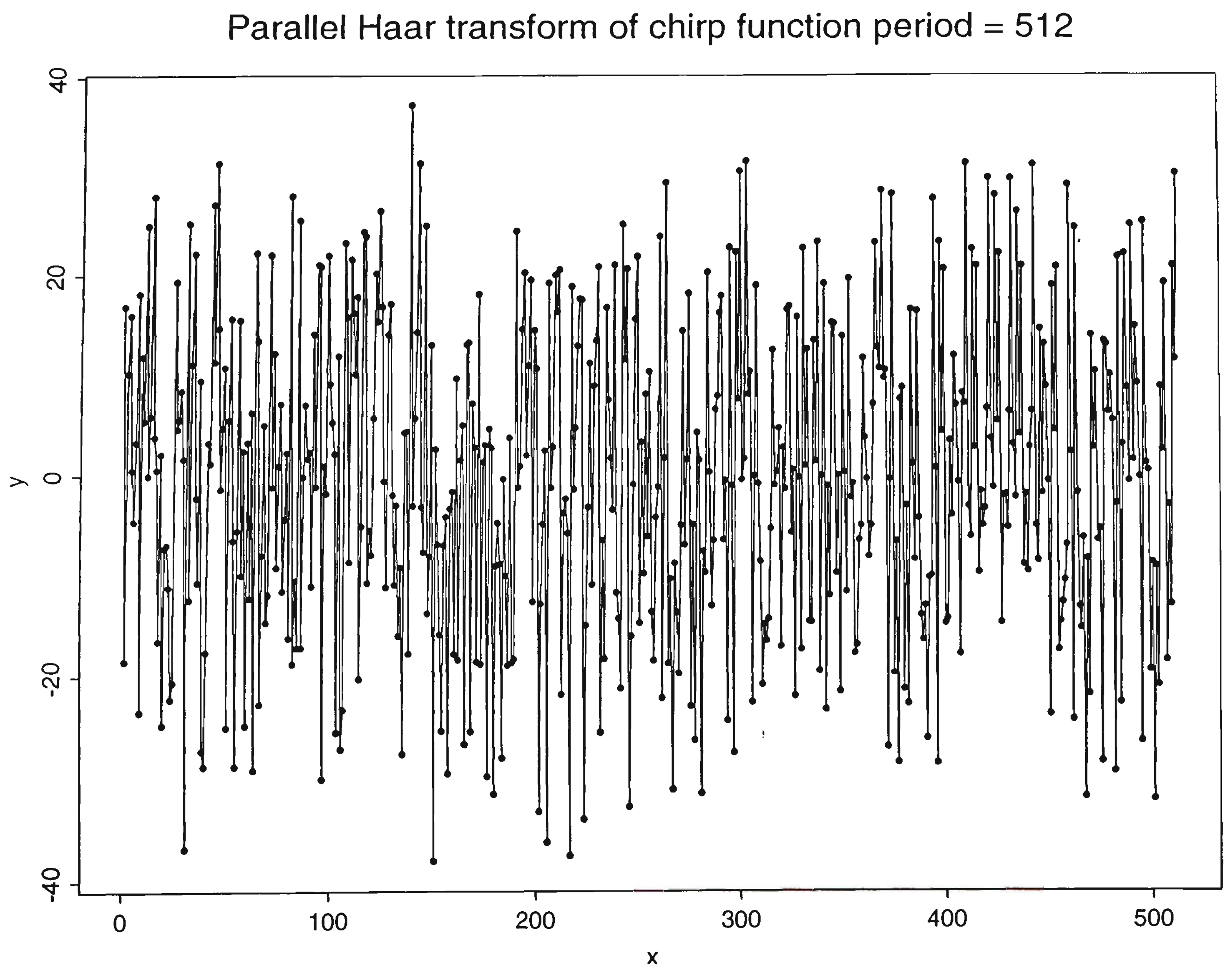
Parallel Haar transform of step1 function period = 64



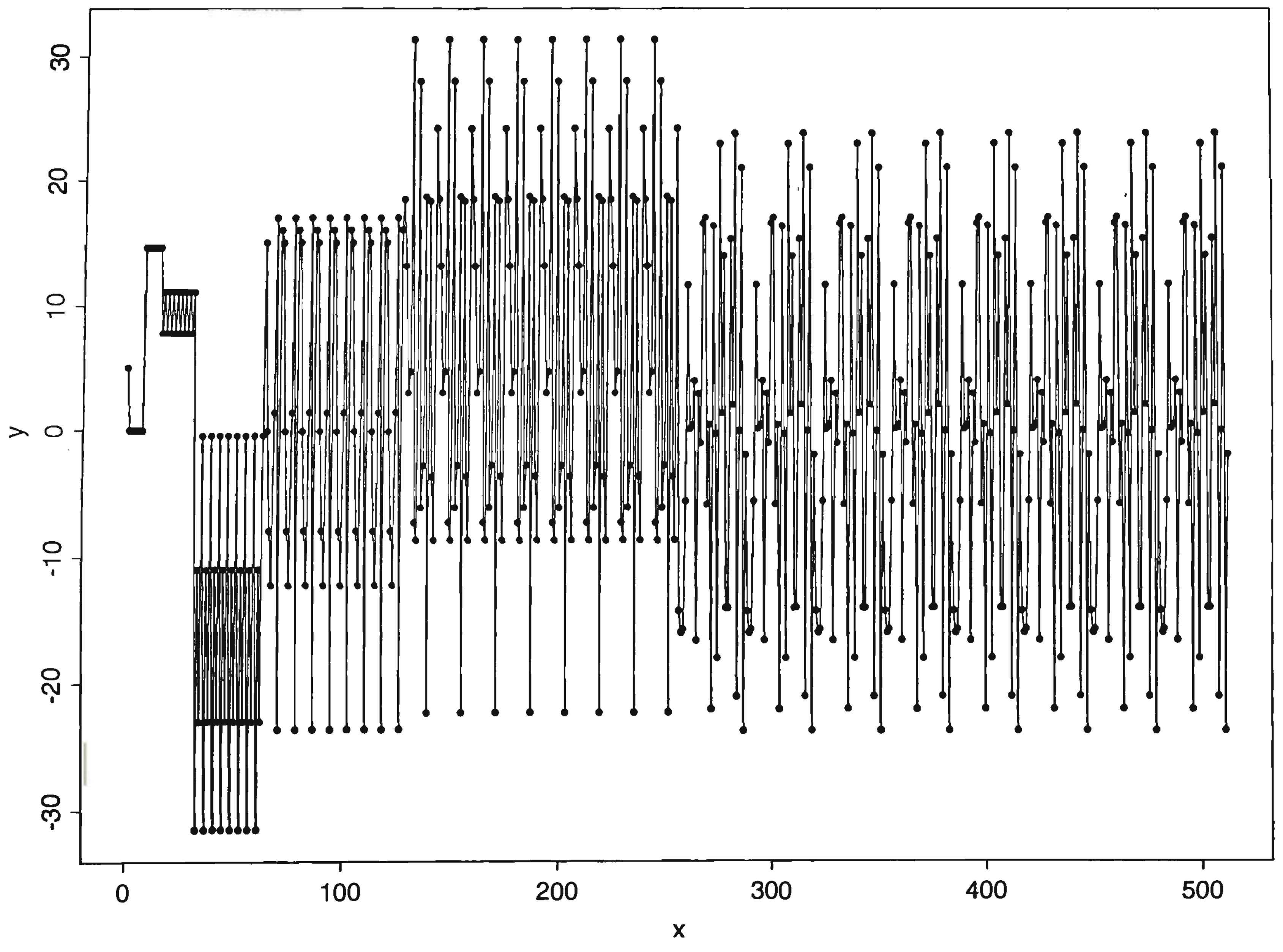

Parallel Haar transform of step2 function period = 512
y
0
100
200
300
400
500
x
0
100
200
300
400
500

