TRANSMISSION OF DIGITAL IMAGES
USING DATA-FLOW ARCHITECTURE

A Thesis Presented to
The Faculty of the College of Engineering and Technology
Ohio University

In Partial Fulfillment
of the Requirements for the Degree of
Master of Science

by
Nicholas Haddad
August, 1985
ACKNOWLEDGMENTS

I wish to express my sincere gratitude to Dr. Nik A. Alexandridis for his advice, guidance, invaluable comments and suggestions in preparing this thesis.

I am also thankful for my parents, my brothers and my sister for their financial and emotional support throughout my graduate studies.
## TABLE OF CONTENTS

| ACKNOWLEDGEMENTS | ........................................ | i  |
| LIST OF TABLES   | ........................................ | v  |
| LIST OF FIGURES  | ........................................ | vi |

### Chapter

1. **INTRODUCTION** ........................................ 1

2. **DATA FLOW ARCHITECTURE** .............................. 5
   2.1 Data-Flow Concept .................................... 6
   2.2 A Data-Flow Language ................................. 9
   2.3 Program Structuring .................................. 12
   2.4 The Basic Mechanism .................................. 17
   2.5 Data-Flow Language and Simulation .................. 20
      2.5.1 Data-Flow Language ............................... 20
      2.5.2 Simulation of Data-Flow Architecture (Using Concurrent Programming) ................... 21

3. **DIGITAL IMAGE PROCESSING AND TRANSMISSION** ........ 23
   3.1 Introduction ......................................... 23
   3.2 Image Digitization .................................... 25
   3.3 Image Coding ......................................... 27
      3.3.1 Transform Coding ................................ 29

4. **HIERARCHICAL IMAGE REPRESENTATION AND REGULAR DECOMPOSITION** ........................ 34
   4.1 Introduction ......................................... 34
   4.2 Quadtree Structure of an Image ....................... 35
5. DATA-FLOW CONFIGURATIONS FOR TRANSFORM-CODING DIGITAL IMAGES

5.1 Introduction ........................................... 53
5.2 General Theory of the Walsh-Hadamard Transformation ............................ 55
  5.2.1 Introduction ....................................... 55
  5.2.2 The Walsh-Hadamard Transformation .................................. 56
5.3 General Theory of the Fourier Transformation ..................................... 58
5.4 Data-Flow Configurations for Concurrent Image Transformation ................. 60
  5.4.1 Introduction ....................................... 60
  5.4.2 Merging Case ....................................... 61
  5.4.3 Non-Merging Case ................................... 77
5.5 Data-Flow Systems for Inverse Transformation .................................... 81
5.6 Comparison of Algorithms ........................................... 87
5.7 Comparison Via Number of Processors Used ....................................... 88
5.8 Data-Flow Configuration for Processing/Transmitting Hierarchically Segmented Images .................................... 95
  5.8.1 Introduction ....................................... 95
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.8.2</td>
<td>System Description</td>
<td>96</td>
</tr>
<tr>
<td>5.8.3</td>
<td>Example</td>
<td>100</td>
</tr>
<tr>
<td>5.8.4</td>
<td>Conclusion</td>
<td>103</td>
</tr>
<tr>
<td>6.</td>
<td>EXPERIMENTAL RESULTS ON IMAGES</td>
<td>104</td>
</tr>
<tr>
<td>6.1</td>
<td>Gray Level Images</td>
<td>104</td>
</tr>
<tr>
<td>6.2</td>
<td>Binary Images</td>
<td>116</td>
</tr>
<tr>
<td>7.</td>
<td>CONCLUSION</td>
<td>127</td>
</tr>
<tr>
<td>REFERENCES</td>
<td></td>
<td>130</td>
</tr>
</tbody>
</table>

**APPENDICES**

<table>
<thead>
<tr>
<th>Appendices</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Fast Transforms</td>
<td>133</td>
</tr>
<tr>
<td>B</td>
<td>Transform of Homogeneous Quadrants</td>
<td>138</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>141</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>149</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>157</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>164</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>172</td>
</tr>
<tr>
<td>H</td>
<td></td>
<td>176</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Approximate Number of Arithmetic Operations (Real or Complex) Required for</td>
<td>88</td>
</tr>
<tr>
<td></td>
<td>FastImplementation of Various Data-Flow Graphs</td>
<td></td>
</tr>
<tr>
<td>5.7.2</td>
<td>Time Taken (in Function of One Multiplication Time T) to Transform-Code an</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td>Image Using (a) 1 Processor, (b) 4 Processors, (c) 16 Processors and (d) 64</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Processors</td>
<td></td>
</tr>
<tr>
<td>5.7.8</td>
<td>Percentages Used in Graph 5.7.6(a) in Tabular Form</td>
<td>93</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Different x-y Displacements of the Object</td>
<td>119</td>
</tr>
<tr>
<td>6.2.6</td>
<td>Number of Decompositions for the Fully Symmetrical Object</td>
<td>122</td>
</tr>
<tr>
<td>6.2.7</td>
<td>Number of Decompositions for the Dominantly Symmetric Object</td>
<td>123</td>
</tr>
<tr>
<td>6.2.8</td>
<td>Number of Decompositions for the Non-Symmetrical Object in Three Sizes</td>
<td>124</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.1</td>
<td>Data-Flow Instruction States</td>
<td>7</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Data-Flow Algorithm for the Quadratic Equation</td>
<td>8</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Higher Level View of the Quadratic Equation</td>
<td>9</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Processing Elements of the Graphical Language</td>
<td>11</td>
</tr>
<tr>
<td>2.3.1</td>
<td>A Matrix Multiplier Actor</td>
<td>13</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Recursive Newton Graph</td>
<td>13</td>
</tr>
<tr>
<td>2.3.3</td>
<td>An Activity Template</td>
<td>14</td>
</tr>
<tr>
<td>2.3.4</td>
<td>(a) Interconnection of Operations and (b) Configuration of Activity Template for the Program Graph of (a)</td>
<td>15</td>
</tr>
<tr>
<td>2.3.5</td>
<td>A Conditional Schema (a) and Its Implementation (b)</td>
<td>16</td>
</tr>
<tr>
<td>2.3.6</td>
<td>An Iterative Schema (a) and Implementation (b)</td>
<td>17</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Basic Instruction Execution Mechanism</td>
<td>18</td>
</tr>
<tr>
<td>3.1.1</td>
<td>The Realm of Image Processing</td>
<td>24</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Digitizing an Analog Image to Convert it into a Rectangular Grid of Pixels. (a) The Digitized Image. (b) The 256 Gray Levels of an 8-Bit Pixel Integer</td>
<td>26</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Block Diagram of a Transform Coding System</td>
<td>31</td>
</tr>
<tr>
<td>4.2.1</td>
<td>(a) Quadtree Representation of an 8x8 Image Showing the Nodes' Numbers and the Levels. (b) Figure Showing Quadrants Accessed by Each Node Index of (a)</td>
<td>36-37</td>
</tr>
<tr>
<td>4.2.2</td>
<td>One-Dimensional Array Representing the Quadtree Structure in Memory</td>
<td>37</td>
</tr>
<tr>
<td>4.5.1</td>
<td>The Resulting Reduced Tree Representation of (a) Following the Regular Decomposition Algorithm (Up to Level 2)</td>
<td>46-47</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>4.5.2</td>
<td>The Image of Figure 4.5.1(a) Reduced Through the Regular Decomposition Algorithm (with $\theta_1 = 0.25$ and $\theta_2 = 0.45$)</td>
<td>49</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Flow Chart Describing an Efficient Software Implementation of the Regular Decomposition Algorithm</td>
<td>51</td>
</tr>
<tr>
<td>5.4.7</td>
<td>Data-Flow Configuration for the Hadamard Transformation (Merging Case)</td>
<td>64</td>
</tr>
<tr>
<td>5.4.8</td>
<td>A Proposed Data-Flow Graph that Substitutes the Mergers of Figure 5.4.7</td>
<td>67</td>
</tr>
<tr>
<td>5.4.17</td>
<td>Data-Flow Configuration for the Fourier Transformation (Merging Case)</td>
<td>71</td>
</tr>
<tr>
<td>5.4.18</td>
<td>Data-Flow Configuration for FOURIER PROCESSOR</td>
<td>74</td>
</tr>
<tr>
<td>5.4.19</td>
<td>Data-Flow Configuration for FOURIER GENERATOR</td>
<td>75</td>
</tr>
<tr>
<td>5.4.20</td>
<td>General Data-Flow Graph Applicable for Either Fourier or Hadamard Transformation (Merging Case)</td>
<td>76</td>
</tr>
<tr>
<td>5.4.21</td>
<td>Data-Flow Configuration for the Hadamard Transformation (Non-Merging Case)</td>
<td>78</td>
</tr>
<tr>
<td>5.4.22</td>
<td>Data-Flow Configuration for the Fourier Transformation (Non-Merging Case)</td>
<td>79</td>
</tr>
<tr>
<td>5.4.23</td>
<td>General Data-Flow Graph Applicable for Either Fourier or Hadamard Transformation (Non-Merging Case)</td>
<td>80</td>
</tr>
<tr>
<td>5.5.14</td>
<td>Data-Flow Configuration for the Inverse Hadamard Transformation (Merging Case). * RESULT Divides Its Inputs by $N^2$</td>
<td>84</td>
</tr>
<tr>
<td>5.5.18</td>
<td>Data-Flow Configuration for the Inverse Fourier Transformation (Merging Case). * RESULT Divides Its Inputs by $N^2$</td>
<td>86</td>
</tr>
<tr>
<td>5.7.6</td>
<td>(a) Percentage of Time Saved and (b) Percentage of Time Consumed, When Using 4, 16 or 64 Processors via l</td>
<td>91</td>
</tr>
<tr>
<td>5.7.9</td>
<td>Time via Image Size for 1, 4, 16 and 64 Processors</td>
<td>92</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.8.1</td>
<td>The Block Diagram of a System to Process/Transmit Regularly-Decomposed Transform-Coded Images</td>
<td>95</td>
</tr>
<tr>
<td>5.8.2</td>
<td>Data-Flow Configuration for Transform-Coding/Transmitting Regularly-Decomposed Images</td>
<td>97</td>
</tr>
<tr>
<td>5.8.3</td>
<td>(a) Image and (b) What Remains of It After Regular Decomposition. (c) Reduced Tree Representation of the Image in (a)</td>
<td>101-102</td>
</tr>
<tr>
<td>6.1.1</td>
<td>(a) Cross Shape. (b) What Remains After Regular Decomposition (Different Displacements of the Object)</td>
<td>105, 106, 108, 109, 111, 112, 114, 115</td>
</tr>
<tr>
<td>6.1.7</td>
<td>(a), (b), (c) Show Three (32x32) Binary Images Centered at the Middle (x=9, y=9). (d) Show the Different Displacements Considered</td>
<td>117-118</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Totally Symmetric Object (Cross Shape) of Three Different Sizes</td>
<td>121</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Dominantly Symmetric Object (T-Shape) of Three Different Sizes</td>
<td>121</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Non-Symmetric Object of Three Different Sizes</td>
<td>121</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION

Digital image processing/transmission has recently received increased attention brought about by the need to handle large quantities of imagery data, for example, in office and factory automation applications, in biomedical applications (e.g., computerized tomography), in carrying out real-time reconnaissance activities from satellites or planes, etc. The enormous amount of data needed to be processed, usually within extremely small time intervals, impose severe requirements on the system’s throughput. For example, a full color image of reasonable spacial resolution 512 x 512 requires the processing of over six million bits of information, often repeatedly. If images are to be processed by machine in real-time, the higher throughputs require the use of parallel architectures; for example, the image can be segmented and large parallel arrays of processors used, each processor operating on one parallel-sensed segment of the image. Such systems (sometimes called supersystems) exhibit large throughput capabilities.

The parallel architecture implemented in this thesis is the data-flow architecture. This architecture differs significantly from the conventional architecture. The concept of data-flow architecture eliminates the idea of instruction streams and control flow, and the concept of memory as a passive repository for program variables. It provides a means for detecting and taking advantage of opportunities
for parallel processing. The main motivation for data-flow architectures comes from the need of vastly increased amounts of computational power.

In this thesis, we will be applying the data-flow concept for transform-coding/transmitting digital images, thus improving the throughput of such systems. Two types of transformations will be considered: The Fourier transform and the Walsh-Hadamard transform. Transform-coding will be also applied to hierarchically segmented images using the data-flow concept.

This thesis is organized as follows. Chapter 2 introduces the data-flow architecture. More specifically, it discusses the data-flow concept as compared to conventional architectures. Then it explains a data-flow graphical language, program structuring and the basic mechanism of data flow. Finally, a brief discussion of data-flow languages and simulation will be given.

Chapter 3 covers digital image processing and transmission techniques. It presents the general configuration of a digital image processing system and briefly discusses the types of processing performed by each stage. After giving a short description of how analog images are scanned and converted into two-dimensional digital form, the chapter reviews image coding in general and transform coding particularly. We introduce the notion of segmenting the picture into subpictures (or blocks) of smaller size and processing those subpictures concurrently.
Chapter 4 presents a hierarchical representation of the image (the quadtree) and the regular decomposition algorithm. Since many digital images are characterized by "over-richness" of data (not all pixels convey useful information), the regular decomposition algorithm provides the reduced quadtree representation of the image by eliminating "non-interesting" regions of the picture. Finally, the chapter proposes an efficient software implementation (using PASCAL pointers) for the regular decomposition algorithm.

Chapter 5 presents different data-flow configurations for transform-coding/transmitting digital images. It starts by briefly introducing both Fourier and Walsh-Hadamard transforms. Then the image segmentation technique will be applied to concurrently transform-code the four quadrants of the segmented image. The data-flow configuration is given for both Fourier and Hadamard transformations. For each transformation, two cases are considered, namely the merging and non-merging cases. A comparison of those data-flow systems (in terms of computations required) is given. The inverse transformation (Fourier or Hadamard) will be briefly discussed in the chapter. Next, we discuss the trade-off between having 4, 16 or 64 processors concurrently transform-coding the segmented image. Finally, the regular decomposition algorithm is combined with the data-flow systems described above in order to concurrently decompose the image (i.e., select its informative areas), transform-code informative quadrants and asynchronously transmit those quadrants.
Chapter 6 has some experimental results related to the regular decomposition algorithm. Different objects with different sizes will be considered.

Chapter 7 contains the conclusion. Appendix A includes detailed steps in the derivation of the fast Fourier and Hadamard transformations. Appendix B includes steps in the derivation of transforms of homogeneous areas. All other appendices include computer programs implementing the developed algorithms.
Chapter 2

DATA FLOW ARCHITECTURE

Introduction [1]

Conventional computer architecture generally falls into three configurations: pipeline computers, array processors and multi-processors. At every stage in their development, large computer architecture has revealed the efforts of their designers to achieve the maximum possible processing speed attainable.

One of the features of current architectural research is the development of complex parallel systems that are equipped with either adaptable or data flow architectures. The first is capable of adapting to the specificities of programs via software, whereas the second can exploit the maximum parallelism present in programs.

The proliferation of both types of architectures was encouraged by advances in LSI and VLSI technology that significantly reduce the cost and enhance the reliability of computer components. Recently, many attempts have been made to develop VLSI devices for signal/image processing. For this reason, it is now feasible to design reliable modular computer systems containing more hardware resources than ever before, resulting in a sharp increase in the number of instruction and data streams that may be computed by a parallel system.

Systems with such exceptional throughputs have been named "super-systems". The second generation of these systems began with the
announcement of the CRAY-1 in 1975, swiftly followed by the CDC Cyber 203 in 1976, the Burroughs BSP in 1977 and the Cyber 205 in 1980.

In this chapter we will be concerned with data flow architecture. The architecture deals with the possibility of implementing the data flow principles inherent in modern program structure by allowing each instruction to be executed as soon as its operands arrive. It is also capable of realizing computational concurrencies present in the program since each time it may execute as many instructions in parallel as are ready for computation—i.e., they receive both operands needed for computation. This is an extremely important factor in computing complex real-time algorithms with time restrictions. A data flow architecture may minimize the time required for the execution of such an algorithm by assigning all its concurrent instructions to separate resources.

More specifically, we will be discussing the data flow concept and language, its program structuring, its basic mechanism, and finally we will discuss data flow languages and its simulation using concurrent programming (PASCAL PLUS).

2.1 Data Flow Concept [2]

The most striking aspect of data flow architecture is its elimination of all the fundamental properties of conventional programming languages and machine architecture. In a data flow architecture there is no concept of variables, rather data values move from one instruction to another as a program executes.
Second, there is no concept of control flow, instruction counters or branching. Instead, instructions/statements are data driven. An instruction is considered to be enabled when a datum exists on its input port and no datum exists on its output port. When an instruction is executed, the data on its input ports disappear and a result appears on its output port. Programs are represented by connecting instructions in a directed graph. Thus the order of instruction execution is controlled not by an instruction counter, but by the flow of data among instructions.

These principles of instruction execution are illustrated in Figure 2.1.1. Here large circles represent arcs or ports between instructions, and black circles represent data. The first three instructions illustrate those in the disabled state. The first has no input data available, the second does not possess data on all its input

Figure 2.1.1. Data-Flow Instruction States
arcs, and the third has a datum on its output arc. The fourth instruction achieves the properties needed for enabling. When the instruction is executed, its inputs disappear from the input arcs, and a result is created on its output arc.

Figure 2.1.2 is a data flow representation of the quadratic equation. Here we must introduce two additional concepts. One is a link, which is an operation with one input arc and multiple outputs. It is enabled when its input arc has a datum and its output arcs are empty; its function is to distribute the input datum onto all its output arcs. The link is represented by a small black circle. The second addition is an endless source of constant values to an instruction. For instance, after the multiplication operation 2 and a "fires" (is executed), the constant 2 is immediately regenerated on one of the input arcs.

Figure 2.1.2. Data-Flow Algorithm for the Quadratic Equation
One should observe that, at arbitrary points in time, multiple operations are enabled and thus could be performed concurrently. The diagram illustrates the principle that instructions do not "ask" for their operands in the traditional manner; but specify, by the connection of arcs, to which instructions their output is directed.

One can visualize a data-flow graph as a higher level entity, such as is done in Figure 2.1.3.

![Diagram](image)

**Figure 2.1.3. Higher Level View of the Quadratic Equation**

### 2.2 A Data Flow Language [2]

As mentioned above, a data-flow language has no concept of variables and control flow; the expression of a program is a set of data-activated operators connected by unidirectional data paths. There are three major concepts:

1. Processing elements. A processing element is an operation that is enabled by the arrival of information on its input arcs and the absence of information on its output arc. The two categories of processing elements are actors and links. An actor is an operation with one output arc and one or more input arcs. A link is an operation with one input arc and multiple output arcs.
2. Information. Information exists in the form of tokens which are transmitted over arcs and consumed and created by processing elements. The two basic types of information are data values and control or boolean values. The language does not currently distinguish types of data values. Extensions to the language include support for data structures.

3. Arcs. An arc is an unidirectional path for information from one processing unit to another. An arc can be empty or, at most, contain a single token of information. As such the arc is the replacement for the traditional concepts of variables and storage.

Figure 2.2.1 illustrates the processing elements of the language. Solid arrows represent connections for data arcs; dashed arrows represent connections for control arcs, a link operation is enabled when a token appears on its single input arc, and all its output arcs are empty. It distributes the input token to the output arcs. An operator actor normally has one or two input arcs. It is enabled for execution when data tokens are present on all input arcs and its output arc is empty. It absorbs the input tokens, performs some function across these values, and places a result data token on its output arc.

The decider actor is similar, except that it produces a boolean result. It computes some predicate across the data inputs, which produces a boolean value as a result.

The remaining three actors have both data and control inputs. The T-gate (true-gate) actor is enabled when it has both a data token and a control token available as input (and as always its output arc is
empty). Like all other elements, it absorbs its inputs when executed. If the control value is true, the data input is placed on the output arc. If the control value is false, no output is produced. The F-gate actor is similar, but a false, rather than a true, control value causes the data token to be passed to the output arc.

Fig. 2.2.1. Processing Elements of the Graphical Language

The merge actor is an exception in that its execution does not cause all tokens on its input arcs to be destroyed, and it does not require all input tokens to be present to become enabled. It is enabled (1) when a true control token is present and a data token is present on the data arc labeled T, or (2) when a false control token is present and a data token is present on the data arc labeled F, and in both cases, the output arc is empty. If the control input is true, the
A data token on the T-input arc is transmitted to the output arc. These two input tokens are destroyed, but a token, if present, on the F-input arc remains there. If the control input is false, the data token on the F-input arc is transmitted to the output arc. These two input tokens are destroyed, but a token, if present, on the T-input arc remains there.

2.3 Program Structuring [3, 4]

It is cumbersome to deal with graphical programs consisting of single very large graphs. Just as subroutines and procedures are used to structure conventional programs, macrofunctions can be used to structure graphical ones. This replacement is called macroexpansion, in analogy to the similar concept used in conventional languages. It is valid to view data flow languages as performing macroexpansion during execution rather than during compilation.

Macrofunctions often aid in understanding and developing graphical programs. For example, one might wish to encapsulate a data flow subgraph and think of it as one actor. For instance, a flow graph that performs matrix multiplication is now thought of as an actor having two matrices to be multiplied as shown in Figure 2.3.1.

Recursion is easy to visualize in data flow graphs. As mentioned above, a node labeled with a macrofunction can be thought of as replaced by its consequent. This rule can be adopted for recursively specified functions, such as those in which a series of macroexpansions from a node labeled G can result in a subgraph containing a node labeled G (Figure 2.3.2).
Figure 2.3.1. A Matrix Multiplier Actor

Figure 2.3.2. Recursive Newton Graph
Another representation of data flow programs— one much closer to the machine language used in data flow computers—is useful in understanding the working of these machines. In this scheme, a data flow program is a collection of activity templates, each corresponding to one or more actors of a data flow program graph. An activity template corresponding to the ADD operator is shown in Figure 2.3.3. There are four fields: an operation code specifying the operation to be performed; two receivers, which are places waiting to be filled in with operand values; and destination fields, which specify what is to be done with the result of the operation on the operands.

![Figure 2.3.3. An Activity Template](image)

An instruction of a data flow program is the fixed portion of an activity template. It consists of the operation code and the destinations, that is,

```
<opcode, destinations>
```

Figure 2.3.4.b shows how activity templates are joined to represent a program graph, specifically the composition of operands in Figure 2.3.4.a. Each destination field specifies a target receiver by giving
the address of some activity template and an input integer specifying which receiver the template is the target, that is,

destination:

<address, input>

Figure 2.3.4. (a) Interconnection of Operators and
(b) Configuration of Activity Template for the Program Graph
of (a).

Program structures for conditionals and iteration are illustrated in Figures 2.3.5 and 2.3.6. These use two new data flow actors, switch and merge, which control the routing of data values. The switch actor sends a data input to its T- or F-output to match a true or false boolean control input. The merge actor forwards a data value from its T- or F-input according to its boolean input value. The conditional
Figure 2.3.5. A Conditional Schema (a) and Its Implementation (b)

program graph and implementation in Figure 2.3.5 represent computation of

\[ y = (\text{IF } x > 3 \text{ THEN } x + 2 \text{ ELSE } x - 1) \times 4 \]

and the program graph and implementation in Figure 2.3.6 represent the iterative computation

\[ \text{WHILE } x > 0 \text{ DO } x = x - 3 \]

Execution of a machine program consisting of activity templates is viewed as follows. The contents of a template activated by the presence of an operand value in each receiver takes the form

operation packet:

\(<\text{opcode}, \text{operands}, \text{destinations}>\)
Such a packet specifies one result packet of the form

\[
\text{result packet:} \quad \langle \text{value, destination} \rangle
\]

for each destination field of the template. Generation of a result packet in turn causes the value to be placed in the receiver designated by its destination field.

2.4 The Basic Mechanism [4]

The basic instruction execution mechanism used in several current data flow projects is illustrated in Figure 2.4.1. The data flow program describing the computation to be performed is held as a collection of activity templates in the activity store. Each activity template has a unique address which is entered in the instruction queue.
unit (a FIFO buffer store) when the instruction is ready for execution. The fetch unit takes an instruction address from the instruction queue and reads the activity template from the activity store, forms it into an operation packet, and passes it on to the operation unit. The operation unit performs the operation specified by the operation code on the operation values, generating one result packet for each destination field of the operation packet. The update unit receives result packets and enters the values they carry into operand fields of activity templates as specified by their destination fields. The update unit also tests whether all operand and acknowledge packets required to activate the destination instruction have been received and, if so, enters the instruction address in the instruction queue.

**Figure 2.4.1. Basic Instruction Execution Mechanism**
During program execution, the number of entries in the instruction queue measures the degree of concurrency present in the program. The basic mechanism of Figure 2.4.1 can exploit this potential to a limited but significant degree: once the fetch unit has sent an operation packet off to the operation unit, it may immediately read another entry from the instruction queue without waiting for the instruction previously fetched to be completely processed. Thus a continuous stream of operation packets may flow from the fetch unit to the operation unit as long as the instruction queue is not empty.

This mechanism is called a circular pipeline—actively controlled by the flow of information packets traverses the ring of units leftwise. A number of packets may be flowing simultaneously in different parts of the ring on behalf of different instructions in concurrent execution. Thus, the ring operates as a pipeline system with all of its units actively processing packets at once. The degree of concurrency possible is limited by the number of units on the ring and the degree of pipelining within each unit. Additional concurrency may be exploited by splitting any unit in the ring into several units which can be allocated to concurrent activities. Ultimately, the level of concurrency is limited by the capacity of the data paths connecting the units of the ring. The basic mechanism is essentially that implemented in a prototype data flow processing element built by a group at the Texas Instruments Company.
2.5 Data Flow Languages and Simulation [5,6]

2.5.1 Data Flow Language

The success of any computer, data flow or otherwise, depends on the quality of its programming languages. Data flow machines demand high-level languages, since graphs, their machine languages, are not an appropriate programming medium; they are error-prone and hard to manipulate.

Three high-level language classes have been considered by data flow researchers. The first is the imperative class. For instance, the Texas Instrument group considered the use of a modified ASC Fortran compiler for their data flow machine. Compiler techniques for the translation of imperative high-level languages into data flow graphic languages have also been studied at Iowa State University. The second is the functional class. By functional, we mean those resembling pure Lisp, which is based on Church's lambda calculus, and Backus' FP, which is based on Curry's combinatory logic. This second class is now being studied in a data flow context at the University of Utah.

The third class consists of the so-called data flow languages, which are designed with data flow machines in mind. The most notable examples are Id, LAU, and Val. The syntax of these languages is essentially that of imperative languages. For example, all data flow languages include IF and LOOP statements. On the other hand, their semantics are basically that of functional languages.
2.5.2 Simulation of Data Flow Architecture (Using Concurrent Programming)

The hardware implementation of a dedicated data flow system appears to be quite complex. This is due to the problem complexity such an architecture is designed to solve. In the case of digital sequential circuits, the microprocessor (or the "programmable sequential machine") has provided logic designers with the feature of using less hardware components (hence reducing the cost of complex digital systems) and the flexibility of its adaptation to many problems. The cost of this flexibility is dependent upon more involved software design. In non-conventional architectures such as data flow architecture, we may expect that in the future dedicated systems will be simulated (or even implemented) on supercomputers. In fact, this architecture has so much parallelism and concurrency that make the use of a computer with a number of CPU's an attractive alternative. The data flow between processors and the activation of a processor and deactivation of another are provided via software that constitutes a so-called concurrent programming languages.

There has been a number of concurrent programming languages designed to achieve more processing power, faster computation, and flexible application of hardware to changing requirements. The term concurrency, however, has come to have more than one meaning. It is often applied to what might be called "apparent" concurrency; that is a number of processes sharing a single processor, slicing up its time among them. A less ambiguous term for this type of operation is multitasking, sometimes called multiprogramming.
Among these programming languages, we can mention ADA, concurrent C, and concurrent PASCAL (many versions). In this thesis, data flow machines are simulated using PASCAL PLUS programming language. Hence a "hardware" processor will be equivalent to a PASCAL PLUS process. Communication between one process and another is done with the help of monitors (channel monitors in our case). Data transfer from one process to another is accomplished with the use of data structures (mostly records). For instance, if a matrix and its size need to be passed to a process, a record of two fields (one for the matrix and the other for its size) is created in order to concatenate both data types into one. Let's note that PASCAL PLUS is not a real-time concurrent language but is time-shared. In other words, in function of time we don't really save anything because execution of concurrent processes is time-multiplexed by the operating system.
Chapter 3

DIGITAL IMAGE PROCESSING AND TRANSMISSION

3.1 Introduction

Digital image processing is the manipulation of images by computer. In a broader sense, image processing means the processing of multi-dimensional signals which is the case of most signals in the world. In general, image processing operations can be implemented by analog (optical, electro-optical, photographic) as well as digital methods. However, the inherent advantages in digital methods (applicability, flexibility, cost and accuracy) result in an increasing use of digital image processing. The rapid progress in computer-related technologies, such as LSI and VLSI, and the fact that processing units, memories and peripheral devices will continue to become faster and less expensive, will give a much greater impetus to digital image processing.

Any digital image processing scheme involves the following steps: data acquisition, image enhancement, feature extraction and/or segmentation, and finally, image understanding (interpretation) or evaluation (Figure 3.3.1).

a) Data acquisition: This step involves the acquisition, recording, sensing and converting the image into a digitized grid available for digital processing. Image digitization is done by scanning the image, sampling the analog gray scale values, quantizing them (i.e., discretizing the values obtained) and converting them into digital numbers.
b) Image enhancement: In this step an attempt is made to improve the quality of the image, by using a priori knowledge on the type of image to be processed or some known limitations and defects in the data acquisition and digitization process.

c) Feature extraction and/or segmentation: This step receives the gray-scale digital image and performs a sequence of numerical operations on it in order to extract features which are characteristic of the picture and non-redundant as possible. Such numerical operations may include noise filtering, contrast enhancement, edge sharpening, etc. More recently, the emphasis has been on the task of segmenting the image to first reduce the dimensionality of the search space, or partitioning it according to some relationship (such as isolate objects or regions in the image) and then perform a more reliable feature extraction within the context of a particular object.
d) Image Analysis: Image analysis performs numerical or logical operations on the extracted features (i.e., it deals with features and does not operate on the original picture elements). Thus, the first question raised is what is a feature and how does it describe the picture. A study in descriptive representation of pictorial data carried out in [7] by Firschein and Fishler gives the following types of goal-specific description: "descriptions for reconstruction", used to reconstruct a picture and provide detailed information on location, size, shape, color, texture, etc.; "descriptions for classification", used to distinguish one scene from another or classify a scene into one of many categories; "descriptions for retrieval of pictures", which must take into account possible user queries, and accordingly capture the content or meaning of the picture; and "descriptions for picture comprehension", used to aid the observer in understanding a picture. Usually the image analysis process is performed in a hierarchical way using either bottom-up techniques or top-down techniques.

3.2 Image Digitization [8]

Image or picture digitization, i.e., the process of converting the two-dimensional analog signal into numerical form, precedes any application of digital computers to image processing. The digitization of analog images is performed by a scanning process which sequentially addresses small "spots" on the analog image. The most common scanning process is the raster scan which addresses the image one line at a time and generates the earlier mentioned rectangular grid $f(i,j)$,
i = 0, 1, \ldots, M-1, j = 0, 1, \ldots, N-1 having the form shown in Figure 3.2.1(a). The grid contains M horizontally scanned lines, where each line consists of N adjacent "spots" or small regions called picture elements or pixels. (Usually the numbers M and N are integral powers of 2.) In this "series of rows" representation, the "spots" have a non-zero diameter given by the optical properties of the scanner. The "spread or spot size may differ between scanning devices; furthermore,

Figure 3.2.1 Digitizing an Analog Image to Convert it into a Rectangular Grid of Pixels. (a) The Digitized Image. (b) The 256 Gray Levels of an 8-Bit Pixel Integer. (The Notation Used by LANDSAT to Represent Gray Scale is Given in Parentheses)
pixels will not correspond exactly to the same spots in a picture even if the same scanner is used twice on a given image.

The scanning device samples the image brightness at each pixel location. Furthermore, it quantizes the measured values to discrete units and represents them by numerical values. The integer value generated by this scanning process at each pixel is called the gray level of the pixel. If the quantization allocates, say, 8 bits per integer value, then the gray level of a pixel may take any one of the possible \( 2^8 = 256 \) values (Figure 3.2.1(b)).

Therefore, each pixel has a location or address (line number \( i \) and column number \( j \)) and an 8-bit gray level value denotes as \( f(i,j) \). For binary or two-level images, \( f(i,j) \) can only take the value of 1 (white) and 0 (black).

3.3 Image Coding [9]

In this section, we discuss in some more detail common techniques used to code images and prepare them for storage.

Transmission of television signals for the 625-line system requires an analog transmission bandwidth of 5.5 MHz and a practical sampling rate of 13 MHz. Digital transmission techniques, however, have the advantage over analog that the signal impairment is much less, particularly for long distance transmission. If straightforward digitization is used, and if each pixel is represented by 8 bits, then the digital transmission of TV signals requires a transmission rate of 13 x 8 = 104 Mbits/sec. (Transmitting digitized speech signals requires
a transmission rate of 64 Kbits/sec.) Regarding storage, efficient techniques are required, for example, in archiving x-ray pictures or implementing picture databases that may include such things as engineering drawings, fingerprints, etc. Since, in general, digital images to be stored or transmitted involve a tremendous number of bits, the constraints on local storage and on existing transmission channels make it mandatory to efficiently code and compress the bandwidth of the image.

Reducing the transmission rate represents a more difficult task than compressing the image for storage, because the former requires coding that must be performed in real time. The (usually) non-real time storage reduction permits one to consider more complicated coding algorithms. Therefore, the aims of efficient image coding can be summarized as follows: (1) With regard to transmission, the aim is to reduced the required transmission rate for a given image quality (bandwidth reduction); (2) With regard to storage, the aim is to minimize the total number of bits required to code the special domain (efficient representation).

A number of coding techniques exist to compress imagery information and reduce their transmission bit rates, and as already mentioned, they are classified as intraframe (in-frame) coding techniques or interframe (frame-to-frame) coding techniques. Compared to intraframe techniques, further reduction is possible when interframe techniques are used, since they utilize the correlation between frames. For example, in typical television pictures there is usually an object in
motion, set against a stationary background. Two such successive frames are highly correlated; the background that is not changing conveys no new information and thus this represents an unnecessary redundancy. A significant reduction can be achieved by transmitting only frame differences, which are very small.

Our discussion here is restricted only to techniques used in intraframe coding. More specifically, we will be focusing on transform coding.

3.3.1 Transform Coding

a) Using the original \(N \times N\) picture array: In this coding technique, a mathematical transform of the picture is taken (i.e., a linear combination of the neighboring pixels from one line or adjacent lines is taken), and then the (selected) transform coefficients are quantized, processed and transmitted. Consider, for example, an \(N \times N\) picture \([f(x,y)]\) where \(f(x,y), x, y = 0, 1, \ldots, N-1\) represent the amplitude of image domain (original special domain) samples over the square picture array. Then its two-dimensional transform is given by another \(N \times N\) square array \([F(u,v)]\) where \(F(u,v), u,v = 0, 1, \ldots, N-1\) represent now the amplitude of transform domain coefficients. It is known that for the orthogonal transforms discussed in Chapter 5 (i.e., Fourier and Walsh-Hadamard transforms), the total energies in the two-dimensional special and transform domain are equal, as expressed by the following Parseval's relationship [10]
Furthermore, the transformation operation exhibits the property of concentrating the energy of the original image, which is usually uniformly distributed in the special domain, near the origin (i.e., near to smaller values of $u$ and $v$ indices) of the transform domain [11,12].

The major application of transforms is in the design of filters for image enhancement and also in certain data compaction techniques. They also provide the foundations for image reconstruction.

Through such a transformation the statistically dependent pixels are converted into "somewhat independent" (uncorrelated) coefficients [13]. Each coefficient is independently quantized and coded before transmission. The number of bits allocated to code the coefficients depends on the number of quantizer levels which is, in turn, dictated by the sensitivity of the human vision to the subjective effect of the quantization error. Figure 3.3.2 shows the block diagram of a transform coding system [14]. Since the transform coefficients are statistically "independent", the task of finding the optimum bit assignment (coding) is simplified significantly.

Besides obtaining the above mentioned "independent" or uncorrelated transform coefficients, we would like to use a transform that would also satisfy the second requirement that the image energy be compacted into as few coefficients as possible. In other words, if few of the transform domain coefficients are of large magnitude, then due to the
energy conservation property between the original image special domain and the transform domain, many of the remaining coefficients would be of low magnitude, and may, therefore, be discarded for bandwidth reduction purposes. When dropping some of these coefficients (usually the smaller valued ones), the mean-square error in reconstructing the original (sub) picture is minimum.

Since transform coding (such as Fourier, Hadamard, . . . ) require matrix multiplication on big image array (512 x 512, 1024 x 1024) and,
therefore, requires a substantial number of multiplication, fast transformations have been the solution to the time-consuming transformation. For instance, the computation of the Fourier transform requires $N^2$ calculations for the one-dimensional case and $N^4$ for the two-dimensional. It is shown that the one-dimensional Fourier transform requires $N \log_2 N$ calculations. The same algorithm can be used to compute the two-dimensional Fourier transform in $N^2 \log_2 (N^2)$ (Appendix A).

b) Dividing the picture array into subpictures: So far, we discussed transform coding on the entire $N \times N$ picture. However, it is also possible to divide the picture array into subpicture (called blocks) of smaller sizes, and to take two-dimensional transform independently in each subpicture converting it into a set of "more independent" transform coefficients. If a separate processor is allocated to each subpicture, this independent processing immediately suggests a parallel array structure system or a multiprocessor data flow configuration. Furthermore, as it will be explained in Chapter 5, the system should provide reconfigurable capability to adjust to the various sizes of these subpictures. Wintz [15] has used the Hadamard transform on a $256 \times 256$ picture. Specifically the picture was divided into 256 $16 \times 16$ subpictures and each subpicture was independently Hadamard transformed. Further reduction of the data space can be achieved by applying non-adaptive and adaptive truncation. In non-adaptive truncation, only few coefficients (those having the largest energy on the
average) out of the 256 of each subpicture are retained. (This is called "zonal sampling" or "masking" in the transform domain.) In adaptive truncation, all 256 coefficients are first evaluated and then only those coefficients that exceed a threshold are retained. (This is sometimes referred to as "threshold sampling".)

Dividing a picture into subpictures has the drawback that it neglects the redundancies that exist between the subpicture. On the other hand, however, Wintz showed that it has the advantage that it is more immune to effects of bit errors in the coefficients [15]. When the receiver reconstructs the pixels from linear combinations of the coefficients, an error in one coefficient leads to error in all the pixels reconstructed from it. If the whole NxN picture is transformed as a unit, then one or more errors in the coefficients result in some error in all the reconstructed pixels. If the NxN picture is first divided into kxk subpictures and each subpicture coded independently, then only the subpicture with errors are effected. Furthermore, the transform coding of a subdivided picture reduces the word length required for each transform coefficient, because the magnitude of all coefficients is bounded by the magnitude of the zero sequency term, which is now $k^2A$ instead of $N^2A$ where $A$ is the maximum value of the (sub)picture.