Parallel Haar transform of step2 function period = 64

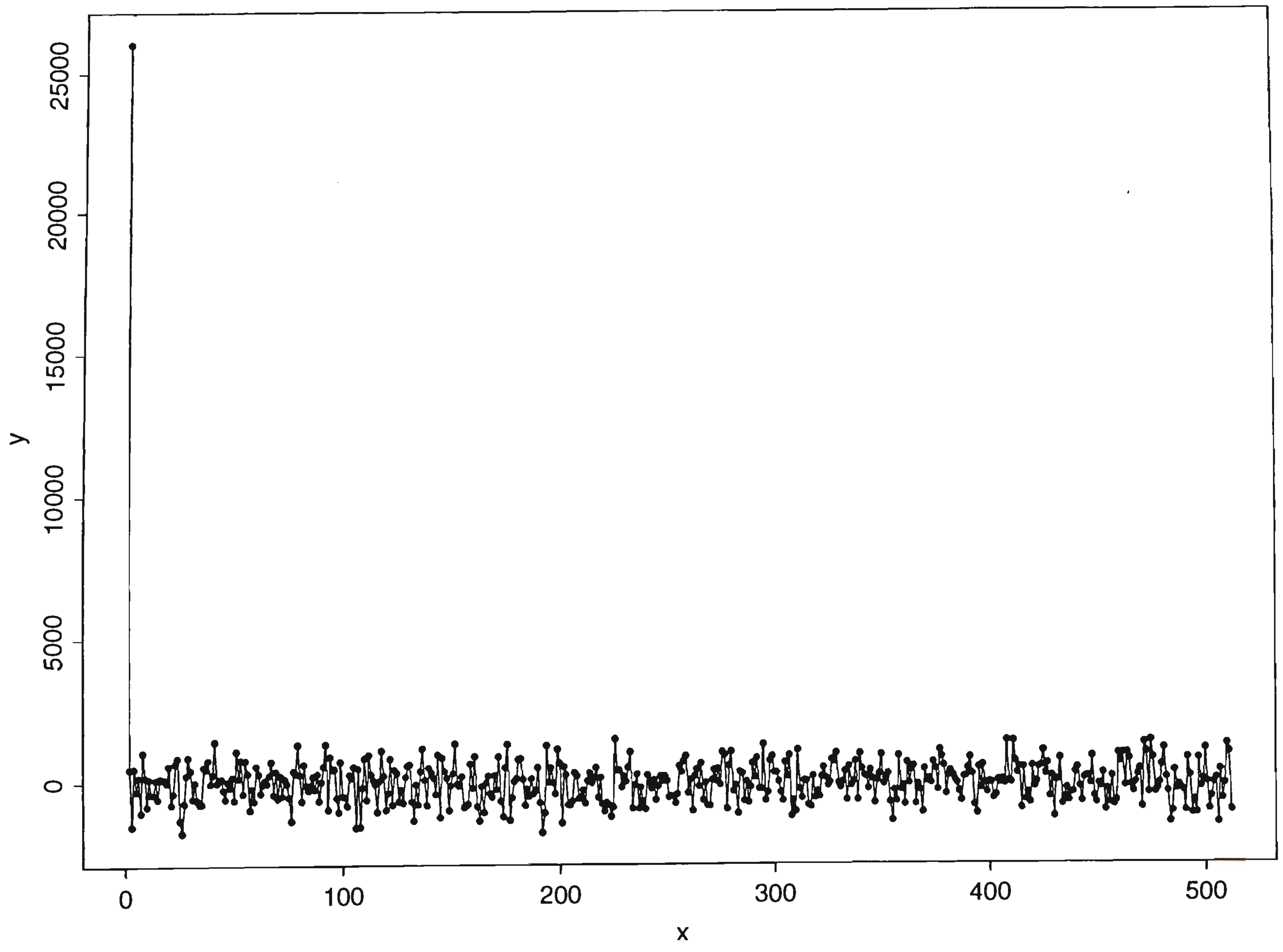



Parallel Haar transform of chirp function period = 512
y
40
20
0
-20
-40
x
0
100
200
300
400
500

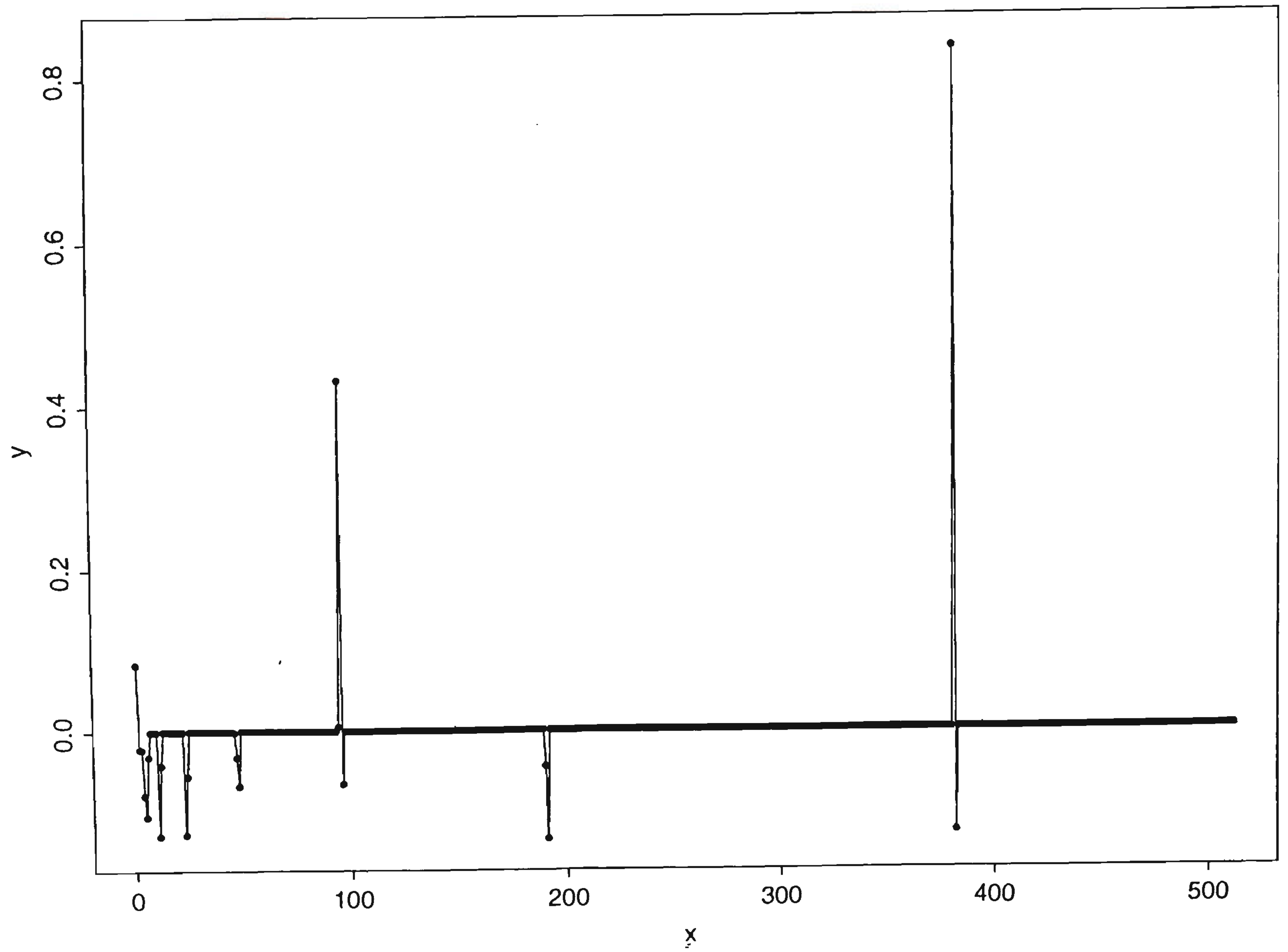
Parallel Haar transform of chirp function period = 64