In Chapter 5 an extensive discussion of both Fourier and Hadamard transforms (along with the data-flow implementation) is explained. Programs simulating those transforms along with examples for illustrations are shown in Chapter 5.
4.1 Introduction

The quest to have a hierarchical knowledge of the image which greatly assists in analyzing the image in a simple way and represents it, in a compact and cost effective manner, is becoming increasingly demanding. One of the very important outcomes of it is a hierarchical data structure called quadtree. Hierarchical data structures are appealing because of their conceptual clarity and ease of implementation. It gives us the ability to focus only on the interesting subsets and the image data. This focusing results in an efficient representation and improved execution time which is necessary in many image processing applications. These data structures are based on the principle of recursive decomposition. Hierarchical data structures can be differentiated on the basis of: a) the type of data that they are used to present; b) the principle guiding the decomposition process; and c) resolution either variable or not.

In this chapter, we will introduce the quadtree* data structure of an image and discuss its importance in image processing. Section 4.3 gives some of the mathematical relations the quadtree structure has. Those relations will be useful for the software implementation of many

* In a quadtree, each father has four sons. Quadtree and tree will be used interchangeably in this chapter to refer to the quadtree.
algorithms in this thesis. Section 4.4 explains the regular decomposition algorithm with an example in Section 4.5. Section 4.6 explains an efficient software implementation for the regular decomposition algorithm.

4.2 Quadtree Structure of an Image

It is known that an N x N picture array can also be represented as a quadtree of different levels, each representing a resolution level of the image. For instance, level 0 (starting level) has the average intensity of the whole image, level 1 has four nodes each containing the average intensity of one of the quadrants P1, P2, P3 or P4. The second level has sixteen nodes. Each four of them contain the average intensities of the sons of P1, P2, P3 or P4. Figure 4.2.1(a) shows the quadtree of an 8 x 8 image. The index shown next to each node indicates the quadrant number at any level (Figure 4.2.1(b)) of the image. Each of those nodes contain the average intensity of the image area allocated by the index next to it.

Therefore, the quadtree gives a hierarchical representation of the image. In general, an N x N (N = 2^n) image can be represented in a quadtree having n+1 levels ranging from level 0 to level n. Level n has the highest resolution of the image which is of size N x N (or 2^n x 2^n). In this discussion, we will consider that the quadtree is stored in memory as a one-dimensional array such that tree (1) has the average intensity of the whole image; Tree (2), Tree (3), Tree (4) and Tree (5) have the average intensities of quadrants P1, P2, P3 and P4 of level 1.
Figure 4.2 (a): Quadtree representation of an 8 x 8 image showing the nodes' numbers and the levels.
Figure 4.2.1(b): figure showing quadrants (or regions) accessed by each node index of figure 4.2.1(a).

Figure 4.2.2: One dimensional array representing the quadtree structure in memory.
and so on. This indexing scheme is shown in Figure 4.2.2 (same as the one shown in Figure 4.2.1). We will see that this scheme is quite helpful to obtain quadtree relationships.

Each level \( k \) stores \( 2^k \times 2^k \) nodes. A node of a level \( k \) represents the average intensity of a quadrant (allocated as discussed above) of size \( 2^{n-k} \times 2^{n-k} \) where \( n = \log_2 N \). Therefore, if we go from level \( k \) to level \( k+1 \), the number of nodes will increase by

\[
\left(1 - \frac{2^k \times 2^k}{(2^{k+1}) \times (2^{k+1})}\right) \times 100\% = 75\%
\]

However, the quadrants allocated by each node at level \( k+1 \) will decrease in size by

\[
\left(1 - \frac{2^{n-(k+1)} \times 2^{n-(k+1)}}{2^{n-k} \times 2^{n-k}}\right) \times 100\% = 75\%
\]

From this, we can see that the higher the level is the more storage is required and the better the image is represented. This is what defines a small or high resolution image. In some applications (where speed may be a burden), a final description of the image may not be required. Therefore, a low resolution level of the image can be processed/ transmitted much faster.

Consider, for example, a raster scan image of 512 x 512 pixels. Assume that each pixel can take on values ranging from 0 to 255, which means that 8 bits are allocated to present a pixel. If such an image is to be transmitted over a 1200 bandline, it will take approximately...
half an hour (or longer if the image is in full color) before the receiver has the whole information. If such an image is to be displayed on a line-by-line basis, then it may take from 15 to 20 minutes before the user has any notion of what the final picture will be like. However, in many applications, the user may not require all the finest details of the original image. In other situations that involve interactive operations, he may want to stop receiving the image (or stop the device that displays the image for him) after he has obtained only a rough sketch of the image and identified that this is not the particular image of interest to him. In this case it is preferable for the transmitter to first transmit coarse information about the image (by starting transmission of nodes at a low level of the tree), and if the need arises, only then, continue to transmit successive refinements of earlier transmitted approximations to the original image.

In this chapter, we will see that the quadtree data structure will be used to apply the so-called regular decomposition of an image. In some applications, we need to extract an object that occupies some area in the image and reject the background which is of little or no importance. The regular decomposition algorithm will decompose the tree starting at level 0 up to a certain higher level. We will see that the quadtree provide the ultimate structure for the regular decomposition procedures.
4.3 Quadtree Mathematical Relations

In this section, we will give some relations that will be used for the software implementation of tree-based algorithms.

Let's consider an \( N \times N \) image \( P(N = 2^n) \), represented in the form of a quadtree structure stored as a one-dimensional array (as shown in Figures 4.2.1 and 4.2.2). As we have seen in Section 4.2, each node at a level \( k \) represents an image quadrant of size \( 2^{n-k} \times 2^{n-k} \). There are \( 2^k \times 2^k \) such nodes on one level. The total number of nodes a tree has is

\[
1 + 4 + 16 + 64 + \ldots + N^2 = (4N^2-1)/3 \quad (4.3.1)
\]

From this equation it is seen that the storage requirement of a tree structure exceeds that required by the image by about 33%.

The "father" of four sons is such as

\[
father = (son1 + son2 + son3 + son4)/4 \quad (4.3.2)
\]

The starting index of a level \( k \) is given by the relation

\[
STRT = (4^{k+2}) \text{ DIV } 3 \quad (4.3.3)
\]

The last index of a level \( k \) is given by the relation

\[
LAST = STRT + (2^k \times 2^k) -1 \quad (4.3.4)
\]

If at a certain time a node of index \( m \) at level \( k \) is of our interest and we would like to access the corresponding quadrant of that node at the last level \( n \), then we may proceed as follows. The rank of the node \( m \) at level \( k \) is given by

\[
RANK = m - STRT \quad (4.3.5)
\]

The starting level of the last level is given by

\[
STRT1 = (4^n + 2) \text{ DIV } 3 \quad (4.3.6)
\]
Therefore, the first node index of the quadrant to be accessed is given by

\[ \text{FIRSTINDEX} = \text{STRT1} + \text{RANK} \times (2^k \times 2^k) \]  \hspace{1cm} (4.3.7)

The last node index will be

\[ \text{LASTINDEX} = \text{FIRSTINDEX} + (2^k \times 2^k) - 1 \]  \hspace{1cm} (4.3.8)

FIRSTINDEX and LASTINDEX will enable us to construct a two-dimensional quadrant stored as a one-dimensional array starting at TREE (FIRSTINDEX) and ending at TREE (LASTINDEX). Appendix G has a procedure to construct the two-dimensional quadrant of an image from the one-dimensional array stored at level n of the tree (refer to procedure BUILDER).

In some situation having a node m at level k, we need to access its sons (this will be useful in regular decomposition algorithm). STRT and RANK are obtained from Equations (4.3.4) and (4.3.5). The first son of the father (node m) is then given by

\[ \text{FIRSTSON} = \text{STRT2} + \text{RANK} \times 4 \]  \hspace{1cm} (4.3.9)

where

\[ \text{STRT2} = (4^{k+1} + 2) \div 3 \]  \hspace{1cm} (4.3.10)

Similarly, we can work backwards to obtain the father's index from one of its sons' index. Those relations will be extensively used in the simulating programs in this thesis.

4.4 The Regular Decomposition Algorithm

The regular decomposition technique is presented here to yield a reduced quadtree representation of an image. The way picture decomposition in quadrant is performed is as follows [16,17]:
Initially, the entire picture is considered as a quadrant. One examines the whole quadrant to see whether there is anything informative there. The two critical issues in this procedures are: (1) to define an "importance factor" for the area examined which will give a measure of "informativeness" (this is very difficult because, in the general case, both syntactic and semantic information must be taken into account), and (2) to incorporate an appropriate discrimination function for looking into further details of the informative area. The following three possible cases exist:

1. If the area is characterized as non-informative, then that area of the picture may be eliminated from the data-structure without significant data loss. Such cases are, for example, large homogeneous areas.

2. The area is characterized as very informative (the discrimination function identifies many important features), and therefore, these data must be saved in the data-structure.

3. In the third case, the area is characterized with enough importance that makes us—at least so far—not sure of whether it should be eliminated from or saved in the data-structure. The picture must, therefore, be further subdivided into four quadrants and the same questions asked about each one of these four subquadrants.

To simplify the presentation, let us use average intensity as the coarse picture parameter (instead of the general "vector information") and thresholding as the basic processing operation to be used in decomposing a quadrant. The importance of any subquadrant is defined
relative to only its parent quadrant. The relative importance \( d(P,P^*) \) of a subquadrant \( P^* \) (where \( * = a, b, c \) or \( d \)) within a quadrant \( P \) is defined as

\[
d(P,P^*) = \frac{\text{average intensity of subquadrant } P^*}{\text{average intensity of quadrant } P} \quad (4.4.1)
\]

Thus, \( d(P,P^*) \) represents a percentage of information relative to the surrounding area, and this function is convenient since it is independent of depth in the tree.

The picture description function \( \alpha(d) = \alpha(d(P,P^*)) \) is defined as the importance of a quadrant \( P^* \) in picture \( P \). The following is an arbitrary discrimination function using threshold values \( \theta_1 \) and \( \theta_2 \):

\[
\begin{align*}
\alpha(d) &= \text{very informative}, & \theta_2 < d(P,P^*) < 1.0 \\
&= \text{not sure}, & \theta_1 < d(P,P^*) < \theta_2 \\
&= \text{non-informative}, & 0.0 < d(P,P^*) < \theta_1
\end{align*}
\quad (4.4.2)
\]

Thus, the tree-structure-creating algorithm is as follows:

Algorithm 4.4: Constructing the reduced tree representation (regular decomposition).

1. Find total average intensity of the entire picture which is considered as a quadrant.

2. Partition this original picture (quadrant) into four sub-regions (subquadrants).

3. Refine any "not sure" subregion, i.e., one with average intensity greater than or equal to \( \theta_1 \) and less than \( \theta_2 \) of the one higher level average intensity.
4. Do not retain any "non-informative" subregion, i.e., one with average intensity less than $\theta_1$ of the one higher level average intensity.

5. Stop refining, when a "very informative" subregion, i.e., one with average intensity greater than or equal to $\theta_2$ of the one higher level average intensity is found.

The advantages of regular decomposition as stated in [16,17] are:

1. Via unconditional partitioning, pictures which are physically too big to store in fast memory at one time can be processed as a sequence of subpictures extracted during preprocessing.

2. Regular decomposition enables addressing for rapid access to any geographical part of the image.

3. Regular decomposition retains explicitly in the data-structure a hierarchical description of picture patterns, elements and their relationships. Hence, this scheme may also be used in conjunction with syntactic pattern recognition algorithm.

4. Representations permit recursive analysis of subpictures.

5. The procedures treats all picture points alike. Hence, it does not prefer objects with any directional orientation or picture location.

6. The decomposition algorithm contains major routines (traversal, tree-creation) which are independent of image class. Small changes can adapt regular decomposition to widely different types of pictures.
7. The resultant tree data-structure distinguishes object from non-object (or background) and enables processing to locate separate objects.

4.5 Example

Consider the non-binary image of Figure 4.5.1(a). This entire picture is considered as a quadrant and its total intensity is 560.

Following the regular decomposition algorithm, this picture is partitioned into the four quadrants shown at the left side of Figure 4.5.1(b). Its quadtree representation is shown at the right side of Figure 4.5.1(b). Through Equation (4.4.1), the relative importance \( d(P, P^*) \) of each of these four subquadrants is: \( 140/560 = 0.25 \). By using the specific arbitrary threshold values \( \theta_1 = 0.25 \) and \( \theta_2 = 0.45 \) in Equation (4.4.2), these four quadrants are characterized as not sure since the relative importance of each of them is 0.25. Therefore, each of those quadrants are further decomposed as shown in the figure.

Partitioning further each of these "not sure" quadrants into four subquadrants yields the 4x4 array at the left end of Figure 4.5.1(c) and a new resolution level has now been added to its quadtree representation as shown at the right end of Figure 4.5.1(c). Scanning the 4x4 image row-by-row from left to right we compute the following relative importance of the quadrants: \( (4/140) = 0.028; (40/140) = 0.28; (40/140) = 0.28; (56/140) = 0.40 \ldots \) (by symmetry, all other quadrants have the same relative importances). It is noticed (through Equation (4.4.2)) that the tree nodes (or picture quadrants) with
Total intensity = 560

(a)

(b)

(c)
Figure 4.5.1: The resulting reduced tree representation of (a) following the regular decomposition algorithm (up to level 2). The black squares have an intensity of 18.

- □: Non informative
- ■: Very informative
- •: Not sure
values 40/140, 56/140 are characterized as not sure nodes and, therefore, will be further refined. Similarly, tree nodes with values 4/140 are characterized as "non-informative" and, therefore, will be dropped from the tree representation. Partitioning each of these new "not sure" quadrants into four subquadrants yields the 8 x 8 array of Figure 4.5.1(d), adding a new resolution level to the tree representation. Then again, using the same $\theta_1 = 0.25$ and $\theta_2 = 0.45$ in Equation (4.4.2), we come up with the reduced tree data structure for the non-binary picture of Figure 4.5.1(a) shown in Figure 4.5.1(d), where all "non-informative" nodes have been eliminated from the final representation.

If the original image was given with greater resolution, then all those "not sure" nodes (if any) would be further partitioned to generate the level $k = 3$ of tree representation. If no greater resolution is given, then these "not sure" nodes at level $k = 2$ are "leaf nodes", and therefore, are considered as "very informative" nodes since they are part of the final tree representation of the original picture of Figure 4.5.1(a).

Figure 4.5.2 shows the reduced tree representation of Figure 4.5.1(d) as an 8 x 8 image matrix.

4.6 Efficient Software Implementation of the Regular Decomposition Algorithm [18]

This section suggests a fast algorithm for the regular decomposition procedure described in Section 4.4. The tree is stored as a
one-dimensional array as explained in Section 4.1. If the image is of size \(N \times N\) \((N = 2^n)\), then its corresponding tree is stored at \(\text{TREE}(1), \text{TREE}(2), \ldots, \text{TREE}((4N^2 - 1)/3))\). The described algorithm is to decompose the tree starting at level \(k_1\) up to level \(k_2\). If at a certain level of the decomposition we find a certain number of not sure nodes, we would like to decompose their sons at the next level. This suggests a stack structure that keeps track of not sure nodes only. This way nodes labeled very informative or non-informative are not pushed on the stack and, therefore, their sons will never be visited (and hence their grandsons, \ldots). The stack in consideration is called dynamic stack. It has been implemented using Pascal pointers. In other words, the stack is of a variable size (which is the number of not sure nodes detected some time in the decomposition). We will see
that this structure (dynamic stack) saves memory locations since after a node has been decomposed we can delete it from the structure (by disposing the pointer pointing to it).

The algorithm is described in the flow chart of Figure 4.6.1. At the beginning, the starting level kl and the last level k2 are specified. Then, the indexes (or node numbers) of level kl are pushed on the stack. Those nodes are now considered as not sure since only not sure nodes are allowed to be pushed on the stack. On the next step, we scan the nodes on the stack and decompose them. For each node on the stack, we check his sons. If any of those sons is not sure, it is pushed on the top of the stack, otherwise it is labeled very informative or non-informative. After one node is decomposed, it is deleted from the structure (by disposing the pointer pointing to it). Then, the next node of level kl is decomposed similarly and so on until we reach the last node of level kl (scanning down the stack). At that instance, all nodes of level kl have been deleted from the structure and their not sure sons (at level kl + 1) have been added to the structure. If, however, no not sure sons have been detected, the stack will be empty. That means that all nodes of level kl + 1 have been labeled either very informative or non-informative. Then, the regular decomposition procedure is stopped. If, on the other hand, there were "not sure" sons pushed on the stack (stack not empty), we increment kl and proceed similarly until the last level k2 of decomposition is reached. Let's note that k2 may never be reached since at a certain level, the stack may become empty and decomposition would stop (as
Figure 4.6.1: Flow chart describing an efficient software implementation of the regular decomposition algorithm
discussed above). If, however, the last level \( k_2 \) has been reached, then there may be a number of not sure nodes pushed on the stack from the previous level decomposition. Since those nodes may contain informative sons, the stack is flushed and all its nodes considered are now labeled very informative. Those nodes are now deleted from the stack structure (by disposing all PASCAL nodes\(^*\)) and the stack is now empty.

This algorithm clearly has the following advantages:

1. It keeps track of only not sure nodes (indexes) on the stack. Therefore, sons of nodes labeled either informative or non-informative will never be visited.

2. The stack is dynamic. This enables us to have the minimum memory locations by disposing "no longer useful" nodes.

3. At the end of the process, the empty stack does not allocate any computer memory.

The PASCAL implementation of this algorithm is shown in Appendix G. Appendix H has a program to build the tree structure from the two-dimensional image. Besides it has a similar program (in PASCAL PLUS) that builds the tree structure concurrently. Each process is to build one branch of the tree and the father node at level 0 will then be computed.

\(^*\) For no confusion, the discussion talks about tree nodes (particularly the indexes as shown in Figure 4.2.1) rather than PASCAL nodes.
Chapter 5

DATA FLOW CONFIGURATIONS FOR TRANSFORM-CODING DIGITAL IMAGES

5.1 Introduction

As already mentioned, transform coding is used extensively in the image processing/transmission area. If we assume that \( f(x,y) \) represents the brightness intensity samples of an \( NxN \) (where \( N=2^n \)) discretized square image in the special domain, then its transform is denoted as another square array of transform coefficients \( F(u,v) \). In matrix form, an orthogonal transformation is generally described as follows:

\[
[F(u,v)] = \alpha [O_T] [f(x,y)] [O_T] \tag{5.1.1}
\]

where \([f(x,y)]\) represents the digitized \(NxN\) image, \([O_T]\) the transformation matrix, \(\alpha\) a constant dependent on the image size and \([F(u,v)]\) the transformation matrix. A look at this equation conveys how much computations is required to transform code an image. (In the Fourier case, the computations are complex multiplications and additions.)

Although fast algorithms have been designed to accelerate the speed of such transformation, the time requirement puts nevertheless a heavy burden on the efficiency of such systems. For a 1024 x 1024 gray level image digitized with 8 bits/pixel, \(8192\) Kbits are to be coded often repeatedly. To get around this burden, a system with a set of processors, each processing/coding a quadrant or subset of the image, is the key to the solution.
Another solution to reduce the number of pixels to be processed or coded is to remove redundant areas that convey very little information about the image. Indeed, in some applications, we may be dealing with images having an object occupying a small area of the image and surrounded by an homogeneous background. Little or no time should be spent on those areas in order to process such "non-informative" data. As we have seen in Chapter 4, applying the regular decomposition algorithm on this type of images seems to be the ultimate way to extract those objects from an image (and reject the background or spend little time on it) and process them. Not only would we be saving in time but also in storage and this helps overcoming the problem of low bandwidth communication channels. In this chapter we are concerned with two types of orthogonal transformations, the Walsh–Hadamard (often referred to as Hadamard) and the Fourier transforms.

For each case a general theory of the specified transformation is given. Besides, we will be designing data flow based systems having four processors to do the transformation by segmenting the image into four quadrants. Two cases will be considered for each transformation: the merging and non-merging cases. Then we will consider the receiver case where we will have to do inverse transformation. A generalized data flow diagram that adapts to either type of transformation will be discussed. Then, we will be comparing the tradeoff between having 1, 4, 16 and 64 processors executing concurrent transformations on the segmented image. Finally, we are to marry each of these systems (transmission systems) with the regular decomposition algorithm in
order to concurrently decompose and transmit informative areas of the
image.

Each of these data flow systems is simulated in a concurrent
program using PASCAL PLUS programming language. The simulation is done
according to the data flow concept as we will see later in this chapter.

5.2 General Theory of the Walsh-Hadamard Transformation [19]

5.2.1 Introduction

The Walsh transform of images (like the well-known Fourier
transform) has been used in pattern recognition, image transmission,
bandwidth reduction and image enhancement. For example, in pattern
recognition, instead of looking at the original image, one can search
for recognizable configurations of Walsh coefficients, or classifica-
tions may be accomplished by the investigation of the distribution of
image energy in the transform domain, or by matched filtering.
The Walsh-Hadamard transform has also been used to reduce image trans-
mision bandwidth and redundancy. For example, one may take the
two-dimensional Walsh-Hadamard transform of the image and, instead of
the original image, transmit its transform. Since many of the low
magnitude transform coefficients contain little information about the
image, they can be filtered out and not transmitted, thus reducing the
required transmission bandwidth. Furthermore, the two-dimensional
Walsh transform has the property that a second Walsh transform on the
function $F(u,v)$ will result in the original image $f(x,y)$. Thus, at the
receiving end, an approximation to the original image can be obtained
by taking the inverse Walsh transform on the received truncated coefficients. Finally, to provide image enhancement, each Walsh coefficient may be multiplied by an appropriate function of sequences, before being used in the inverse transform operation to yield an improved received image.

5.2.2 The Walsh–Hadamard Transformation

Since we are interested in pictures of $N \times N$ pixels with $N = 2^n$, a more convenient representation of the $N$ Walsh functions to be used for transformation is the form of the $N \times N$ Hadamard matrix. This matrix is a square array of plus and minus ones whose rows and columns are orthogonal to each other. Using the matrix form, high-speed computational algorithms perform fast Walsh–Hadamard transformations (Appendix A).

Given the core $2 \times 2$ Hadamard matrix, $[H_1]$, in matrix form

$$[H_1] = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad (5.2.1)$$

then the $4 \times 4$ Hadamard matrix is generated by

$$[H_2] = [H_1] \times [H_1] \quad (5.2.2)$$

where $\times$ is the Kronecker product. Thus,

$$[H_2] = [H_1] \times [H_1] = (h_{ij} [H_1]) = \begin{bmatrix} [H_1] & [H_1] \\ [H_1] & [-H_1] \end{bmatrix} \quad (5.2.3)$$

Multiplying $[H_2]$ of Equation (5.2.3) with $[H_1]$, we get:
\[ [H_2] \times [H_1] = ([H_2], hij) = \begin{bmatrix} [H_2] & [H_2] \\ [H_2] & [-H_2] \end{bmatrix} = [H_3] \quad (5.2.4) \]

In general, the \(2^k \times 2^k\) Hadamard matrix \([H_k]\) is obtained from the previous \(2^{(k-1)} \times 2^{(k-1)}\) Hadamard matrix as follows:

\[
[H_k] = \begin{bmatrix}
[H_{k-1}] & [H_{k-1}] \\
[H_{k-1}] & [-H_{k-1}]
\end{bmatrix}, \quad k = 1, 2, \ldots, n \quad (5.2.5)
\]

By taking advantage of this recursive formula for generation of the Hadamard matrices, a transform algorithm can be developed which requires only \(N^2 \log_2 N^2\) additions or subtractions (Appendix A). Also efficiency may be enhanced still further by use of a matrix factorization technique developed in [20]. For instance, the \(16 \times 16\) Hadamard matrix is generated recursively and is the following:

\[
\begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & -1 & -1 & -1 & 1 & -1 & -1 & -1 & 1 & -1 & -1 & -1 & 1 & -1 & -1 & -1 \\
1 & 1 & -1 & -1 & 1 & -1 & -1 & -1 & 1 & -1 & -1 & -1 & 1 & -1 & -1 & -1 \\
1 & -1 & -1 & -1 & 1 & -1 & -1 & -1 & 1 & -1 & -1 & -1 & 1 & -1 & -1 & -1 \\
1 & 1 & 1 & -1 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\
1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
1 & 1 & 1 & -1 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1
\end{pmatrix}
\]
In general, given an original N\times N picture \([P_n]\) (where \(N=2^n\)), its Hadamard transform coding is an N\times N picture \([F_n]\) given by the formula:

\[ [F_n] = [H_n] [P_n] [H_n] \]  

(5.2.6)

where \([H_n]\) is the \(N\times N\) Hadamard matrix.

Some of the properties of the Hadamard matrices are:

a) Since each Hadamard vector (matrix row or column) corresponds to a Walsh function and since Walsh functions are orthogonal, then the dot product of the two Hadamard vectors is zero.

b) A Hadamard matrix generated from Kronecker products is symmetric, i.e.,

\[ H(i,j) = H(j,i) , \ i,j = 0, 1, \ldots, N-1 \]  

(5.2.7)

and, therefore, in matrix form we have:

\[ [H_n] = [H_n^T] \]  

(5.2.8)

c) Since the Hadamard matrix is symmetric (from its definition), we have the inverse property

\[ [H_n] = N[H_n^{-1}] \]  

(5.2.9)

d) The Hadamard transform of a homogeneous image (having all its pixels of the same intensity value \(C\)) is a matrix \(A_{ij}\) such that

\[ A_{11} = C \quad \text{and} \quad A_{ij} = 0 \quad \text{(for any } (i,j) \neq (1,1).) \]

This is proved in Appendix B.

5.3 General Theory of the Fourier Transformation [19,21]

This section briefly describes the two-dimensional discrete Fourier transform. In general, if \(f(x,y)\) represents the intensity samples of the \(N\times N\) discretized image in its spacial domain, then the
image's two-dimensional discrete Fourier transform is defined as:

\[
F(u,v) = \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) \exp\left(-\frac{2\pi i}{N} (ux + vy)\right) \quad (5.3.1)
\]

\[
u, v = 0, 1, 2, \ldots, N-1
\]

If the Fourier transform of an image is transmitted, then the receiver can reconstruct back the original image by computing the inverse two-dimensional Fourier transform of the received information given by:

\[
f(x,y) = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} F(u,v) \exp\left(\frac{2\pi i}{N} (ux + vy)\right) \quad (5.3.2)
\]

\[
u, x = 0, 1, 2, \ldots, N-1
\]

Instead of using the double summation equations above, the two-dimensional Fourier of an image may also be expressed as the product of matrices. The matrix notation is very convenient when implementing the transform algorithms on digital computers, since matrix factorization techniques can be used to reduce the number of computations and the time for obtaining the transform of an image.

Let the input image be denoted as an \(N\times N\) square array of pixels \([f(x,y)], x,y = 0, 1, \ldots, N-1\). In the Fourier case, if we let

\[W = e^{-i2\pi/N} = \exp\left(-\frac{i2\pi}{N}\right) \quad (5.3.3)\]

then the \(N\times N\) complex Fourier matrix is given as the following array of \(W^{2k}\) elements:
Then the two-dimensional Fourier transform of the input image matrix \([f(x,y)]\) is given by the matrix product
\[
[F(u,v)] = [W^k] [f(x,y)] [W^k]
\]
(5.3.5)

5.4 Data Flow Configurations for Concurrent Image Transformation

5.4.1 Introduction

As we already mentioned, the huge amount of data involved in processing/transmitting digital images puts a heavy burden on the processing time required by the transformation processor. In this section, we are to exploit the parallelism that those transformations may offer in order to increase the efficiency of the system in question. We will be applying the data flow concept in order to accomplish
concurrent processing and have an optimal system throughput by keeping its processors "busy" all the time. For each type of transformation (Fourier or Hadamard), we will be considering two cases: merging and non-merging. For both cases, we are to come up with a unified approach that is adaptable (in modular form) to either transformation. Each of these data flow configurations will be simulated using concurrent programming PASCAL PLUS.

5.4.2 Merging Case

For an \(N\times N\) image \(N=2^n\), we wish to subdivide it into four quadrants \(P_1, P_2, P_3\) and \(P_4\) and assign to each quadrant a processor to transform code it. Each quadrant is of size \((N/2 \times N/2)\). The transformation equation (of general form Equation 5.1.1) will be applied to each of these quadrants. However, the transform of the whole image is different from obtaining the transforms of the four quadrants independently. Therefore, we need to somehow merge the independent transforms (of each quadrant) together in order to obtain the transform of the whole image. This is why we differentiate between two cases: the merging case in which the image is treated as a whole and the non-merging case in which each quadrant is treated independently. In this section, we will consider the merging case for both Fourier and Hadamard transforms.

a) Data flow configuration for Hadamard transformation: Let \(P\) denote an \(N\times N\) digital image \((N=2^n)\). The Hadamard transform of the image \(P\) is an \(N\times N\) picture \(F\) given by the formula:
\[ [F] = \frac{1}{N^2} \cdot [H] \cdot [P] \cdot [H] \]  

(5.4.1)

where \([H]\) is the \(N \times N\) Hadamard matrix which is defined by the recursion:

\[
[H] = [H_N] = \begin{bmatrix}
H_{N/2} & H_{N/2} \\
H_{N/2} & -H_{N/2}
\end{bmatrix} \quad N \geq 2 \tag{5.4.2}
\]

\[
[H_1] = \begin{bmatrix}
1 & 1 \\
1 & -1
\end{bmatrix}
\]

From now on, in order to simplify the evaluation procedure, we omit the factor \(\frac{1}{N^2}\) from Equation (5.4.1), yielding to a new expression:

\[ [F] = [H] \cdot [P] \cdot [H] \]  

(5.4.3)

The picture can be written in terms of its four subquadrants, \(P_1, P_2, P_3, \) and \(P_4\), as:

\[
[P] = \begin{bmatrix}
P_1 & P_2 \\
P_3 & P_4
\end{bmatrix} \tag{5.4.4}
\]

and the transform \(F\) can be written in terms of its four subquadrants as:

\[
[F] = \begin{bmatrix}
F_1 & F_2 \\
F_3 & F_4
\end{bmatrix} \tag{5.4.5}
\]

From (5.4.2), (5.4.3), (5.4.4) and (5.4.5), we can write:
where \( A_i = H_{N/2} \cdot P_i \cdot H_{N/2} \), \( i = 1, 2, 3, 4 \). \( A_1, A_2, A_3 \) and \( A_4 \) correspond to the Hadamard transforms of the four quadrants \( P_1, P_2, P_3 \) and \( P_4 \).

We proceed now to introduce a data flow machine to implement this parallelism. In Figure 5.4.7 is shown the transformation procedure for an image of arbitrary size. At the first stage of the system, the IMAGE SUBDIVIDER processor reads the image (as an input to it) from an image buffer and subdivides it into four quadrants \( P_1, P_2, P_3 \) and \( P_4 \) (upper left, upper right, lower left and lower right, respectively). Those quadrants are sent through channels \( R_1, R_2, R_3 \) and \( R_4 \) to the HADAMARD TRANSFORM processors (1, 2, 3 and 4, respectively). At the

---

* The word channel will be used to indicate either the data path between two processors or the communication channel between two processes (in terms of PASCAL PLUS simulation).
Figure 5.4.7: Data Flow configuration for the Hadamard transformation (merging case)
same time the HADAMARD GENERATOR processor generates the corresponding size Hadamard matrix recursively (having as its input half the image size) and sends a version of it along with its size to each of the four transformation processors through channels M1, M2, M3 and M4, respectively. The size will inform the transformation processors of the matrix size it will be operating on (in this case multiplying).

At the next level in the figure, we have the transformation processors: HADAMARD TRANSFORM1, 2, 3 and 4. HADAMARD TRANSFORM 1 receives P1 through R1, and the Hadamard matrix along with its size through M1. Then it transforms P1 into A1 such that:

\[ A_1 = H_{N/2} P_1 H_{N/2} \]

At the end, it passes this matrix along with its size to the four MERGER processors through S11, S12, S13 and S14. Here again the size indicates to the mergers the size of matrices it will be adding or subtracting. The three other HADAMARD TRANSFORM processors basically do the same job except that they do not send any matrix size to the mergers (since already HADAMARD TRANSFORM 1 processor sent that size). Therefore, HADAMARD TRANSFORM2 receives P2 through R2 and \( H_{N/2} \) along with its size through M2, does the transformation \( A_2 = H_{N/2} P_2 H_{N/2} \) and sends this result to the four MERGER processors through S21, S22, S23 and S24. HADAMARD TRANSFORM3 receives P3 through R3 and \( H_{N/3} \) along with its size through M3, does the transform \( A_3 = H_{N/2} P_3 H_{N/2} \) and sends this result to the four MERGER processors through S31, S32, S33 and S34. Similarly, HADAMARD TRANSFORM 4 receives P4 through R4 and \( H_{N/3} \) along with its size through M4, does the transformation
A4 = H_{N/2} P4 H_{N/2} and sends this result to the four MERGER processors through S41, S42, S43 and S44.

The MERGER processors are to add and/or subtract the independent quadrant transforms in order to obtain the overall image transform (not yet allocated to any specific quadrant). For instance, MERGER1, who at this stage received its inputs A1, A2, A3 and A4 through channels S11, S21, S31 and S41, will compute F1 such as

\[ F = A1 + A2 + A3 + A4 \]

and sends it to the RESULT processor along with its size through MM1. MERGER2 computes F2 = A1 - A2 + A3 - A4 and sends it to RESULT through MM2. MERGER3 computes F3 = A1 - A2 - A3 - A4 and sends it to RESULT through MM3. MERGER4 computes F4 = A1 - A2 - A3 + A4 and sends it to RESULT through MM4. Finally, RESULT processor receives F1, F2, F3 and F4. It allocates each of them to its corresponding quadrant [F1 (upper left), F2 (upper right) ... ], serializes the overall transform and routes it (along with some heading information) to the transmitter.

The four mergers do not always constitute four different processors. In this particular transformation, we can exploit a lot of parallelism as shown in Equation (5.4.6). The structure proposed in Figure 5.4.8 shows this parallelism. As we can see from this data flow structure, we save processors. If we consider the previous structure (shown in Figure 5.4.7), we need three processors for each merger (adders or substracters) to compute each Fi the fastest possible way. Overall we would need 12 processors (three internal to each merger), while in the
data-flow graph of Figure 5.4.8 we need only eight knowing that the processing time is still the same.

![Data-flow graph of Figure 5.4.8](image)

**Figure 5.4.8** A Proposed Data-Flow Graph that Substitutes the Mergers of Figure 5.4.7.

b) Data flow configurations for Fourier transformation: In this section our purpose is to come up with a data flow system configuration to transform code an image using the Fourier transformation. By definition, the Fourier transform of an image P of size NxN is an NxN picture F given by the formula:

\[
[F] = [W^{\frac{N}{2}}] \cdot [P] \cdot [W^{\frac{N}{2}}]
\]  

(5.4.9)
where \( W^{\ell k} \) denotes the \( N \times N \) Fourier matrix such as:

\[
W = \exp\left(-i \frac{2\pi}{N}\right)
\]  

and

\[
\ell, k = 0, 1, \ldots, N-1
\]

For example when \( N=4 \), the Fourier matrix becomes

\[
egin{bmatrix}
  W^0 & W^0 & W^0 & W^0 \\
  W^0 & W^1 & W^2 & W^3 \\
  W^0 & W^2 & W^4 & W^6 \\
  W^0 & W^3 & W^6 & W^9
\end{bmatrix}
\]  

In this transformation, we cannot trivially generate the Fourier matrix of level \( k \) from level \( k-1 \). The first restriction is that \( W \) is a function of the image size. Therefore, the recursive matrix generation is not applicable to this type of transformation as it was for the Hadamard case. Hence, we may want to generate each quadrant separately since they are not the same as in the previous case. Let's subdivide the Fourier matrix into four submatrices \( W_a^{\ell k}, W_b^{\ell k}, W_c^{\ell k} \) and \( W_d^{\ell k} \) such as:

\[
[W^{\ell k}] = 
\begin{bmatrix}
  W_a^{\ell k} & W_b^{\ell k} \\
  W_c^{\ell k} & W_d^{\ell k}
\end{bmatrix}
\]  

(5.4.12)
where $W_a^{\ell k}$, $W_b^{\ell k}$, $W_c^{\ell k}$ and $W_d^{\ell k}$ are defined as follows:

- $W_a^{\ell k}$ is such as $\ell = 0, 1, \ldots, N/2-1$, $k = 0, 1, \ldots, N/2-1$;
- $W_b^{\ell k}$ is such as $\ell = 0, 1, \ldots, N/2-1$, $k = \frac{N}{2}, \frac{N}{2} + 1, \ldots, N$;
- $W_c^{\ell k}$ is such as $\ell = \frac{N}{2}, \frac{N}{2} + 1, \ldots, N$, $k = 0, 1, \ldots, N/2-1$;
- $W_d^{\ell k}$ is such as $\ell = \frac{N}{2}, \frac{N}{2} + 1, \ldots, N$, $k = \frac{N}{2}, \frac{N}{2} + 1, \ldots, N$;

$W_a = W_b = W_c = W_d = W = \exp(-i2\pi/N)$.

From now on, we will omit $l_k$ and denote the above matrices: $W$, $W_a$, $W_b$, $W_c$ and $W_d$.

Using Equations (5.4.4), (5.4.5), (5.4.9) and (5.4.12), we can write

\[
\begin{bmatrix}
F_1 & F_2 \\
F_3 & F_4
\end{bmatrix} =
\begin{bmatrix}
W_a & W_b \\
W_c & W_d
\end{bmatrix}
\begin{bmatrix}
P_1 & P_2 \\
P_3 & P_4
\end{bmatrix}
\begin{bmatrix}
W_a & W_b \\
W_c & W_d
\end{bmatrix}
\]

or

\[
F_1 = W_a P_1 W_a + W_b P_3 W_a + W_a P_2 W_c + W_b P_4 W_c \quad (5.4.13)
\]
\[
F_2 = W_a P_1 W_b + W_b P_3 W_b + W_a P_2 W_d + W_b P_4 W_d \quad (5.4.14)
\]
\[
F_3 = W_c P_1 W_a + W_d P_3 W_a + W_c P_2 W_c + W_d P_4 W_c \quad (5.4.15)
\]
\[
F_4 = W_c P_1 W_b + W_d P_3 W_b + W_c P_2 W_d + W_d P_4 W_d \quad (5.4.16)
\]

A look at these equations does not obviously imply a similar structure as for the Hadamard case since $F_i$ ($i = 1, 2, 3, 4$) is a function of $P_1$, $P_2$, $P_3$ and $P_4$. However, if we treat all the computations function of $P_1$ ($i = 1, 2, 3, 4$) in one block, we will have a
similar structure. The proposed data flow diagram is shown in Figure 5.4.17. Here is a brief explanation of how the system works.

1. IMAGE SUBDIVIDER: As for the Hadamard transformation case, this processor reads the image from an image buffer and subdivides it into four quadrants, P1, P2, P3 and P4. Those quadrants are sent along with their size through channels R1, R2, R3 and R4.

2. FOURIER GENERATOR: This processor generates the proper size Fourier matrix subdivided into four submatrices (Wa, Wb, Wc and Wd) and routes a triplet of three such submatrices to each of the Fourier processors through M1, M2, M3 and M4. The triplets sent through those channels are different.