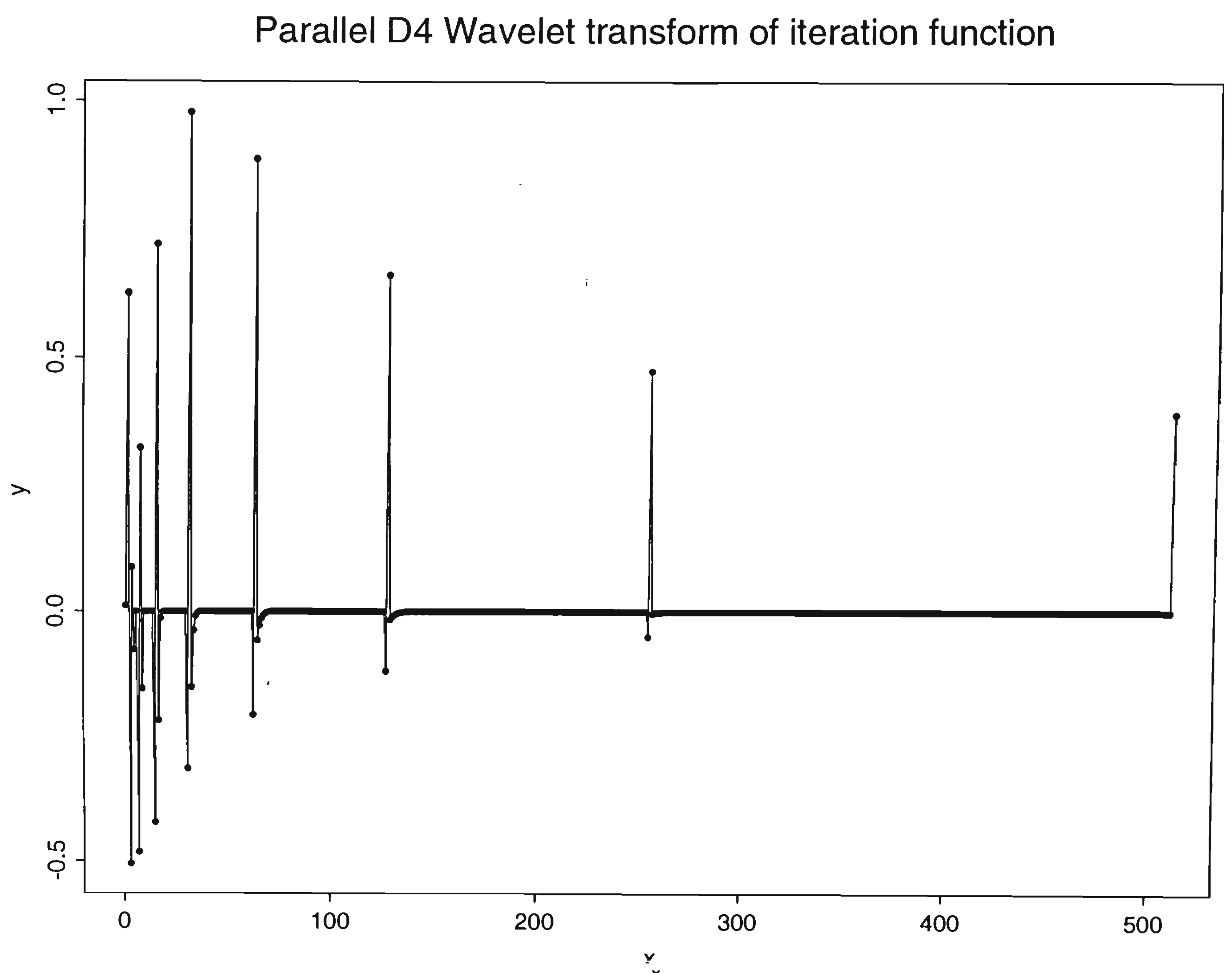


Parallel Haar transform of random data

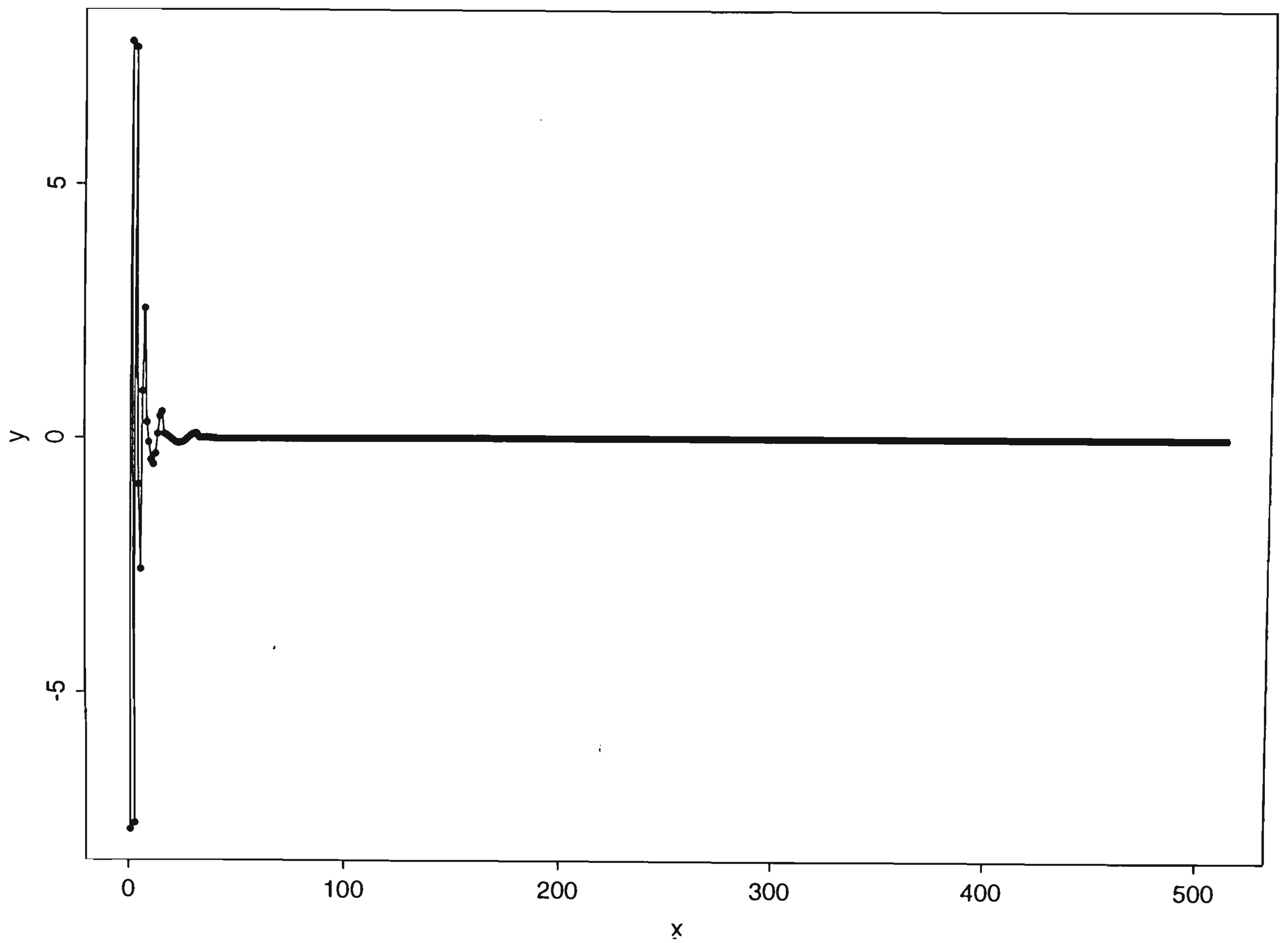


Parallel D4 Wavelet transform of delta function

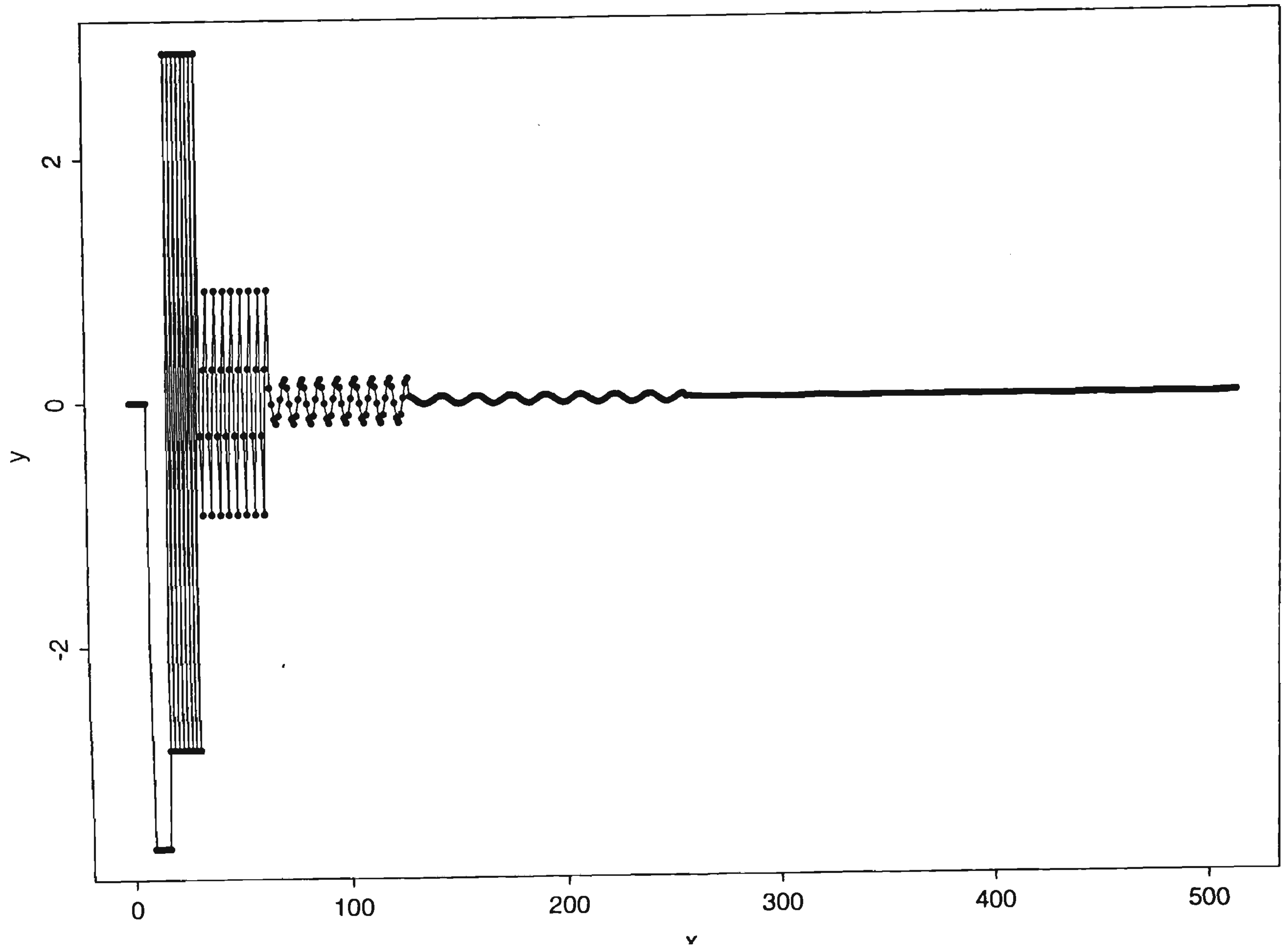