3. FOURIER PROCESSOR1:
   a) It receives P1 and its size through R1 and the triplet (Wa, Wb, Wc) through M1.
   b) It does the following concurrent transformations:
      \[ A = WaP1Wa \quad B = WaP1Wb \quad C = WcP1Wa \quad D = WcP1Wb \]
   c) It sends A, B, C and D to MERGER1, 2, 3 and 4 through S11, S12, S13 and S14, respectively.

4. FOURIER PROCESSOR2:
   a) It receives P2 and its size through R2 and the triplet (Wa, Wc, Wd) through M2.
   b) It does the following concurrent transformations:
      \[ A = WaP2Wc \quad B = WaP2Wd \quad C = WcP2Wc \quad D = WcP2Wd \]
   c) It sends A, B, C and D to MERGER1, 2, 3 and 4 through S21, S22, S23 and S24, respectively.
Figure 5.4. 17: Data flow configuration for the Fourier transformation (merging case)
5. FOURIER PROCESSOR3:
   a) It receives P3 and its size through R3 and the triplet (Wa, Wb, Wd) through M3.
   b) It does the following concurrent transformations:
      \[ A = WbP3Wa, \quad B = WbP3Wb, \quad C = WdP3Wa, \quad D = WdP3Wb \]
   c) It sends A, B, C and D to MERGER1, 2, 3 and 4 through S31, S32, S33 and S34, respectively.

6. FOURIER PROCESSOR4:
   a) It receives P4 and its size through R4 and the triplet (Wb, Wc, Wd) through M4.
   b) It does the following concurrent transformations:
      \[ A = WbP4Wc, \quad B = WbP4Wd, \quad C = WdP4Wc, \quad D = WdP4Wd \]
   c) It sends A, B, C and D to MERGER1, 2, 3 and 4 through S41, S42, S43 and S44, respectively.

7. MERGER1:
   a) It receives its input (including the matrix size) through S11, S21, S31 and S41.
   b) It computes F1 such as
      \[ F1 = WaP1Wa + WbP3Wa + WaP2Wc + WbP4Wc \]
   c) It sends F1 and its size to RESULT through MM1.

8. MERGER2:
   a) It receives its input (including the matrix size) through S12, S22, S32 and S42.
   b) It computes F2 such as
      \[ F2 = WaP1Wb + WbP3Wb + WaP2Wd + WbP4Wd \]
c) It sends F2 to RESULT through MM2.

9. MERGER3:
   a) It receives its inputs (including the matrix size) through S13, S23, S33 and S43.
   b) It computes F3 such as
      \[ F3 = WcP1Wa + WdP3Wa + WcP2Wc + WdP4Wc \]
   c) It sends F3 to RESULT through MM3.

10. MERGER4:
    a) It receives its inputs (including the matrix size) through S14, S24, S34 and S44.
    b) It computes F4 such as
       \[ F4 = WcP1Wb + WdP3Wb + WcP2Wd + WdP4Wd \]
    c) It sends F4 to RESULT through MM4.

11. RESULT:
    a) It collects its inputs from channels MM1, MM2, MM3 and MM4 (F1 along with the matrix size of RESULT processor, F2, F3 and F4, respectively).
    b) It allocates each transform quadrant to its corresponding location (upper left, upper right, . . . ).
    c) It serializes the transformation matrix row by row and routes it to the transmitter along with the same heading information.

Although in modular form the two systems look alike, there are minor differences. For instance, the FOURIER GENERATOR processor generates four different submatrices and each of the FOURIER PROCESSOR processors does four different computations as seen in the above
discussion. This makes the system slower. However, we indicated that for each FOURIER PROCESSOR the computations are done concurrently. Figure 5.4.18 shows the configuration used inside FOURIER PROCESSOR1.

As far as the FOURIER GENERATOR is concerned, we may use the same scheme. In fact the four Fourier submatrices are generated simultaneously as shown in Figure 5.4.19. Then each triplet is concatenated into one data structure and sent to the corresponding receiver processor.

Let's note that even with those improvements to the Fourier transform system, we may never be able to make it as fast as the Hadamard transform system due to the complex computations required by
the former system. A comparison between those systems is given later on in this chapter.

c) Software simulation: The systems of part a and b are simulated using PASCAL PLUS programming language. Appendix C has the program for Hadamard transformation and Appendix D has the program for Fourier transformation. The simulation is done according to the data-flow configurations of Figures 5.4.7 and 5.4.17. Along with each program, there are different examples for illustration.

d) Generalized transformation system: Due to the similarities of both transformations, as we have seen from the previous discussion, Figure 5.4.20 shows a general data flow diagram. This diagram is applicable to either transformation. It may be also proved that this configuration could be general to any orthogonal transformation if we
Figure 5.4.20: General data flow graph applicable for either Fourier or Hadamard Transformation (merging case).
follow the same approach. This suggests an adaptable system that could adapt to any type of orthogonal transformation.

5.4.3 Non-Merging Case

In this section, transformation is applied concurrently to the four quadrants of the image. However, they are not merged together as in the previous case. In other words, each quadrant is treated independently. A brief discussion of those systems will be given in this section.

a) Hadamard transformation: The data flow system in question is shown in Figure 5.4.21. Each HADAMARD TRANSFORM processor transform-codes one quadrant of the image and sends the results to RESULT processor, which allocates each transform to its correspondent quadrant then serializes the transform coefficients row by row so they can be transmitted. As compared to the previous case (merging), we reduce drastically the number of additions required by the MERGER processor.

b) Fourier transformation: The corresponding data flow system is shown in Figure 5.4.22. Each FOURIER TRANSFORM processor transform-codes one quadrant of the image and sends the results to RESULT processor. The difference between this system and the previous one (merging) is

1. The FOURIER GENERATOR processor generates a matrix of size equal one fourth of the overall Fourier matrix. This matrix is what
Figure 5.4.21: Data flow configuration for the Hadamard transformation (Non-merging case).
Figure 5.4.22: Data flow configuration for the Fourier transformation (Non-merging case).
Figure 5.4.23: General Data flow graph applicable for either Fourier or Hadamard transformation (Non-merging case).
we previously called Wa. Therefore, saving in time and data is obvious.

2. The FOURIER TRANSFORM processors do only one operation as compared to the previous case where we had to do four.

3. We omit the MERGE processors.

c) Software simulation: The system of part a is simulated in PASCAL PLUS in Appendix E along with some examples. The simulation of the system in part b follows exactly the same principle.

d) Generalized transformation system: Figure 5.4.23 shows a general data flow diagram that could be adapted to either transformation. It may be proved that this configuration could be applied to any orthogonal transformation. The generalized system of Figure 5.4.23 is more adequate than the one of Figure 5.4.20 since there are no subprocessors in any of the processors shown. Besides, this configuration is much faster than the previous one due to the substantial reduction in additions consumed by the MERGER processor.

5.5 Data Flow Systems for Inverse Transformation

So far, we have considered data flow systems that transform images. The parallelism scheme used makes those systems quite fast and efficient. Therefore, at the receiver end, we would like to use a fast scheme as well in order to do inverse transformation on transformed images. In this section our purpose is to prove that the four
previous systems used for transmission could be used with minor changes to do inverse transformation on received images. We will, therefore, consider four cases.

a) Inverse Hadamard transformation (merging case): This system is to be used with the one in Figure 5.4.7. In that system we implement the four following equations:

\[
\begin{align*}
F_1 &= (A_1 + A_2) + (A_3 + A_4) \quad (5.5.1) \\
F_2 &= (A_1 - A_2) + (A_3 - A_4) \quad (5.5.2) \\
F_3 &= (A_1 + A_2) - (A_3 + A_4) \quad (5.5.3) \\
F_4 &= (A_1 - A_2) - (A_3 - A_4) \quad (5.5.4)
\end{align*}
\]

where

\[ A_i = H_{N/2} \cdot P_i \cdot H_{N/2} \]

Pre- and post multiplying the above equations by \( H_{N/2}^{-1} \) (inverse Hadamard matrix), we get the four following equations:

\[
\begin{align*}
H_{N/2}^{-1} \cdot F_1 \cdot H_{N/2}^{-1} &= (P_1 + P_2) + (P_3 + P_4) \quad (5.5.5) \\
H_{N/2}^{-1} \cdot F_2 \cdot H_{N/2}^{-1} &= (P_1 - P_2) + (P_3 - P_4) \quad (5.5.6) \\
H_{N/2}^{-1} \cdot F_3 \cdot H_{N/2}^{-1} &= (P_1 - P_2) - (P_3 + P_4) \quad (5.5.7) \\
H_{N/2}^{-1} \cdot F_4 \cdot H_{N/2}^{-1} &= (P_1 + P_2) - (P_3 - P_4) \quad (5.5.8)
\end{align*}
\]

Since the Hadamard matrix is symmetric, we can write

\[ H_{N/2} \cdot H_{N/2} = N/2 \cdot I \]

or

\[ H_{N/2}^{-1} \cdot H_{N/2}^{-1} = N/2 \cdot H_{N/2}^{-1} \cdot H_{N/2}^{-1} \]

This implies

\[ H_{N/2}^{-1} = \frac{2}{N} \cdot H_{N/2} \]
Using Equation (5.5.9) and adding Equations (5.5.5), (5.5.6), (5.5.7) and (5.5.8) together, we obtain the equation

\[ P_1 = \frac{1}{N^2} H_{N/2} \cdot (F_1 + F_2 + F_3 + F_4) \cdot H_{N/2} \quad (5.5.10) \]

Similarly we can obtain the equations:

\[ P_2 = \frac{1}{N^2} H_{N/2} \cdot (F_1 - F_2 + F_3 - F_4) \cdot H_{N/2} \quad (5.5.11) \]
\[ P_3 = \frac{1}{N^2} H_{N/2} \cdot (F_1 + F_2 - F_3 - F_4) \cdot H_{N/2} \quad (5.5.12) \]
\[ P_4 = \frac{1}{N^2} H_{N/2} \cdot (F_1 - F_2 - F_3 + F_4) \cdot H_{N/2} \quad (5.5.13) \]

A close look at Equations (5.5.10), (5.5.11), (5.5.12) and (5.5.13) shows that these equations are similar to Equations (5.5.1), (5.5.2), (5.5.3) and (5.5.4), respectively. The only difference is the multiplying factor \(1/N^2\). This suggests that at the receiver we can use the same system of Figure 5.4.7 by dividing the matrices going to RESULT by \(N^2\). Figure 5.5.14 shows this system's configuration.

In the software simulation of Figure 5.4.7, the same program is used to do transform and inverse transform (Hadamard) of the image. A boolean variable named TRANSMIT is set to true when the program is used for transformation and to false for inverse transformation.

b) Inverse Fourier transformation (merging case): This system is to be used in conjunction with the one of Figure 5.4.17. For an image \(P\) of size \(N\times N\), the Fourier transform is given as:

\[ F = W \cdot P \cdot W \quad (5.5.15) \]

where

\[ W = W_N^{2k} \quad (k = 0, 1, \ldots, N-1) \quad W_N = \exp(-j \frac{2\pi}{N}) \]
Figure 5.5.14: Data Flow configuration for the inverse Hadamard transformation (merging case). RESULT divides its inputs by $N^2$. 
This definition clearly shows that the Fourier matrix is symmetric. Therefore,

\[ W_N \cdot W_N^{-1} = N \cdot I = N \cdot W_N \cdot W_N^{-1} \]

or

\[ W_N^{-1} = (1/N) \cdot W_N \]  \hspace{1cm} (5.5.16)

Combining Equations (5.5.15) and (5.5.16), we get

\[ P = (1/N^2) W \cdot F \cdot W \]  \hspace{1cm} (5.5.17)

This equation clearly shows that subdividing the transformed image as done in Sections 5.4.2(b) and applying the same algorithm of Figure 5.4.17 (with the exception of dividing the result by \( N^2 \)) will give us the original image. Figure 5.5.18 shows the data-flow diagram used at the receiver end. As a conclusion we can say that an adaptable system that does either transformation or inverse transformation is realizable for both types of transformation.

c) Inverse Hadamard and Fourier (non-merging case): In this section the systems in question will be used in conjunction with those of Figure 5.4.21 and 5.4.22. In the Hadamard case, using Equation 5.5.9 we can write

\[ P_1 = \left( \frac{2}{N} \right)^2 H_{N/2} \cdot F_1 \cdot H_{N/2} \]  \hspace{1cm} (5.5.19)

Therefore, the data-flow system of Figure 5.4.21 can be used to compute inverse Hadamard transformation by multiplying the result by \( (2/N)^2 \). Similarly, we can prove (using Equation 5.5.16) that the data-flow system of Figure 5.4.22 can be used to compute the inverse Fourier transformation by multiplying the result by \( (2/N)^2 \). In
Figure 5.5.18: Data Flow configuration for the inverse Fourier transformation (merging case). *RESULT divides its inputs by $N^2$. 
software simulation (Appendix E), a TRANSMIT boolean value is used to specify the mode the program is to execute (transmit or receive).

5.6 Comparison of Algorithms [8,19,21]

In this section, we would like to evaluate the number of computations each of the previous systems takes. However, simultaneous computations are not added, otherwise we would not be having a parallel processing system.

It has been shown (Appendix A) that the one-dimensional FFT (fast Fourier transform) requires \( N \log_2 N \) complex multiplications and additions and the one-dimensional FHT (fast Hadamard transform) requires \( N \log_2 N \) additions or subtractions. Appendix A describes how we can use an algorithm to compute two-dimensional FFT from the one-dimensional FFT. This algorithm requires \( N^2 \log_2 N^2 \) computations (\( N \) is the matrix size). Following the same approach, the two-dimensional FHT requires \( N^2 \log_2 N^2 \) additions or subtractions.

In the data-flow systems discussed earlier in this chapter, we would like to use the fast implementation of the discrete transforms to evaluate the number of arithmetic operations (real or complex). Since we subdivide the image into four quadrants, the number of complex computations (done concurrently) required by the FOURIER TRANSFORM processors is \( (N/2)^2 \log_2 (N/2)^2 \) (non-merging case). Similarly, for the HADAMARD TRANSFORM processors (non-merging case), \( (N/2)^2 \log_2 (N/2)^2 \) real additions or subtractions are required. For the non-merging case, the same number of computations is required by the transform
processors Fourier and Hadamard since in the Fourier case four sub-processors are executing concurrently as already mentioned. The mergers in each case form two-stage pipelines that add or subtract matrices of size N/2. Each stage does \((N/2)^2\) concurrent computations. The two-stage pipeline executes in a time proportional to \(2(N/2)^2\) computations. Table 5.6.1 summarizes the number of concurrent computation of each case.

<table>
<thead>
<tr>
<th></th>
<th>TRANSFORM PROCESSORS</th>
<th>MERGER PROCESSORS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOURIER (merging)</td>
<td>((N/2)^2 \log(N/2)^2) complex additions and multiplications</td>
<td>(N^2/2) Complex additions</td>
</tr>
<tr>
<td>HADAMARD (merging)</td>
<td>((N/2)^2 \log(N/2)^2) real additions and substractions</td>
<td>(N^2/2) real additions or substractions</td>
</tr>
<tr>
<td>FOURIER (non-merging)</td>
<td>((N/2)^2 \log(N/2)^2) complex additions and multiplications</td>
<td>no-merging</td>
</tr>
<tr>
<td>HADAMARD (non-merging)</td>
<td>((N/2)^2 \log(N/2)^2) real additions and substractions</td>
<td>no-merging</td>
</tr>
</tbody>
</table>

Table 5.6.1 Approximate Number of Arithmetic Operations (Real or Complex) Required for Fast Implementation of Various Data-Flow Graphs.

5.7 Comparison Via Number of Processors Used

In this section, we will compare the time saved when we use an array of 2x2, 4x4 and 8x8 processors, to transform-code an image P, via one processor. To simplify the problem let's consider the non-merging case. More specifically, we will be concerned with the
Fourier transformation knowing that the same discussion is applicable to the Hadamard transformation.

To transform-code (Fourier transformation) an NxN image, \( N^2 \log_2 N^2 \) computations are required using one processor. If we use an array of 2x2 processors, then \( (N/2)^2 \log_2 (N/2)^2 \) computations are executed concurrently.

The percentage of time saved using an array of 2x2 processors via one processor is given by

\[
A = \left( 1 - \frac{(N/2)^2 \log_2 (N/2)^2}{N^2 \log_2 N^2} \right) \times 100\% \quad (5.7.1)
\]

Simplifying this equation, we get

\[
A = \left( 1 - \frac{1}{4} \frac{L_n(N/2)}{L_n N} \right) \times 100\% \quad (5.7.2)
\]

The limit of the above expression as \( N \) goes to infinity is

\[
\lim_{N \to \infty} \left( 1 - \frac{1}{4} \frac{L_n(N/2)}{L_n N} \right) \times 100\%
\]

\[
= \left( 1 - \left( \frac{1}{4} \right) (1) \right) \times 100\% = 75\%
\]

Therefore, the maximum percentage of time saved using four processors via one is 75\% (this, of course, is for an image of infinite size).

If an array of 4x4 processors is used, then \( (N/4)^2 \log_2 (N/4)^2 \) computations are executed concurrently. The percentage of time saved using an array of 4x4 processors via one processor is given by
The limit of \( B \) as \( N \) goes to infinity is

\[
B = \left( 1 - \frac{(N/4)^2 \log_2(N/4)^2}{N^2 \log_2 N^2} \right) \times 100\% \quad (5.7.3)
\]

More generally, if an array of \( M \times M \) processors (\( M \) is a power of 2) are concurrently transforming an image \( P \) of size \( N \times N \), then the maximum percentage of time saved using the \( M \) processors via one processor is given by

\[
C = \left( 1 - \frac{1}{M^2} \right) \times 100\% \quad (5.7.4)
\]

Table 5.7.5 shows the time taken to transform an image of different sizes using 1, 4, 16 or 64 processors, assuming that each computation takes \( T \) seconds on the average to be executed. The graph of Figure 5.7.6(a) shows the percentage of time saved when using 4, 16 and 64 processors via one processor for an image of different sizes. The formulae used are 5.7.1, 5.7.3 and 5.7.7 (shown below), respectively.

\[
D = \left( 1 - \frac{(N/8)^2 \log_2(N/8)^2}{N^2 \log_2 N^2} \right) \times 100\% \quad (5.7.7)
\]

Figure 5.7.6(b) shows the percentage of time it takes 4, 16 and 64 processors to compute the transformation via the time taken by one processor.
Figure 5.7.6. (a) Percentage of time saved and (b) Percentage of time consumed when using 4, 16 or 64 processors via I.
Figure 5.7.9: Time via Image Size for 1, 4, 16 and 64 Processors.
Table 5.7.5: shows the time taken (in function of 1 multiplication time T) to transform-code an image using a) 1 processor, b) 4 processors, c) 16 processors and d) 64 processors.

<table>
<thead>
<tr>
<th>SIZE</th>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>20971520 T</td>
</tr>
<tr>
<td>512</td>
<td>4718592 T</td>
</tr>
<tr>
<td>256</td>
<td>1048576 T</td>
</tr>
<tr>
<td>128</td>
<td>229376 T</td>
</tr>
<tr>
<td>64</td>
<td>49152 T</td>
</tr>
<tr>
<td>32</td>
<td>10240 T</td>
</tr>
<tr>
<td>16</td>
<td>2048 T</td>
</tr>
<tr>
<td>8</td>
<td>384 T</td>
</tr>
</tbody>
</table>

Table 5.7.8: Percentages used in graph 5.7.6 (a) in tabular form.