Parallel D4 Wavelet transform of iteration function
y
x
1.0
0.5
0.0
-0.5
0
100
200
300
400
500

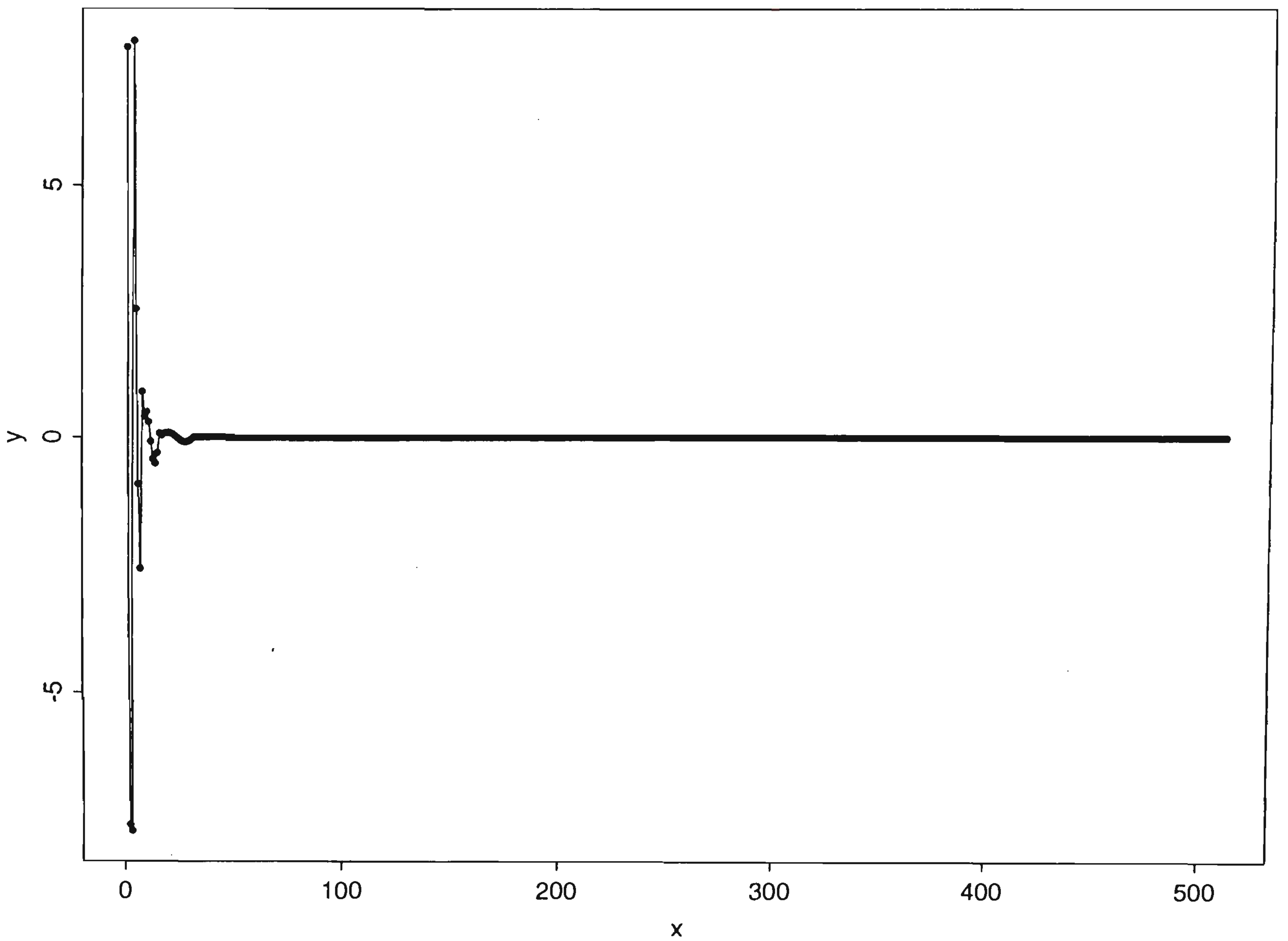
Parallel D4 Wavelet transform of cos(x) period = 512

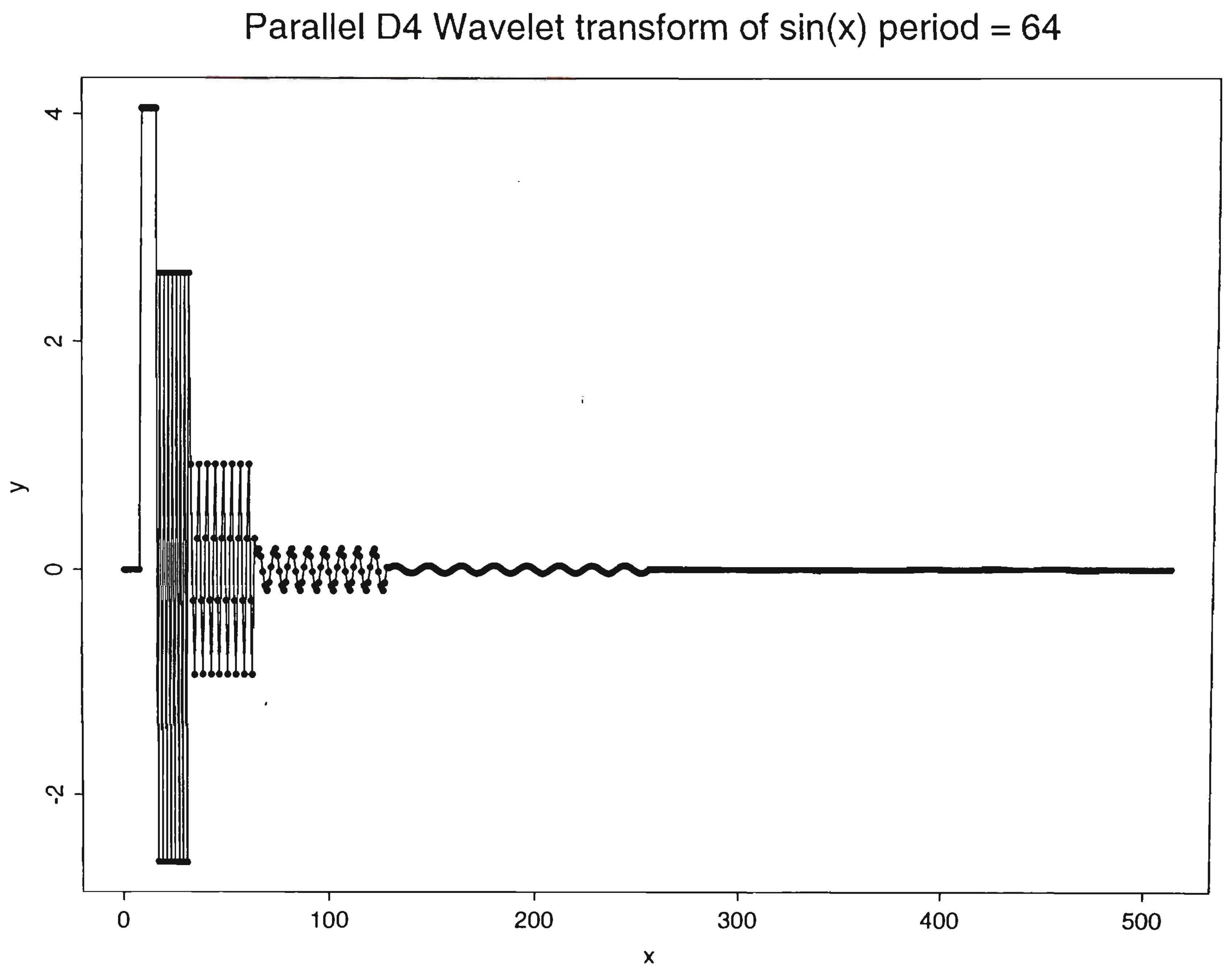


Parallel D4 Wavelet transform of cos(x) period = 64

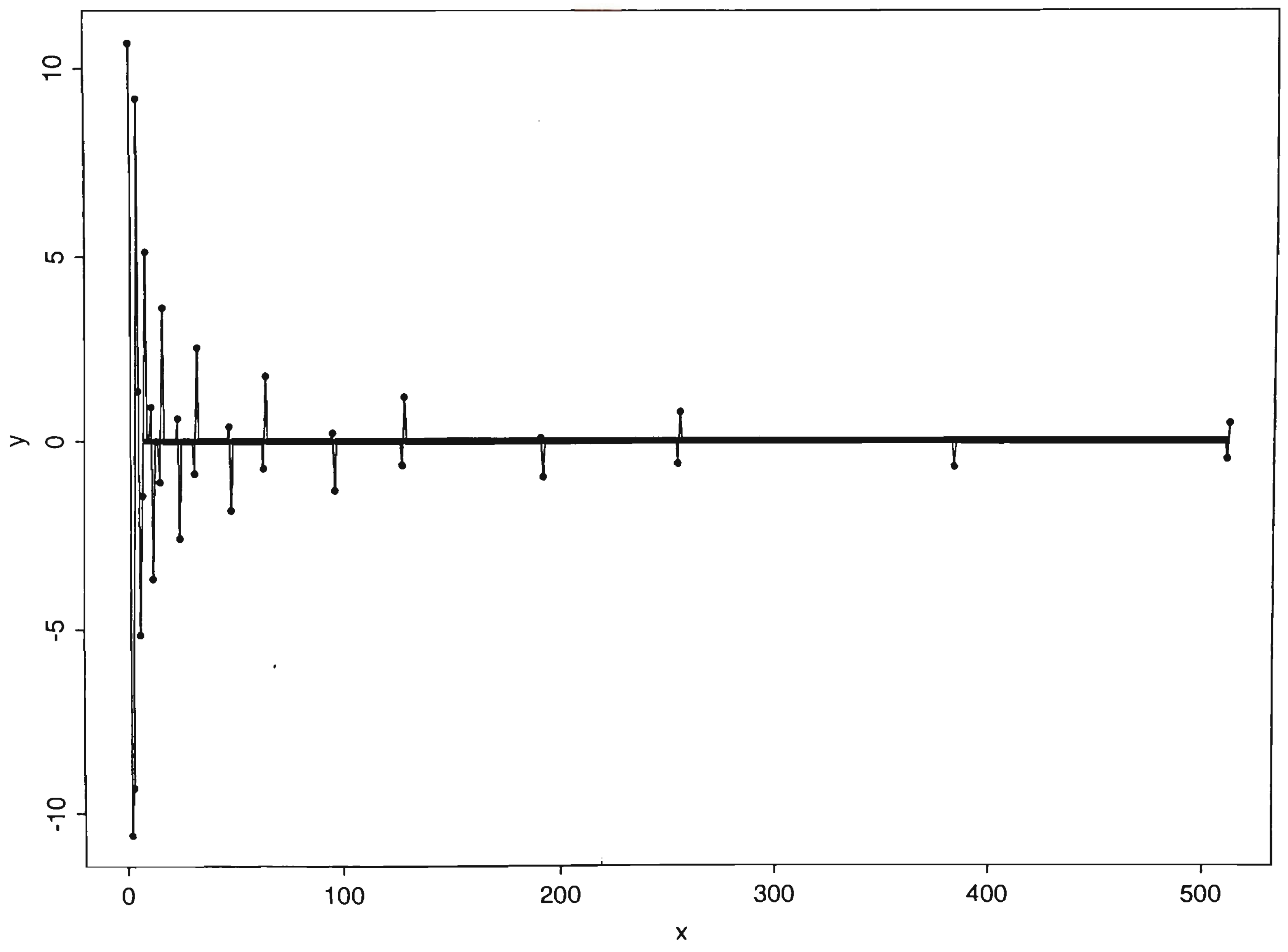


Parallel D4 Wavelet transform of sin(x) period = 512