<table>
<thead>
<tr>
<th>SIZE</th>
<th>%TIME SAVED</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>77.5</td>
</tr>
<tr>
<td>512</td>
<td>77.77</td>
</tr>
<tr>
<td>256</td>
<td>78.125</td>
</tr>
<tr>
<td>128</td>
<td>78.57</td>
</tr>
<tr>
<td>64</td>
<td>79.16</td>
</tr>
<tr>
<td>32</td>
<td>80.00</td>
</tr>
<tr>
<td>16</td>
<td>81.25</td>
</tr>
<tr>
<td>8</td>
<td>83.33</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SIZE</th>
<th>%TIME SAVED</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>95.00</td>
</tr>
<tr>
<td>512</td>
<td>95.138</td>
</tr>
<tr>
<td>256</td>
<td>95.30</td>
</tr>
<tr>
<td>128</td>
<td>95.53</td>
</tr>
<tr>
<td>64</td>
<td>95.83</td>
</tr>
<tr>
<td>32</td>
<td>96.25</td>
</tr>
<tr>
<td>16</td>
<td>96.875</td>
</tr>
<tr>
<td>8</td>
<td>97.916</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SIZE</th>
<th>%TIME SAVED</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>98.90</td>
</tr>
<tr>
<td>512</td>
<td>98.95</td>
</tr>
<tr>
<td>256</td>
<td>99.02</td>
</tr>
<tr>
<td>128</td>
<td>99.10</td>
</tr>
<tr>
<td>64</td>
<td>99.21</td>
</tr>
<tr>
<td>32</td>
<td>99.375</td>
</tr>
<tr>
<td>16</td>
<td>99.809</td>
</tr>
<tr>
<td>8</td>
<td>100.00</td>
</tr>
</tbody>
</table>
From those graphs we can see that a substantial amount of time is saved using the parallel scheme of transformation. This percentage (refer to Figure 5.7.6(a)) is higher for lower image sizes and drops for lower values as the image size increases. Next, we notice that for higher sizes of the image, the percentage of time saved slightly varies (especially for a higher number of processors, i.e., 64 processors case). Another interesting result is that, using 64 processors we save 98% for an image of size 1024 x 1024 while we save 95% using 16 processors. This difference (3%) is not much knowing that we are now paying for cost of 64 processors instead of 16. This difference is more attractive when we compare the four processors case via the 16 processor case. In fact, for a 1024 x 1024, 95% is saved using 16 processors via 77.5% using 4 processors. The difference is 17.5%. Therefore, we conclude that the 16 processors scheme is the optimum choice considering both time and cost. The 64 processors case is not very attractive due to its high cost and slight improvement in the time factor (compared to the 16 processors case). The four processors case saves a substantial amount of time (about 78%) at a very low cost. Table 5.7.8 shows the percentages in the graph of Figure 5.7.6(a) in tabular form. Figure 5.7.9 shows the graph of time via image size for 1, 4, 16 and 64 processors.
5.8 Data-Flow Configuration for Processing/Transmitting Hierarchically Segmented Images

5.8.1 Introduction

In this section we will combine the idea of regular decomposition presented in Chapter 4 with the data-flow systems presented earlier in this chapter. The overall structure of the system under consideration has the block diagram of Figure 5.8.1. First, the image is sensed and digitized in a fast manner (as mentioned in Chapter 3) to generate the hierarchical (tree) representation of it. Then, the regular decomposition algorithm described in Chapter 4 is applied to obtained a reduced quadtree structure of the image. At this point, the original amount of picture data has been compressed by not retaining non-informative areas of the picture. The next step is to perform transform (Fourier or Hadamard) coding on the regularly-decomposed quadtree structured picture. If the picture is then to be transmitted, this transform coding represents an averaging or smearing

![Figure 5.8.1 The Block Diagram of a System to Process/Transmit Regularly-Decomposed Transform-Coded Images](image-url)
process that concentrates the transformed components' energy near the origin, thus decorrelating the picture elements and allowing further reduction of the contained information bandwidth through regular 2-D filtering techniques.

5.8.2 System Description

The transform coding block of Figure 5.8.1 is any of the data-flow system described earlier in this chapter. In other words, for any informative node detected in the regular decomposition algorithm, the corresponding region that node represents will be transform-coded using the parallel scheme of Figures 5.4.7, 5.4.17, 5.4.21 or 5.4.22. In this section, the discussion is applicable to any of those systems. For the purpose of implementation, we will be considering the system of Figure 5.4.21 (non-merging Hadamard transformation) since we have seen that the small number of computations involved (Section 5.6) is quite attractive. At this stage, we will consider that our system has as an input the full blown tree structure representing the image in hierarchical form.

The data-flow system in question is shown in Figure 5.8.2. The main advantage of such a system is primarily due to its ability of carrying out simultaneously the two operations: 1) applying the regular decomposition algorithm and forming the reduced quadtree representation of the image, and 2) executing transform coding only on the newly added "very informative" quadtree nodes for every successive tree level as they are generated by the regular decomposition.
Figure 5.8.2: Data flow configuration for transform coding/transmitting regularly-decomposed images.
procedure. While the decomposition algorithm proceeds to refine "not sure" nodes to identify the "very informative" quadtree nodes of the next level, the already transform-coded "very informative" nodes of the previous level can be transmitted along with a small header containing the respective node's level and position on the quadtree (which identifies the topological position of a subquadrant in the picture along with its size). Quadtree nodes for which further subdivision has stopped (because they were labelled "non-informative"), are not transform-coded nor transmitted. It is obvious that the transform (Hadamard) matrices are of variable size depending upon the particular quadtree level on which they are operating. Due to the structure of the quadtree representation, each node of a subsequent level represents a subpicture of size $m/2 \times m/2$ if $m \times m$ is the size of the parental node picture.

The mechanism of operation of the proposed system is as follows. Regular Decomposition processor takes as an input the tree structure representing the image. Then it applies the regular decomposition algorithm to that tree and once an informative node is detected, its corresponding geographical region or quadrant is routed to the Quadrant Subdivider processor along with its size. At the same time Regular Decomposition processor sends half of the quadrant size allocated to Hadamard Generator processor. At this time, while Quadrant Subdivider would be subdividing the quadrant into four, the Hadamard Generator processor would be generating the proper size Hadamard matrix. At the next stage, the Hadamard Transform processors
will transform-code each of the subquadrants allocated and those results are then routed to RESULT processor where the transform coefficients will be serialized and sent to the transmitter. If a non-informative node is detected, then it is skipped and its sons, "grandsons", . . . are never visited (refer to Chapter 3). A not sure node is further decomposed and so on.

In this scheme, it is obvious that the decomposition procedure is not allowed to go to the last level, since we cannot subdivide one pixel into four. Therefore, the maximum allowable level of decomposition is the one before the last. Therefore, we should specify this level to the REGULAR DECOMPOSITION processor. Since we may decompose different types of images, theta1 and theta2 need to be specified. In some cases, starting at level 0, applying regular decomposition in the tree results in chopping off a piece of the object. Therefore, the starting level is also to be specified as seen in Figure 5.8.2.

Hence REGULAR DECOMPOSITION processor, besides decomposing the tree, acts as a system controller. Once it detects an informative node, it triggers the operation of all other processors. Therefore, while HADAMARD TRANSFORM processors would be concurrently transform-coding the four subquadrants of a quadrant, QUADRANT SUBDIVIDER may be subdividing another quadrant and REGULAR DECOMPOSITION processor may be detecting another informative node and so on. If at a certain time an informative node is detected and at the same time QUADRANT SUBDIVIDER is busy, then this processor does not receive the new quadrant until it has sent its output to the subsequent processors.
Therefore, the system is asynchronous and self-synchronized. This concept is literally applied in the PASCAL PLUS simulation (Appendix F) of the data-flow chart of Figure 5.8.2.

5.8.3 Example

Let's consider the 32 x 32 T object shown in Figure 5.8.3(a). Applying the regular decomposition algorithm, we reject all the non-informative area surrounding that object as shown in Figure 5.8.3(b). The reduced tree structure of this image (for $\theta_1 = 0.25$ and $\theta_2 = 0.75$) is shown in Figure 5.8.3(c). The maximum level of decomposition considered is level three. The number of decompositions it takes to decompose the image (stopping at level three) is eleven. From Figure 5.8.3(b) we can see that there are 12 quadrants (each of size $4 \times 4$) to be transform-coded. The number of computations required to do this, using the parallel scheme shown in Figure 5.8.2, is

$$(4/2)^2 \cdot \log_2 (4/2)^2 \cdot (12) = 96 \text{ computations}$$

The time required to do those computations is $96 T_1$ (where $T_1$ is the time it takes the processor to do one of those computations). The time required to decompose the tree is $11 T_2$ (where $T_2$ is the time it takes the decomposition processor to do one decomposition).

Each decomposition is basically equivalent to four divisions and four comparisons. For the purpose of implementation, let's assume that $T_2$ is approximately equal to $6 T_1$. Therefore, the overall time
Figure 5.8.3: (a) Original image and (b) what remains of it after regular decomposition. (Shadowed area represents non-informative quadrants.)
it takes the system of Figure 5.8.2 to transform-code the hierarchically-decomposed image of Figure 5.8.3(a) is roughly

$$96 T_1 + 11 \cdot (6) \cdot T_1 = 162 T_1$$

If we were to transform-code the whole image using one processor and without applying the regular decomposition algorithm, the time consumed would be

$$\left(32\right)^2 \log_2 \left(32\right)^2 \cdot T_1 = 10240 T_1$$

Therefore, the time saved by this efficient system is about

$$(1 - (162/10240)) \times 100\% = 98.41\%.$$ In other words, it is a substantial reduction in the time factor knowing that each transformed quadrant is transmitted right away instead of having to wait for the overall image to be transformed in order to send it to the transmitter.

Another benefit of this system is the data reduction and hence transmission bandwidth. The number of pixels to be transmitted are
4 \times 4 \times 12 = 192 \text{ pixels instead of 1024 pixels. Therefore, the data is reduced by 81.25\%}.

5.8.4 Conclusion

We have seen that the parallel system presented in the previous section has the following advantages:

1. The overall data is reduced since only "very informative" areas of the picture are transform-coded and transmitted (non-informative areas are neither transformed nor transmitted).

2. Each transform-coded picture quadrant (or tree node) is independent of the others, so that they may be processed in parallel.

3. Hadamard transformation time is reduced, since smaller dimension Hadamard matrices are simultaneously applied only to "very informative" nodes of a tree level.

4. This technique can easily lead to the design of fast transmitters with parallel hardware implementations, whereby the coding and transmitting of a picture proceeds in parallel with its compression through the regular decomposition process.
6.1 Gray Level Images

In this section, we will experiment on different positions of an object in the image for values of \( \theta_1 \) and \( \theta_2 \). Let's consider the cross object shown in Figure 6.1.1(a). This image is of size 32 x 32 and the cross is centered at the middle of the image. The black area representing the cross has all its pixels of average intensity equal to 26. The lighter area around the cross consists of pixels of average intensity equal to 9. The light background has pixels of average intensity equal to 1 (Figure 6.1.1(c)). This position of the object is characterized by the displacement \( x = 9, y = 9 \) (first pixel number of the middle area). Applying the regular decomposition algorithm, for \( \theta_1 = 0.20 \) and \( \theta_2 = 0.90 \), we get the image shown in Figure 6.1.1(b). We notice that all the redundant area around the cross object has been identified as non-informative and, therefore, rejected. The number of decompositions required by the regular decomposition algorithm to extract the cross from the image is 21. It is this number that concerns us in this experimentation. In fact, we are looking for the object displacement at which there is the minimal number of decompositions. This is equivalent to a minimal quadtree. The minimal quadtree enables us to transmit regularly-decomposed images faster since less number of decompositions is required. Besides, if we were to store the
Figure 6.1.1: (a) Original image and (b) what remains of it after regular decomposition. (Shadowed area represents non-informative quadrants.)
Figure 6.1.1 (c): 32 X 32 image representing the image in figure 6.1.1 (a) showing the average intensity of each pixel.
informative nodes of a quadtree, then the minimal quadtree yields minimal storage.

The same object is now displaced by two pixels up to the y-axis as shown in Figure 6.1.2(a). Its displacement is described by \((x = 9, y = 7)\). The reduced representation of this image is shown in Figure 6.1.2(b). For this displacement, experimentation showed that in order not to chop off any area of the object \(\theta_1\) should be as low as 0.15 (\(\theta_2\) is still 0.9). This value of \(\theta_1\) adds some redundancy to the object (as shown in the figure) of eight \(2 \times 2\) quadrants or \((8 \times 4)/1024 = 3.125\%\) of the whole image. The number of decompositions required to decompose this image is 25. This position of the object has two different symmetric positions:

\[(x = 9, y = 9)\]
\[(x = 7, y = 9)\]

At those positions, we will obtain the same result by symmetry. Therefore, we notice that a small value of \(\theta_1\) insures extracting the informative area we are looking for but could result in adding up redundant or non-informative area to the object to be extracted. The redundant area added is very slight (3.125%) compared to the whole \(32 \times 32\) image.

Next we consider that the object is placed in the upper left quadrant of the image. This displacement is described by \(x = 1, y = 1\) (refer to Figure 6.1.3(a)). The symmetrical positions are:

\[x = 1, y = 17;\]
\[x = 17, y = 1;\]
\[x = 17, y = 17;\]
Figure 6.1.2: (a) Original image and (b) what remains of it after regular decomposition. (Shadowed area represents non-informative quadrants.)
Figure 6.1.3: (a) Original image and (b) what remains of it after regular decomposition. (Shadowed area represents non-informative quadrants.)
the number of decompositions required is 17. In this position, no area of the object has been chopped off and no redundant area has been added to the object. The threshold values used are $\theta_1 = 0.20$ and $\theta_2 = 0.90$. We notice that the first decomposition (father at level 0) rejects the three other quadrant (upper right, lower left, lower right). This way, the decomposition procedure proceeds by refining the upper left quadrant (Figure 6.1.3(b)) only.

Let's place the object at $x = 3$, $y = 3$ as shown in Figure 6.1.4(a). This displacement is symmetrical with respect to the $x = y$ axis and has most of its intensity concentrated in the upper left quadrant. The symmetrical positions for this displacement are at:

- $x = 3$, $y = 15$
- $x = 15$, $y = 3$
- $x = 15$, $y = 15$

The number of decompositions required is 28. No area has been chopped off from the object. There are 18 quadrants, of size $2 \times 2$ each, added to the object. In other words $(18 \times 4)/1024$ or 7.03% of the image has been added to the object. Since the image has two edges in the upper right and lower left quadrants of the image, it can be easily chopped off. A value of $\theta_1$ as low as 0.12 insures that no edge will be chopped off. $\theta_2$ is still 0.9 (refer to Figure 6.1.4(b)). If we were to apply the regular decomposition algorithm on the same image but changing $\theta_1$ to 0.20 (Figure 6.1.5(a)), the edges in the upper right and lower left quadrants are chopped off (Figure 6.1.5(b)). This is $4 \times 4/1024$ or
Figure 6.1.4: (a) original image and (b) what remains of it after regular decomposition. (Shadowed area represent non-informative quadrants.)
Figure 6.1.5: (a) Original image and (b) What remains of it after regular decomposition. (shadowed area represents non-informative quadrants.)
1.56% of the whole image. However, no redundant area is now added to the object and the number of decompositions is 12.

Figure 6.1.6(a) shows the same object at a displacement of $x = 7$, $y = 7$. The object is symmetrical with respect to the $x = y$ axis. The symmetrical positions to this displacement are:

- $x = 11$, $y = 7$
- $x = 11$, $y = 11$
- $x = 7$, $y = 11$

If we use $\theta_1 = 0.20$ and $\theta_2 = 0.90$, then the number of decompositions required is 14 and four ($2 \times 2$) quadrants are chopped off (Figure 6.1.6(b)). This is $(4 \times 4)/1024$ or 1.56%. To avoid this situation, we should select a smaller value of $\theta_1$ (as shown in the previous example). This may, however, increase the number of decompositions and add up redundant area to the object.

In the case of symmetrical positions, the regular decomposition algorithm performs the same number of decompositions, extracts the same object (i.e., same amount of area is chopped off or added up to the object). Figure 6.1.7(a) shows the object displaced such as $x = 15$, $y = 3$. This displacement is symmetrical, with respect to the $y$-axis (at the center of the image), to the object displacement of Figure 6.1.4(a). Figure 6.1.7(b) shows how the object is refined. A look at this figure shows that it is also symmetrical (w.r.t. the same axis) to the one in Figure 6.1.4(b). This tells us that the minimal quadtree representation of an image has the same size and a symmetrical
Figure 6.1.6: (a) Original image and (b) what remains of it after regular decomposition. (Shadowed area represents non-informative quadrants.)
Figure 6.1.7: (a) Original image and (b) what remains of it after regular decomposition. (Shadowed area represents non-informative quadrants.)
Conclusion

From the above discussion, we conclude that the minimal quadtree representation of an image if greatly dependent on the object displacement in the image. Since a minimal quadtree for a certain image has also a minimum number of decompositions, this number was used as a criterion to check for the minimal quadtree. From the previous experimentation and for this particular shape of the object, it seems that placing the object in one of the quadrants (upper left, upper right, lower left or lower right) leads to a minimal number of decompositions and hence a minimal quadtree. Besides, we can conclude that in order not to chop off any informative quadrant in the image a small value of $\theta_1$ should be used. All not sure nodes of the last decomposition level are considered as informative.

6.2 Binary Images

In binary images (such as text, black and white images, . . . ), the intensity values are either 1 or 0. In this section, the object will be displaced on different axis in order to find the axis where we have minimal quadtrees. For this we consider three different shapes of the object: one totally symmetric (cross shape) shown in Figure 6.2.1(a), another symmetric as shown in Figure 6.2.1(b) and the third non-symmetrical (Figure 6.2.1(c)). The three objects are shown
a) Totally symmetric object

b) Dominantly symmetric object
Figure 6.2.1 a), b), c) show three (32x32) binary images centered at the middle (x=9, y=9). d) show the different displacements considered.
centered at the middle. The position is such as $x = 9, y = 9$ taking as reference the first pixel of the $16 \times 16$ quadrant containing the object. Figure 6.2.1(d) shows the different coordinates at which the object will be displaced on the five axes shown in the figure. The reference as already mentioned is the upper left corner of the quadrant containing the object.

Each of the tree objects will be shrinked in size and the same procedure is applied to each size. Therefore, we have three objects with three different sizes, each displaced vertically along five axes and five positions on each axis. The positions in consideration are tabulated in Table 6.2.2.

<table>
<thead>
<tr>
<th>First axis</th>
<th>Second axis</th>
<th>Third axis</th>
<th>Fourth axis</th>
<th>Fifth axis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>3.1</td>
<td>5.1</td>
<td>7.1</td>
<td>9.1</td>
</tr>
<tr>
<td>1.3</td>
<td>3.3</td>
<td>5.3</td>
<td>7.3</td>
<td>9.3</td>
</tr>
<tr>
<td>1.5</td>
<td>3.5</td>
<td>5.5</td>
<td>7.5</td>
<td>9.5</td>
</tr>
<tr>
<td>1.7</td>
<td>3.7</td>
<td>5.7</td>
<td>7.7</td>
<td>9.7</td>
</tr>
<tr>
<td>1.9</td>
<td>3.9</td>
<td>5.9</td>
<td>7.9</td>
<td>9.9</td>
</tr>
</tbody>
</table>

Table 6.2.2 This Table Shows the Different $x$-$y$ Displacements of the Object.
Each object and its three sizes are shown in Figures 6.2.3, 6.2.4 and 6.2.5 in reduced size. The minimum number of decompositions is shown in tabular forms. In each table, the average number of decompositions is given (refer to Tables 6.2.6, 6.2.7 and 6.2.8).

In the binary images, the threshold values taken are $\theta_1 = 0$ and $\theta_2 = 1$. This way we will refine the object and reject all the zero area. Therefore, in binary images no redundancy is added to the object. Besides nothing is chopped off from the object.