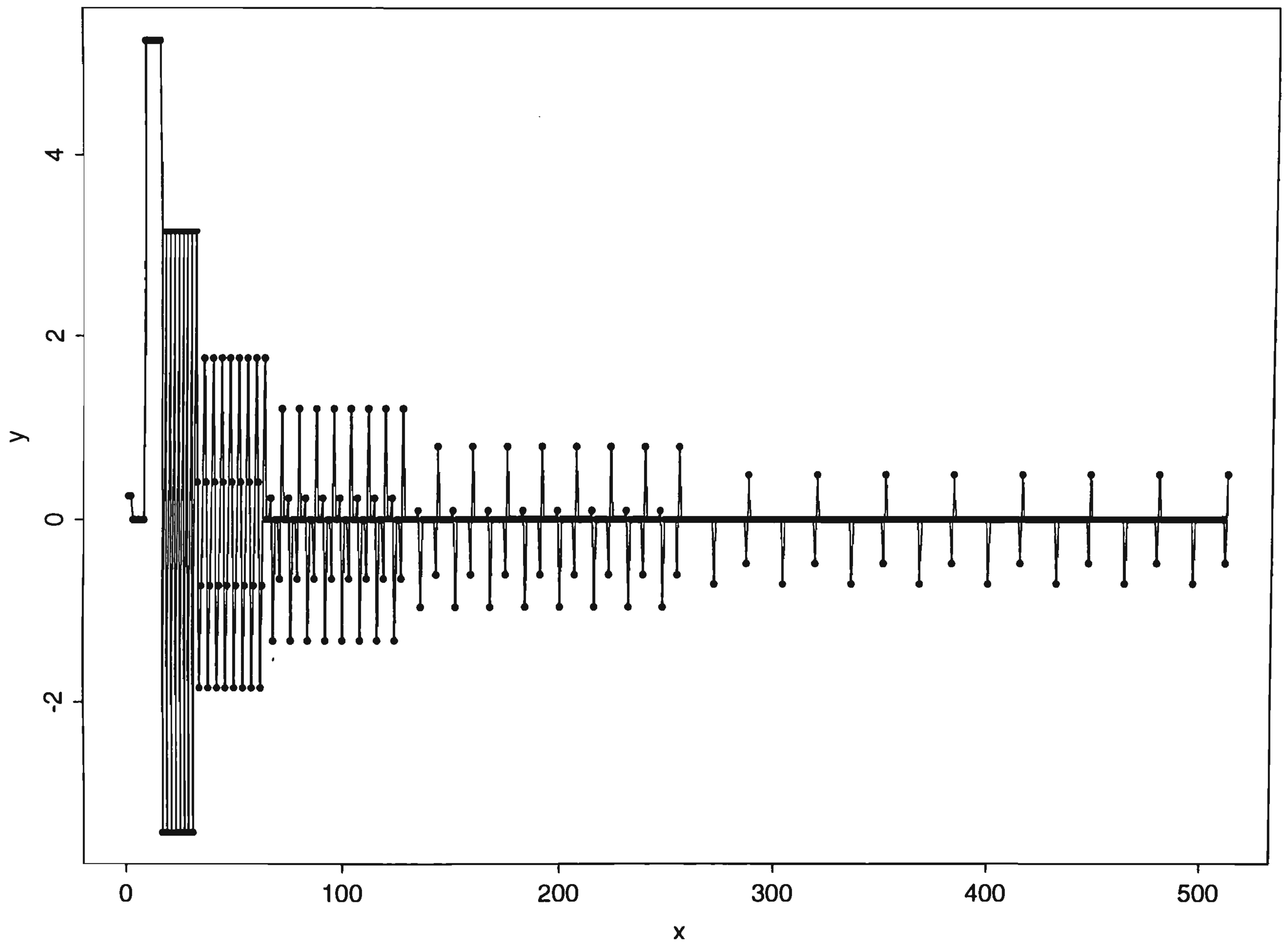



Parallel D4 Wavelet transform of sin(x) period = 64
y
4
2
0
-2
0
100
200
300
400
500
x

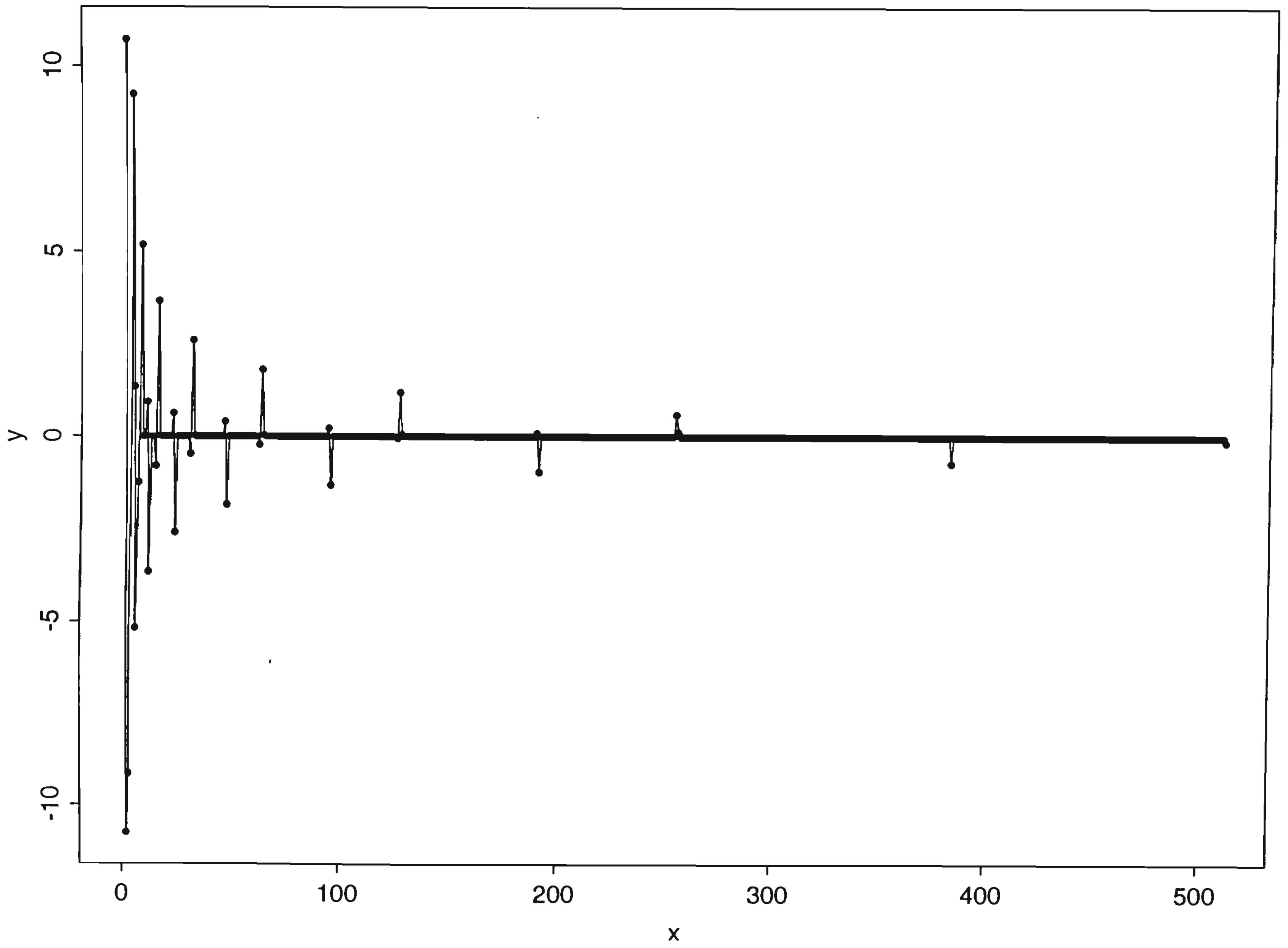
Parallel D4 Wavelet transform of step1 function period = 512



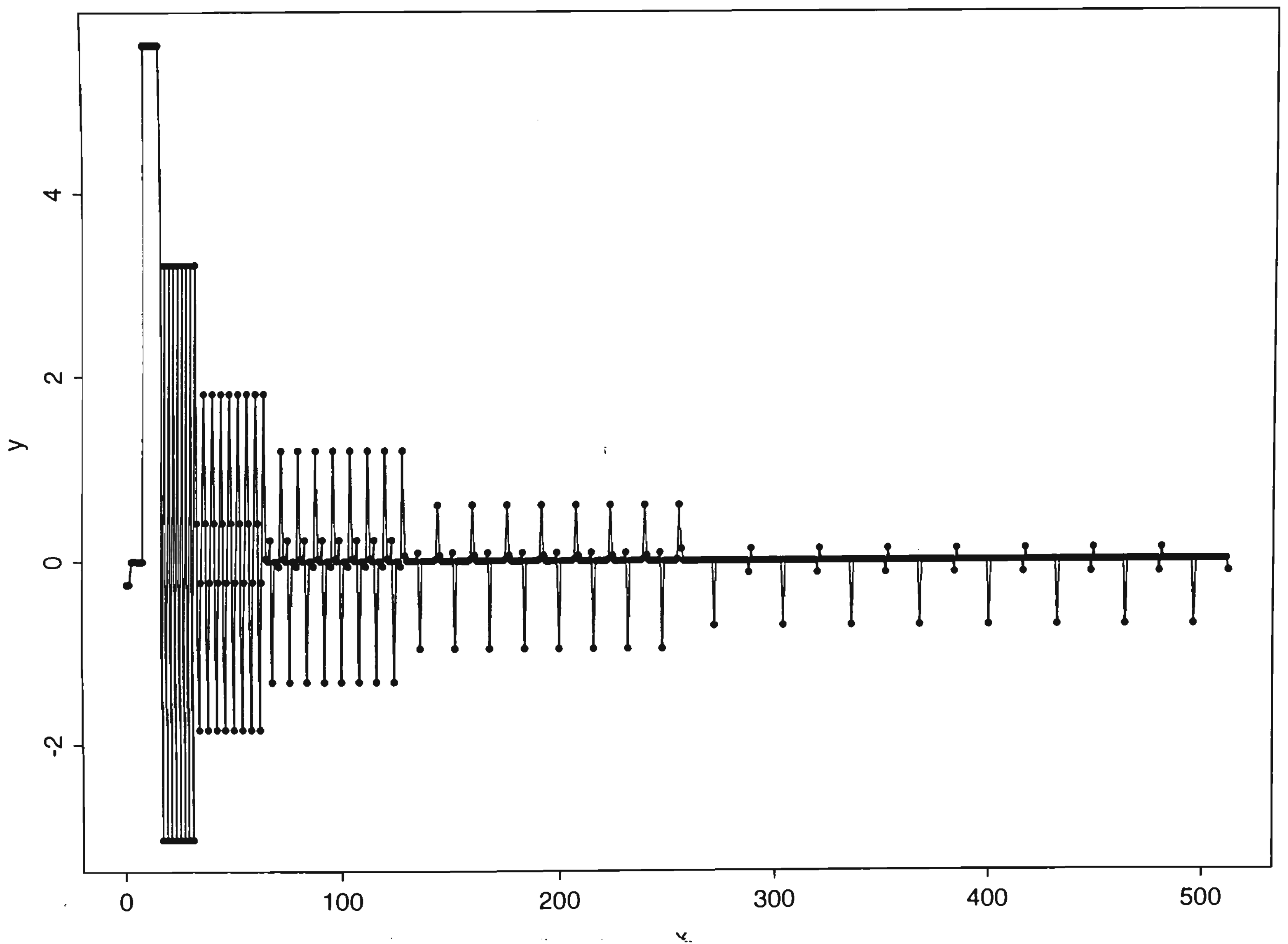
Parallel D4 Wavelet transform of step1 function period = 64



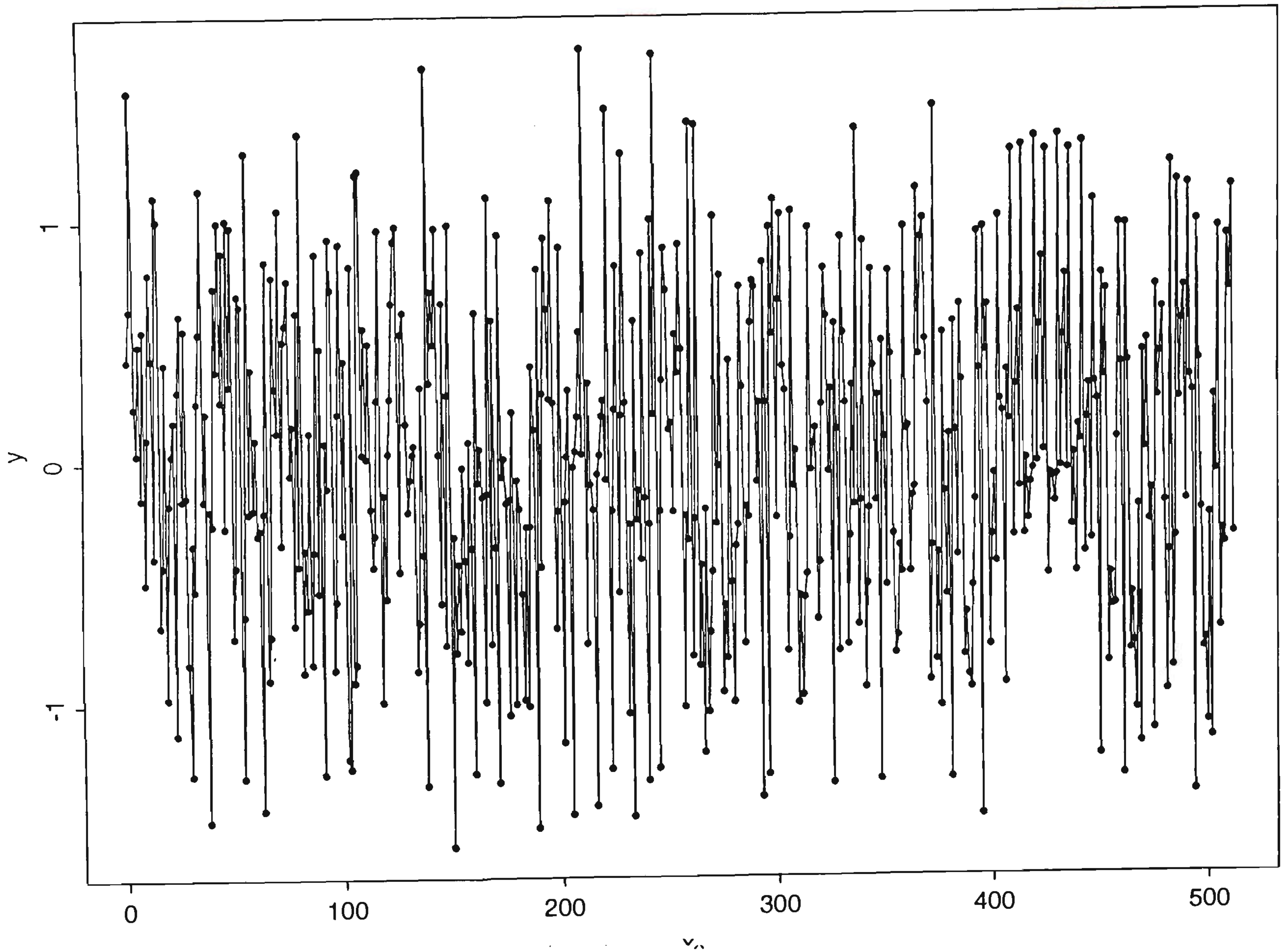
Parallel D4 Wavelet transform of step2 function period = 512