For the large size cross of Figure 6.2.3(a), we notice that the axis $x = 3$ and $x = 9$ have the same number of decompositions (refer to Figure 6.2.6(a)). This means that we will obtain the same quadtrees by moving the object on either one of these axes. The minimal quadtrees are on the $x = 1$ axis since we have the minimum average (20) of decompositions on this axis (Table 6.2.6(a)).

For the medium size cross of Figure 6.2.3(b), there is no axis where we get the same size of quadtrees as previously. The minimal quadtrees are on the $x = 3$ axis and the average of decompositions on this axis is 13. We also notice that all the averages are less than 20 (previous minimum average). The higher average is 19.2 (Table 6.2.6(b)).

For the small size cross of Figure 6.2.3(c), the minimal quadtrees exist on the $x = 5$ axis with an average of 9 (Table 6.2.6(c)). The highest average is 12.4 which is less than 19.2 of the medium size cross.
Figure 6.2.3: Totally symmetric object (cross shape) of 3 different sizes

Figure 6.2.4: Dominantly symmetric object (T shape) of 3 different sizes

Figure 6.2.5: Non symmetric object of 3 different sizes
Table 6.2.6 Number of decompositions for the fully symmetrical object.
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td>23</td>
<td>19</td>
<td>23</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>23</td>
<td>25</td>
<td>17</td>
<td>22</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>17</td>
<td>26</td>
<td>19</td>
<td>23</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td>25</td>
<td>17</td>
<td>22</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>17</td>
<td>25</td>
<td>21</td>
<td>25</td>
<td>19</td>
</tr>
<tr>
<td>AVG</td>
<td>17.62</td>
<td>24.81</td>
<td>18.62</td>
<td>23.01</td>
<td>17.2</td>
</tr>
</tbody>
</table>

(a) Large object

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td>21</td>
<td>19</td>
<td>19</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>16</td>
<td>13</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>17</td>
<td>24</td>
<td>19</td>
<td>20</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>18</td>
<td>14</td>
<td>15</td>
<td>13</td>
</tr>
<tr>
<td>9</td>
<td>17</td>
<td>23</td>
<td>21</td>
<td>21</td>
<td>19</td>
</tr>
<tr>
<td>AVG</td>
<td>14.62</td>
<td>20.41</td>
<td>17.21</td>
<td>17.81</td>
<td>15.8</td>
</tr>
</tbody>
</table>

(b) Medium object

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>17</td>
<td>13</td>
<td>15</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>16</td>
<td>11</td>
<td>14</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>14</td>
<td>11</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>18</td>
<td>12</td>
<td>15</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>13</td>
<td>19</td>
<td>15</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>AVG</td>
<td>11.21</td>
<td>16.81</td>
<td>12.41</td>
<td>14.81</td>
<td>12.4</td>
</tr>
</tbody>
</table>

(c) Small object

Table 6.27: Number of decompositions for the dominantly symmetric object
Table 6.2.8: Number of decompositions for the non symmetric object in three sizes.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td>21</td>
<td>18</td>
<td>21</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>17</td>
<td>13</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>20</td>
<td>16</td>
<td>19</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>17</td>
<td>13</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>23</td>
<td>20</td>
<td>23</td>
<td>18</td>
</tr>
<tr>
<td>AVG</td>
<td>13.4</td>
<td>19.6</td>
<td>16.0</td>
<td>19.0</td>
<td>14.6</td>
</tr>
</tbody>
</table>

(a) Large object

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>14</td>
<td>12</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>18</td>
<td>16</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>12</td>
<td>11</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
</tr>
<tr>
<td>AVG</td>
<td>12.8</td>
<td>16.6</td>
<td>15.4</td>
<td>15.0</td>
<td>14.0</td>
</tr>
</tbody>
</table>

(b) Medium object

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>17</td>
<td>13</td>
<td>15</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>14</td>
<td>10</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>14</td>
<td>11</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>12</td>
<td>9</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>13</td>
<td>19</td>
<td>15</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>AVG</td>
<td>10.4</td>
<td>15.2</td>
<td>11.6</td>
<td>13.6</td>
<td>11.6</td>
</tr>
</tbody>
</table>

(c) Small object
We can conclude that for the symmetrical cross the quadtrees become of smaller size as the size of the object shrinks. Besides, we notice that the axis on which we obtain minimal quadtrees move from left to right ($x = 1, 3, 5$) as the cross size becomes smaller.

Now we consider the $T$ object of Figure 6.2.4. The minimal quadtrees for the large size $T$ (Figure 6.2.4(a)) is on the $x = 9$ axis with an average of 17.2 (Table 6.2.7(a)). The maximum average is 24.8. For the medium size $T$ object of Figure 6.2.4(b), the minimal quadtrees exist on the $x = 1$ axis with an average of 14.6 (Table 6.2.7(b)). The maximum average is now 20.4 on the $x = 3$ axis. We notice that on this axis the quadtrees are (on the average) on larger size than the minimal quadtrees of the large size. The small size $T$ object (Figure 6.2.4(c)) has its minimal quadtrees at $x = 1$ with an average number of decompositions of 11.2. The largest average is 16.8 on the $x = 3$ axis (Table 6.2.7(c)). We also notice that the axis $x = 5$ and $x = 9$ have the same size of quadtrees. Therefore, the object will be decomposed similarly on either axis.

Finally, we consider the non-symmetrical object of Figure 6.2.5. For the large size of this object (Figure 6.2.5(a)), the minimal quadtrees correspond to the $x = 1$ axis with an average number of decompositions of 13.4. The largest average is on the $x = 3$ axis and is 19.6 (Table 6.2.8(a)). The medium size object is Figure 6.2.5(b) has its minimal quadtrees on the $x = 1$ axis with an average of 12.8. The maximum average is on the $x = 3$ axis and is 16.8 (Table 6.2.8(b)). The small size non-symmetrical object (Figure 6.2.5(c)) has its minimal
quadtrees on the $x = 1$ axis with an average number of decompositions of 10.4 (Table 6.2.8(c)). In this type of object, we notice that the minimal quadtrees are on the $x = 1$ axis and the maximal quadtrees are on the $x = 3$ axis, independently of the object size. We also notice that there are two axis ($x = 5$ and $x = 9$) on which we obtain quadtrees of the same size.

Conclusion: The purpose of obtaining the minimal quadtrees is to reduce the memory allocation occupied by such a data structure. This is more eminent when we have a 1024 x 1024 image stored in the form of a quadtree. Then only informative nodes are stored reducing thus the size of this structure. In the scheme discussed in Chapter 5 where we transform-code hierarchically-decomposed images, the minimal quadtree leads to a fast decomposition time, and hence a more efficient system.
Chapter 7

CONCLUSION

In this thesis, we discussed the use of data-flow architecture to effectively handle real-time image processing/transmission application. The recent advances in VLSI technology allow the use of a modular approach to manufacturing such parallel and adaptable configurations.

More specifically, we discussed the transform-coding algorithm considering two types of orthogonal transformations: the Walsh-Hadamard (often referred to as Hadamard) and the Fourier transforms. Transform coding of an image exhibits the property of concentrating the energy of the original image (which is usually uniformly distributed in the spacial domain) near the origin of the transform domain. Besides, if few of the transform domain coefficients are of large magnitude, then, due to the energy conservation property between the original image spacial domain and the transform domain, many of the remaining coefficients would be of low magnitude and may, therefore, be discarded for bandwidth reduction purposes.

The main theme we followed throughout the thesis was that of subdividing the image into a number of subquadrants (or regions). Four processors were assigned to each of the four quadrants of the image to concurrently transform-code those quadrants providing the system with much higher throughput capabilities. We also discussed the possibility of having more than four processors concurrently
transforming the image. The choice of having only four processors seems to be a reasonable choice since it saves a substantial amount of time (77.5% for a 1024 x 1024 image) at a relatively cheap cost of four processors. Systems for inverse transform at the receiver end were also discussed and we showed that such systems can be obtained from the data-flow configurations used at the transmitter end.

Another theme discussed in this thesis was that of filtering out non-interesting quadrants (or regions). We introduced, therefore, the hierarchical image representation (quadtree) for a number of reasons discussed in the text, including the reason that such a representation preserves the geometric relationship among pixels. The hierarchical regular decomposition technique was presented in order to give a "reduced" tree data structure of the image by eliminating "non-informative" regions of the picture and further refining the tree in order to extract "very informative" quadrants. Using this reduced hierarchical representation, we concentrated primarily on performing the transform coding used in image processing and transmission. The data-flow architecture was used to perform the three concurrent operations: decomposing the image, transform-coding "informative" quadrants and transmitting those transformed quadrants.

Finally, we experimented on a set of objects with different sizes trying to locate the object in the image at displacement where it has a minimal quadtree representation. The minimal quadtree is responsible for less memory storage and fast decomposition time.
For all the concurrent algorithm described in this thesis, a simulation in concurrent PASCAL (PASCAL PLUS) was given to strongly implement the idea. In the future, we may expect that such a parallel scheme would be implemented on supercomputers, using a concurrent programming language.
REFERENCES


Appendix A

FAST TRANSFORMS [8,19]

We provide here a brief review of fast transformations emphasizing basically the fast Fourier transform (FFT). A similar discussion is applicable to the fast Hadamard transform. Our starting point is the equation for the one-dimensional Fourier transform:

\[ F(u) = \sum_{k=0}^{N-1} f(k) \exp(-j \frac{2\pi}{N} uk) \quad 0 \leq u \leq N-1 \quad (A.0) \]

Replacing \( \exp(-j \frac{2\pi}{N}) \) by \( z \) in the above equation, we get

\[ F(u) = \sum_{k=0}^{N-1} f(k) z^{uk} \quad 0 \leq u \leq N-1 \quad (A.1) \]

We introduce a new index of summation, \( m \), which varies from 0 to \((N/2)-1\) and which is related to \( k \) by the following equations:

\[ k = 2m \quad \text{for } k \text{ even} \quad (A.2a) \]

\[ k = 2m + 1 \quad \text{for } k \text{ odd} \quad (A.2b) \]

Then Equation (A.1) can be rewritten as

\[ F(u) = \sum_{m=0}^{(N/2)-1} [f(2m)z^{2mn} + f(2m+1)z^{(2m+1)u}] \quad (A.3) \]

Let us define \( M = N/2 \) and

\[ g = \exp(-j \frac{2\pi}{M}) \]

133
and observe that $z^2 = g$. Then Equation (A.3) can be written as

$$F(u) = \sum_{m=0}^{M-1} f_e(m)g^{um} + z^u \sum_{m=0}^{M-1} f_0(m)g^{um} \quad (A.4)$$

where $f_e$ and $f_0$ are two new functions obtained by considering separately the points for even and odd values of the argument. The two sums in Equation (A.4) look very much like Fourier transforms of functions with $M$ points except that $u$ varies from 0 to $N-1$. However, we notice that

$$F(u+N) = F(u) \quad (A.5)$$

since $\exp(-jN\frac{2\pi u}{N}) = 1$. Thus, the $M$ values of the Fourier transform of $f_e$ (or $f_0$) can be used to find $N$ values trivially. Therefore, we have the following recursive equation for the Fourier transform;

$$F(u) = F_e(u) + \exp(-j\frac{2\pi u}{N}) F_0(u) \quad (A.6)$$

Equation (A.6) is the foundation of the fast Fourier transform. First, it provides an easily programmable method for the computation. This is given as Algorithm A.1.

Algorithm A.1. Fast Fourier Transform

Procedure FFT(Nf)

1. If N equals 2, then do:
   
   Begin.
   
   2. Replace $f(0)$ by $f(0) + f(1)$ and $f(1)$ by $f(0) - f(1)$.
   
   3. Return.
   
   End.
4. Else do:
    Begin.
5. Define g as consisting of all points of f which have an even index and h as consisting of the remaining points.
6. Call procedure FFT(N/2, g).
7. Call procedure FFT(N/2, h).
8. Replace \( f(i) \) by \( g(i) + \exp(-j2 \cdot i/N) \cdot h(i) \) for \( i = 0 \) to \( N-1 \).
    End.

The procedure calls itself if \( N \) exceeds 2, while for \( N = 2 \) it uses the formula:

\[
F(0) = f(0) + f(1) \quad (A.7a)
\]
\[
F(1) = f(0) - f(1) \quad (A.7b)
\]

Clearly, the successive halving of \( N \) can proceed smoothly only if \( N \) is a power of 2 and it is only for such values that there is a simple algorithm for the fast Fourier transform. Second, Equation (A.6) can be used to calculate the cost of the computation. Let \( C(N) \) be the cost for \( N \) points. After the evaluation of the two transforms in the right-hand side, Equation (A.6) requires effort proportional to \( N \) because of the multiplication of the terms by the exponential and the subsequent addition. If \( c \) is a constant reflecting the cost of such operation, then we have the following equation for \( C(N) \).

\[
C(N) = 2C\left(\frac{N}{2}\right) + cN \quad (A.8)
\]

Similarly we find that
\[ C \left( \frac{N}{2} \right) = 2C \left( \frac{N}{4} \right) + c \frac{N}{2} \]  

(A.9)

and so forth. Because of Equation (A.7) we know that \( C(2) \) equals the cost of two additions, which we denote by \( c' \). If we write equations similar to Equation (A.9) for \( C \left( \frac{N}{L} \right) \), where \( L = 4, \ldots, N/4 \), multiply each one of them by \( L \), add them to Equation (A.8) and add Equation (A.9) multiplied by two, we find that:

\[ C(N) = \frac{N}{2} c' + cNn \]  

(A.10)

where \( n \) is the number of such equations. In particular \( n \) equals the number of terms in the series 1, 2, 4, \ldots, \( N/4 \) which is equal to the base 2 logarithm of \( N \), \( \log_2 N \), minus 1. Then Equation (A.10) becomes

\[ C(N) = cN \log_2 N + O(N) \]  

(A.11)

where the last expression denotes terms linearly proportional to \( N \).

This shows that only \( N \log_2 N \) operations (complex multiplications and additions) are required for the one-dimensional Fourier transform. Algorithm A.1 can be used to evaluate the two-dimensional transform by finding first the transform of each row of the image, and then the transform of all the columns. This is shown in Algorithm A.2, where FFT\((N,x)\) is a procedure that replaces the \( N \) element array \( x \) by its discrete Fourier transform.

Algorithm A.2. Two-Dimensional FFT

1. For \( l = 0 \) to \( N-1 \) do:
   
   Begin

2. Copy \( f(*,l) \) into the array \( x \).
3. Call FFT(n,x).

4. Replace f(*,l) by x.
   End.

5. For k = 0 to N-1 do:
   Begin.

6. Copy f(k,*) into the array x.

7. Call FFT(N,x).

8. Replace f(k,*) by x.
   End.


Each call to FFT(N,x) entails a computational cost proportional to N log₂N. Since there are 2N such calls, the total cost will be of order N² log₂(N²).

For the fast Hadamard transform, it has been shown [21] that the one-dimensional Hadamard transform of N samples requires N log₂N operations (real additions or subtractions). Similarly, we can use an algorithm similar to Algorithm A.2 to show that the two-dimensional Hadamard transform of an NxN array requires N² log₂N² additions or subtractions.
APPENDIX B

TRANSFORM OF HOMOGENEOUS QUADRANTS

A homogeneous quadrant is one that has the same average intensity all over its pixels. Mathematically, the matrix $A$ is homogeneous if

$$a_{ij} = \text{constant, } i, j.$$

In many images, some quadrants are homogeneous. Very little time should be spent on transform-coding such quadrants. In this appendix, we are to compute the transform (Hadamard or Fourier) of a homogeneous matrix to examine if this matrix provides some fast way to be coded.

In the Hadamard transform case, if the matrix (or quadrant) is such as:

$$A = \begin{bmatrix} a & a \\ a & a \end{bmatrix}$$

then

$$[P] = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} a & a \\ a & a \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 4a & 0 \\ 0 & 0 \end{bmatrix}$$

Hence for a $2 \times 2$ homogeneous matrix, its transform is such as all the matrix elements are zeroes except $A_{11}$ which is $(2)^2 \times a$.

Let's assume this is true for an $N \times N$ image and let's prove recursively that it is true for a $2N \times 2N$ image. If $P_N$ is the $N \times N$ image such as $P_{i,j} = a$ for any $i, j$, then:
\[ [F_N] = [H_N] [P_N] [H_N] \]

or

\[ [F_N] = N^2 \]

\[
\begin{bmatrix}
a & 0 & \ldots & 0 \\
0 & 0 & \ldots & . \\
. & . & & . \\
. & . & & . \\
0 & 0 & \ldots & 0
\end{bmatrix}
\]

\( F_{2N} \) can be written as follows:

\[
[F_{2N}] = \begin{bmatrix} H_N & H_N \\ H_N & -H_N \end{bmatrix} \begin{bmatrix} P_N & P_N \\ P_N & P_N \end{bmatrix} \begin{bmatrix} H_N & H_N \\ H_N & -H_N \end{bmatrix}
\]

\[
= \begin{bmatrix} H_N P_N + H_N P_N & H_N P_N + H_N P_N \\ H_N P_N - H_N P_N & H_N P_N - H_N P_N \end{bmatrix} \begin{bmatrix} H_N & H_N \\ H_N & -H_N \end{bmatrix}
\]

\[ = \begin{bmatrix} 4H_N P_N R & 0 \\ 0 & 0 \end{bmatrix} = (2N)^2 \]

\[
\begin{bmatrix}
a & 0 & \ldots & 0 \\
0 & 0 & \ldots & . \\
. & . & & . \\
. & . & & . \\
0 & 0 & \ldots & 0
\end{bmatrix}
\]

Therefore, for any homogeneous image of size \( N \), the Hadamard transform is a matrix that has all zeroes except its first element \((i = j = 1)\) equal to \( a \cdot N^2 \).
This property is attractive for the reduction of the Hadamard transformation of a homogeneous image.