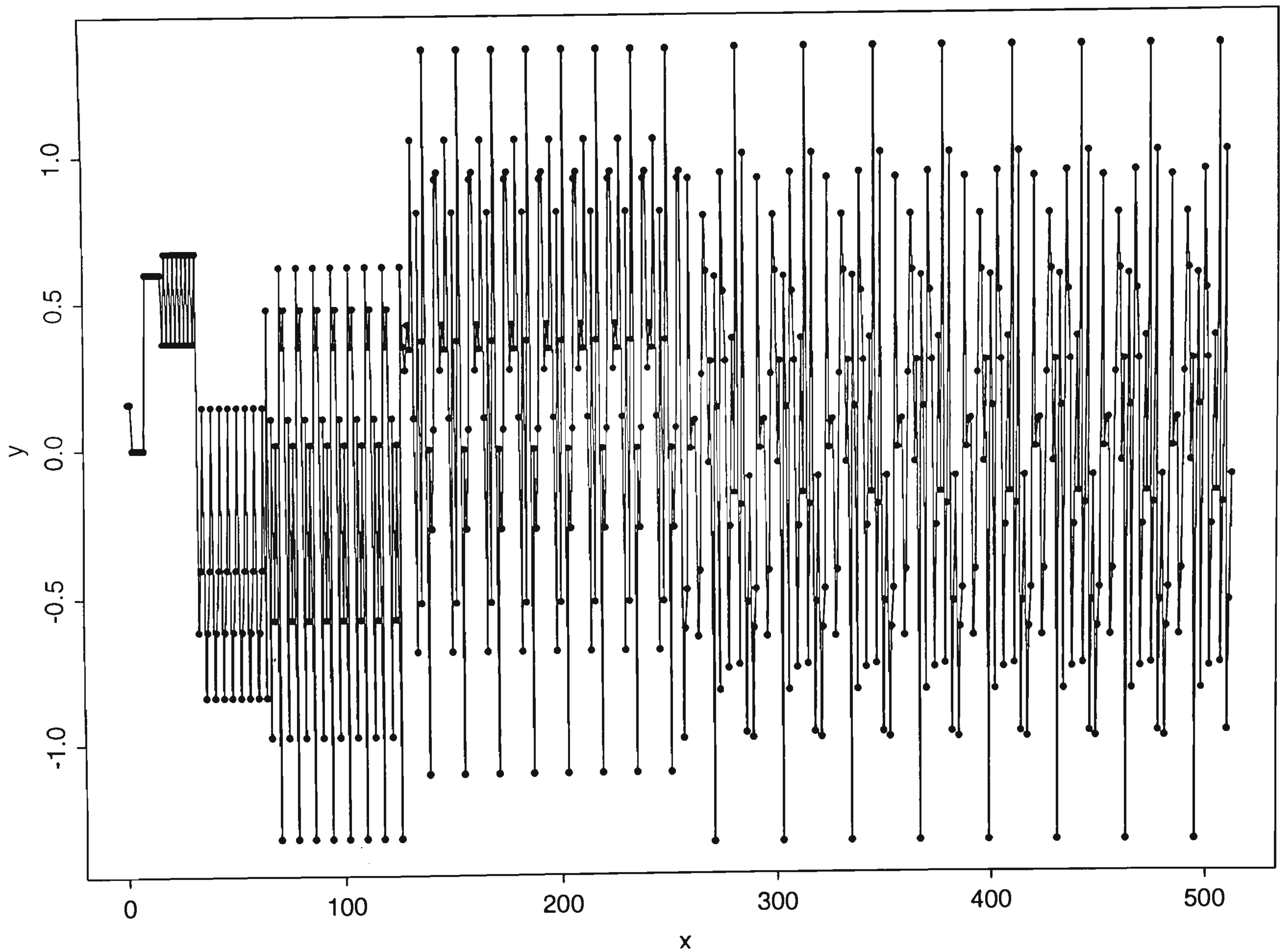
Parallel D4 Wavelet transform of step2 function period = 64



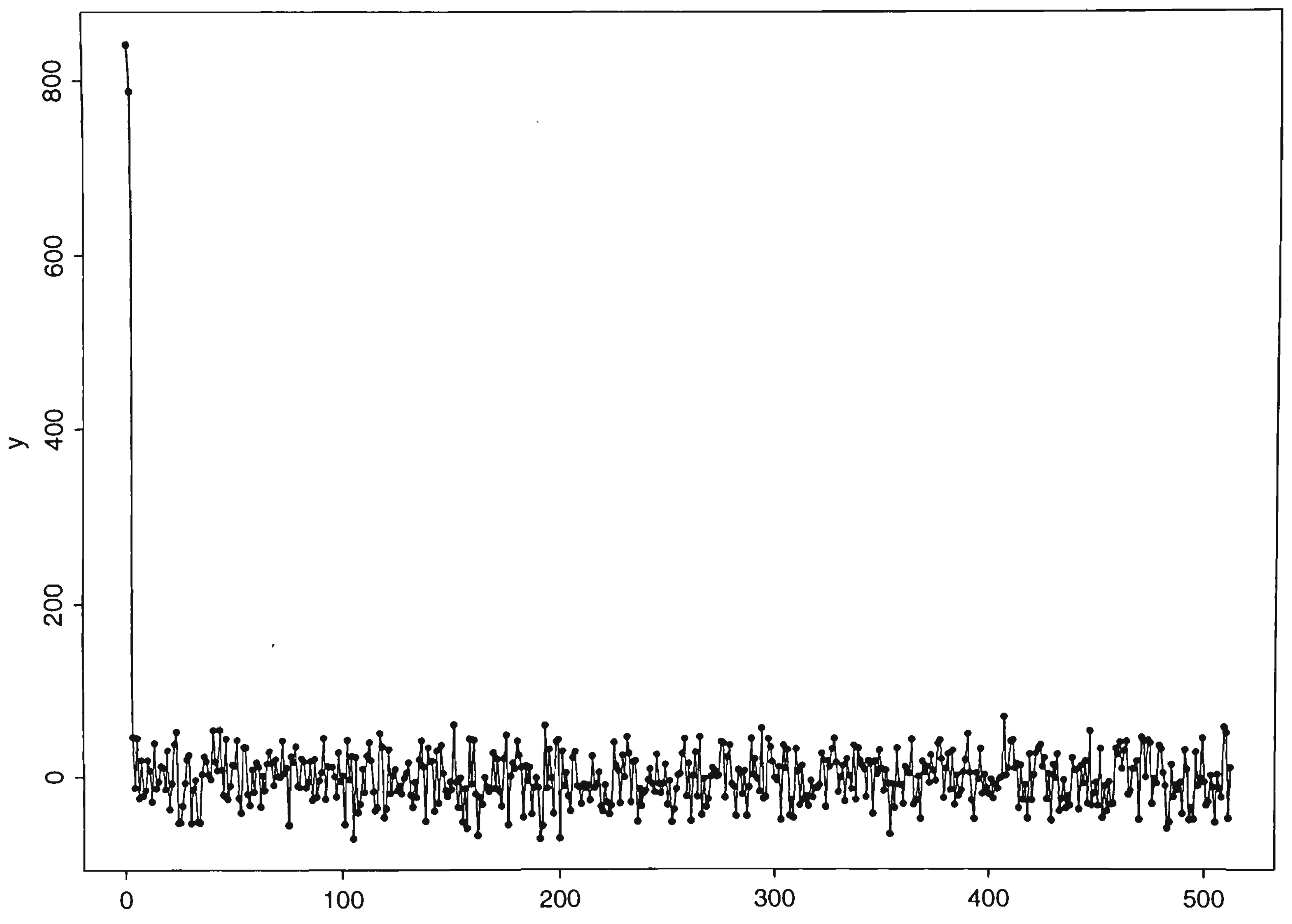
Parallel D4 Wavelet transform of chirp function period = 512



Parallel D4 Wavelet transform of chirp function period = 64



Parallel D4 Wavelet transform of random data



y
0
200
400
600
800
0
100
200
300
400
500