Now, we consider the Fourier transformation case. The Fourier transform of an image \( P \) is given by \( P' \) such as:

\[
[P'] = [w^{\otimes k}] \cdot [P] \cdot [w^{\otimes k}]
\]

The element \( c_{ij} \), multiplication of \([w^{\otimes k}] \) and \([P] \), is

\[
c_{ij} = \sum_{k=0}^{N-1} a_{ik} b_{kj} = \sum_{k=0}^{N-1} a_{ij} \cdot \alpha = \alpha \sum_{k=0}^{N-1} a_{ik}
\]

\[
= \alpha \sum_{k=0}^{N-1} w^{ik}
\]

We note that \( c_{ij} \) is independent of \( j \). In other words, all the elements in one row are the same. The element \( d_{ij} \) ([\( w^{\otimes k} \) [\( P \) [\( w^{\otimes k} \)]) is:

\[
d_{ij} = \sum_{k=0}^{N-1} c_{ik} \alpha_{kj} = \sum_{k=0}^{N-1} \left( \alpha \sum_{k=0}^{N-1} w^{ik} \right) w^{kj}
\]

\[
= \left( \alpha \sum_{k=0}^{N-1} w^{ik} \right) \left( \sum_{k=0}^{N-1} w^{kj} \right)
\]

Hence, we see the Fourier transform of homogeneous image (or quadrant) does not offer a fast computation scheme such as in the Hadamard transformation case.
APPENDIX C

This appendix has a computer program written in PASCAL PLUS to simulate the data-flow system of Figure 5.4.7. Its purpose is to do concurrent Hadamard transformation on the four quadrants $P_1$, $P_2$, $P_3$ and $P_4$ of an image. The image data is read from an external file named IMGBFER. The size of the image is specified by the constant IMSIZE. Each process is equivalent to a processor in Figure 5.4.7. This equivalence is shown below.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Equivalent Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMAGE SUBDIVIDER</td>
<td>DIVIDER</td>
</tr>
<tr>
<td>HADAMARD GENERATOR</td>
<td>HADMARD</td>
</tr>
<tr>
<td>HADAMARD TRANSFORM1</td>
<td>HADTRNS1</td>
</tr>
<tr>
<td>HADAMARD TRANSFORM2</td>
<td>HADTRNS2</td>
</tr>
<tr>
<td>HADAMARD TRANSFORM3</td>
<td>HADTRNS3</td>
</tr>
<tr>
<td>HADAMARD TRANSFORM4</td>
<td>HADTRNS4</td>
</tr>
<tr>
<td>MERGER1</td>
<td>MERGER1</td>
</tr>
<tr>
<td>MERGER2</td>
<td>MERGER2</td>
</tr>
<tr>
<td>MERGER3</td>
<td>MERGER3</td>
</tr>
<tr>
<td>MERGER4</td>
<td>MERGER4</td>
</tr>
<tr>
<td>RESULT</td>
<td>RESULT</td>
</tr>
</tbody>
</table>

An example is also given. The variable TRANSMIT is set true for Hadamard transformation and false for inverse Hadamard transformation. This transformation is named the merging case since independent quadrant transforms are merged together to obtain the Hadamard transform of the overall image.
PROGRAM PROJ1(IMGBFER,OUTPUT);

CONST  IMSIZE = 16;
        TRANSMIT = TRUE;

TYPE
    M2 = ARRAY[1..16,1..16] OF REAL;
RECTYPE = RECORD
    MAT: M2;
    SIZE: INTEGER
END;

VAR IMGBFER: TEXT;

MONITOR CHAN1 = CHANNEL IN LIBRARY(WHERE TYPE BUFFTYPE = M2);
INSTANCE R1,R2,R3,R4,M1,M2,M3,M4,N1,N2,N3,N4,CHAN1;
INSTANCE N21,N21,N22,N23,N24,N32,N33,N34,CHAN1;
MONITOR CHAN2 = CHANNEL IN LIBRARY(WHERE TYPE BUFFTYPE = RECTYPE);
INSTANCE M3,M22,M3,M4,M11,N12,N13,N14,CHAN2;

PROCESS HadTrans3 IN LIBRARY;
INSTANCE XH3: HadTrans3;
PROCESS HadTrans4 IN LIBRARY;
INSTANCE XH4: HadTrans4;
PROCESS HadTrans1 IN LIBRARY;
INSTANCE XH1: HadTrans1;
PROCESS HadTrans2 IN LIBRARY;
INSTANCE XH2: HadTrans2;
PROCESS MERGER1 IN LIBRARY;
INSTANCE XMER1: MERGER1;
PROCESS MERGER2 IN LIBRARY;
INSTANCE XMER2: MERGER2;
PROCESS MERGER3 IN LIBRARY;
INSTANCE XMER3: MERGER3;
PROCESS MERGER4 IN LIBRARY;
INSTANCE XMER4: MERGER4;
PROCESS RESULT IN LIBRARY;
INSTANCE XRESULT: RESULT;
PROCESS HADMARD IN LIBRARY;
INSTANCE XHADMARD: HADMARD(IMSIZE DIV 2);
PROCESS DIVIDER IN LIBRARY;
INSTANCE XDIVIDER:DIVIDER(IMGBFER,IMSIZE);
BEGIN
        RESET(IMGBFER);
        WRITE('THE IMAGE IS OF SIZE : ');WRITELN(IMSIZE);
        WRITELN; WRITELN;
        ***
END.

MONITOR CHANNEL;
VAR
    BUFF: BUFFTYPE; FULL: BOOLEAN; INSTANCE QUEUE: CONDITION;
PROCEDURE *SEND(LINE: BUFFTYPE);
BEGIN
    IF FULL THEN QUEUE.WAIT;
    BUFF:= LINE; FULL:= TRUE; QUEUE.SIGNAL
END;
PROCEDURE *RECEIVE(VAR LINE:BUFFTYPE);
BEGIN
    IF NOT FULL THEN QUEUE.WAIT;
    LINE:= BUFF; FULL:= FALSE; QUEUE.SIGNAL
END;
BEGIN
    FULL:= FALSE;
    ***
END;
PROCESS DIVIDER(VAR INF:TEXT;DIM: INTEGER);
VAR
  I,J,A: INTEGER; TEMP: REAL;
P1,P2,P3,P4: M2;
BEGIN
  RESET(INF);
  IF TRANSMIT THEN
    WRITELN(' THE IMAGE IS THE FOLLOWING ':)'
  ELSE
    WRITELN(' THE RECEIVED TRANSFORM IS : ');
  A := DIM DIV 2;
  FOR I := 1 TO DIM DO
    BEGIN
      FOR J := 1 TO DIM DO
        BEGIN
          READ(INF, TEMP);
          WRITE(TRUNC(TEMP):3, ' ');
          IF (I<DIM DIV 2) AND (J<DIM DIV 2) THEN
            PI[I,J] := TEMP;
          IF (I<DIM DIV 2) AND (J>DIM DIV 2) THEN
            P2[I,J-A] := TEMP;
          IF (I>DIM DIV 2) AND (J<DIM DIV 2) THEN
            P3[I-A,J] := TEMP;
          IF (I>DIM DIV 2) AND (J>DIM DIV 2) THEN
            P4[I-A,J-A] := TEMP;
        END;
        WRITELN;
      END;
      WRITELN;
      SEND(P1);
      SEND(P2);
      SEND(P3);
      SEND(P4);
    END;
  END;
END;

PROCESS HADMARD(DIM:INTEGER);
VAR
  HADM: RECTYPE;
  I,J, NOFTIMES: INTEGER;
FUNCTION TWO(B:INTEGER):INTEGER;
BEGIN
  TWO:=TRUNC(EXP(B*LN(2)))+0.5
END;
PROCEDURE BUILDERCURSIVELY(NUM, START: INTEGER);
VAR
  I,J,A,B: INTEGER;
BEGIN
  A:=TWO(NUM-1);
  B:=TWO(NUM);
  FOR I:=1 TO A DO
    FOR J:=START TO B DO
      BEGIN
        HADM.MAT[I,J]= HAAM.MAT[I,J-A];
        HADM.MAT[I+A,J-A]= HADM.MAT[I+J-A];
        HADM.MAT[I+A,J]= -HADM.MAT[I+J-A];
      END;
  END;
BEGIN
  WRITELN(' THIS IS THE HADAMARD MATRIX USED ');
  WRITELN;
  NOFTIMES:= TRUNC(LN(DIM)/LN(2)+0.5);
  HADM.MAT[1,1]= 1;
  FOR I:= 1 TO NOFTIMES DO
    BUILDERCURSIVELY(I, TWO(I-1) + 1);
  FOR I:= 1 TO DIM DO
    BEGIN
      FOR J:= 1 TO DIM DO
        WRITE(TRUNC(HADM.MAT[I,J]):2, ' ');
    END;
  HADM.SIZE:= 2*DIM;
  WRITELN;
  SEND(HADM);
  M22.SENIl(HADM);
  M3.SEND(HADM);
  M4.SEND(HADM);
END;
PROCEDURE MULTIPLY(F: R; M: 2; VAR Q: M2); VAR
I, J, K: INTEGER;
BEGIN
FOR I:= 1 TO DIM DIV 2 DO
FOR J:= 1 TO DIM DIV 2 DO
BEGIN
QCI + J3 := 0;
FOR K:= 1 TO DIM DIV 2 DO
END;
END;
BEGIN
MULTIPLY(HADM, MAT, IM, RESULT1);
MULTIPLY(RESULT1, HADM, MAT, RESULT2, MAT);
RESULT2_SIZE := DIM DIV 2;
N11.SEND(RESULT2); N12.SEND(RESULT2);
N13.SEND(RESULT2); N14.SEND(RESULT2)
END;

PROCEDURE MULTIPLY(F: R; M: 2; VAR Q: M2); VAR
I, J, K: INTEGER;
BEGIN
FOR I:= 1 TO DIM DIV 2 DO
FOR J:= 1 TO DIM DIV 2 DO
BEGIN
QCI + J3 := 0;
FOR K:= 1 TO DIM DIV 2 DO
END;
END;
BEGIN
MULTIPLY(HADM, MAT, IM, RESULT1);
MULTIPLY(RESULT1, HADM, MAT, RESULT2, MAT);
RESULT2_SIZE := DIM DIV 2;
N21.SEND(RESULT2, MAT); N22.SEND(RESULT2, MAT);
N23.SEND(RESULT2, MAT); N24.SEND(RESULT2, MAT)
END;
PROCESS HADTRANS3;
VAR
  IM, RESULT1: M2; RESULT2, HADM: RECType;
  DIM: INTEGER;
PROCEDURE MULTIPLY(F, R: M2) VAR G: M2);
VAR
  I, J, K: INTEGER;
BEGIN
  FOR I:= 1 TO DIM DIV 2 DO
    FOR J:= 1 TO DIM DIV 2 DO
      BEGIN
        QCI.JJ:= 0;
        FOR K:= 1 TO DIM DIV 2 DO
          BEGIN
            QCI.JJ:= QCI.JJ+PCI.K*K*K
          END
        END
  END;
BEGIN
  MJ.RECEIVE(HADM);
  RJ.RECEIVE(IM);
  DIM:= HADM.SIZE;
  MULTIPLY(HADM.MAT, IM, RESULT1);
  MULTIPLY(RESULT1, HADM.MAT, RESULT2.MAT);
  RESULT2.SIZE:= DIM DIV 2;
  N41.SEND(RESULT2.MAT); N42.SEND(RESULT2.MAT);
  N43.SEND(RESULT2.MAT); N44.SEND(RESULT2.MAT)
END;

PROCESS HADTRANS4;
VAR
  IM, RESULT1: M2; RESULT2, HADM: RECType;
  DIM: INTEGER;
PROCEDURE MULTIPLY(F, R: M2) VAR G: M2);
VAR
  I, J, K: INTEGER;
BEGIN
  FOR I:= 1 TO DIM DIV 2 DO
    FOR J:= 1 TO DIM DIV 2 DO
      BEGIN
        QCI.JJ:= 0;
        FOR K:= 1 TO DIM DIV 2 DO
          BEGIN
            QCI.JJ:= QCI.JJ+PCI.K*K*K
          END
        END
  END;
BEGIN
  MJ.RECEIVE(HADM);
  RJ.RECEIVE(IM);
  DIM:= HADM.SIZE;
  MULTIPLY(HADM.MAT, IM, RESULT1);
  MULTIPLY(RESULT1, HADM.MAT, RESULT2.MAT);
  RESULT2.SIZE:= DIM DIV 2;
  N41.SEND(RESULT2.MAT); N42.SEND(RESULT2.MAT);
  N43.SEND(RESULT2.MAT); N44.SEND(RESULT2.MAT)
END;
PROCESS MERGER1;
VAR.
   I, J, A, B, C, D, DIM: INTERGER;
   H, H1: RECTYPE;
BEGIN
   N11 RECEIEVE(H1) IN21 RECEIVE(H2);
   N31 RECEIVE(H3) IN41 RECEIVE(H4);
   A:= 1; B:= 1;
   C:= 1; D:= 1;
   FOR I:= 1 TO DIM DO
     FOR J:= 1 TO DIM DO
     END;
   END;
END;

PROCESS MERGER2;
VAR.
   I, J, A, B, C, D, DIM: INTERGER;
   H1: RECTYPE;
BEGIN
   N12 RECEIVE(H1) IN22 RECEIVE(H2);
   N32 RECEIVE(H3) IN42 RECEIVE(H4);
   A:= 1; B:= 1;
   C:= 1; D:= 1;
   DIM:= H1.SIZE;
   FOR I:= 1 TO DIM DO
     FOR J:= 1 TO DIM DO
     END;
   END;
END;

PROCESS MERGER3;
VAR.
   I, J, A, B, C, D, DIM: INTERGER;
   H1: RECTYPE;
BEGIN
   N13 RECEIVE(H1) IN23 RECEIVE(H2);
   N33 RECEIVE(H3) IN43 RECEIVE(H4);
   A:= 1; B:= 1;
   C:= 1; D:= 1;
   DIM:= H1.SIZE;
   FOR I:= 1 TO DIM DO
     FOR J:= 1 TO DIM DO
     END;
   END;
END;
PROCESS MERGER4;
VAR
  I, J, A, B, C, D: INTEGER;
  H1: RECTYPE; H2, H3, H4: M2;
BEGIN
  N14.RECEIVE(H1); N24.RECEIVE(H2);
  N34.RECEIVE(H3); N44.RECEIVE(H4);
  A:= 1; E:= -1;
  C1:= -1; D1:= 1;
  DIM:= H1.SIZE;
  FOR I:= 1 TO DIM DO
    FOR J:= 1 TO DIM DO
      MM4.SEND(H)
END;

PROCESS RESULT;
VAR
  H1: RECTYPE; H2, H3, H4: M2;
  I, J, A, B: INTEGER;
BEGIN
  MM1.RECEIVE(H1);
  MM2.RECEIVE(H2);
  MM3.RECEIVE(H3);
  MM4.RECEIVE(H4);
  DIM:= H1.SIZE;
  WRITELN(' CONSIDERING THE IMAGE AS A WHOLE,');
  WRITELN(' THE TRANSFORM OF THE IMAGE IS ');
  WRITELN'; WRITELN;
  A:= DIM DIV 2;
  FOR I:= 1 TO DIM DO
  BEGIN
    FOR J:= 1 TO DIM DO
      BEGIN
        CASE I<DIM DIV 2 OF
          TRUE: IF J<DIM DIV 2 THEN HI[I,J]:= H1[MAT1[I,J]] ELSE HI[I,J]:= H2[I,J-A];
          FALSE: IF J<DIM DIV 2 THEN HI[I,J]:= H3[I,J-A] ELSE HI[I,J]:= H4[I-A,J-J-A];
        END;
        WRITE(TRUNC(H1[I,J]));
      END;
    WRITELN
  END;
END;
THE IMAGE IS OF SIZE: 16x16

THIS IS THE HADAMARD MATRIX USED:

```
1 1 1 1 1 1 1
-1 -1 -1 -1 -1 -1 -1
1 -1 -1 1 1 -1 -1
1 -1 -1 -1 -1 -1 1
1 -1 -1 -1 -1 -1 -1
-1 -1 1 1 1 -1 -1
```

THE IMAGE IS THE FOLLOWING:

```
1 1 1 1 1 1 5 5 5 5 5 1 1 1 1 1
1 1 1 1 1 1 5 5 5 5 5 1 1 1 1 1
1 1 1 1 1 1 5 5 5 5 5 1 1 1 1 1
1 1 1 1 1 1 5 5 5 5 5 1 1 1 1 1
1 1 1 1 1 1 5 5 5 5 5 1 1 1 1 1
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
1 1 1 1 1 1 5 5 5 5 5 1 1 1 1 1
1 1 1 1 1 1 5 5 5 5 5 1 1 1 1 1
1 1 1 1 1 1 5 5 5 5 5 1 1 1 1 1
1 1 1 1 1 1 5 5 5 5 5 1 1 1 1 1
1 1 1 1 1 1 5 5 5 5 5 1 1 1 1 1
```

CONSIDERING THE IMAGE AS A WHOLE,
THE TRANSFORM OF THE IMAGE IS:

```
704 0 0 0 0 0 0 192 0 0 0 -192 0 -192 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
192 0 0 0 0 0 0 -64 0 0 0 64 0 64 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-192 0 0 0 0 0 0 64 0 0 0 -64 0 -64 0 0 0
-192 0 0 0 0 0 0 64 0 0 0 -64 0 -64 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```
APPENDIX D

This appendix has a computer program written in PASCAL PLUS to simulate the data-flow system of Figure 5.4.17. Its purpose is to do concurrent Fourier transformation on the four quadrants $P_1$, $P_2$, $P_3$ and $P_4$ of an image. The image data is read from an external file named IMGBFER. The size of the image is specified by the constant IMSIZE. Each processor is equivalent to a processor in Figure 5.4.17. This equivalence is shown below.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Equivalent Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMAGE SUBDIVIDER</td>
<td>FDIVIDER</td>
</tr>
<tr>
<td>FOURIER GENERATOR</td>
<td>FOURIER</td>
</tr>
<tr>
<td>FOURIER PROCESSOR1</td>
<td>FTRANS1</td>
</tr>
<tr>
<td>FOURIER PROCESSOR2</td>
<td>FTRANS2</td>
</tr>
<tr>
<td>FOURIER PROCESSOR3</td>
<td>FTRANS3</td>
</tr>
<tr>
<td>FOURIER PROCESSOR4</td>
<td>FTRANS4</td>
</tr>
<tr>
<td>MERGER1</td>
<td>FFMERGER1</td>
</tr>
<tr>
<td>MERGER2</td>
<td>FFMERGER2</td>
</tr>
<tr>
<td>MERGER3</td>
<td>FFMERGER3</td>
</tr>
<tr>
<td>MERGER4</td>
<td>FFMERGER4</td>
</tr>
<tr>
<td>RESULT</td>
<td>FRES</td>
</tr>
</tbody>
</table>

An example is also given. This transformation is named the merging case since independent quadrant transforms are merged together to obtain the Fourier transform of the overall image. Different records have been used to concatenate data structures of a channel monitor (in PASCAL PLUS).
PROGRAM PROJS(IMGBFER,OUTPUT);
CONST IMFSIZE = 8;
TYPE
  MTR = ARRAY[1..16,1..16] OF REAL;
  MTRREC = RECORD
    RE: MTR;
    IM: MTR
  END;
  MTRREC1 = RECORD
    FIRST: MTRREC;
    SECOND: MTRREC;
    THIRD: MTRREC
  END;
  RECTYPE = RECORD
    MAT: MTRREC;
    CLR: INTEGER;
    URG: TEXT;
  END;
  MONITOR Chan1 = CHANNEL IN LIBRARY(WHERE TYPE BUFFTYPE = MTRREC);
  INSTANCE MM2,MM3,MM4:Chan1;
  INSTANCE S21,S22,S23,S24,S31,S32:Chan1;
  INSTANCE S33,S34,S41,S42,S43,S44:Chan1;
  MONITOR Chan2 = CHANNEL IN LIBRARY(WHERE TYPE BUFFTYPE = MTRREC);
  INSTANCE FA1,FA2,FA3,FA4:Chan2;
  MONITOR Chan3 = CHANNEL IN LIBRARY(WHERE TYPE BUFFTYPE = RECTYPE);
  INSTANCE MM1,R1,R2,R3,R4,S11,S12,S13,S14:Chan3;
  PROCESS FTRANS3 IN LIBRARY;
  INSTANCE XT3:FTRANS3;
  PROCESS FTRANS4 IN LIBRARY;
  INSTANCE XT4:FTRANS4;
  PROCESS FTRANS1 IN LIBRARY;
  INSTANCE XT1:FTRANS1;
  PROCESS FTRANS2 IN LIBRARY;
  INSTANCE XT2:FTRANS2;
  PROCESS FMERGER1 IN LIBRARY;
  INSTANCE XF1:FFMERGER1;
  PROCESS FMERGER2 IN LIBRARY;
  INSTANCE XF2:FFMERGER2;
  PROCESS FMERGER3 IN LIBRARY;
  INSTANCE XF3:FFMERGER3;
  PROCESS FMERGER4 IN LIBRARY;
  INSTANCE XF4:FFMERGER4;
  PROCESS FRES IN LIBRARY;
  INSTANCEXF: FRES;
  PROCESS FDIIVER IN LIBRARY;
  INSTANCE XD1:FDIVER(IMGBFER,IMFSIZE);
  PROCESS FOURIER IN LIBRARY;
  INSTANCE XD4:FOURIER(IMFSIZE);
BEGIN
  RESET(IMGBFER);
  WRITE('THE IMAGE IS OF SIZE:
  WRITELN(IMFSIZE);
  ***
END;

MONITOR CHANNEL;
VAR
  IMAGE: TEXT;
  MONITOR Channel: CHANNEL IN LIBRARY(WHERE TYPE BUFFTYPE = MTRREC);
  INSTANCE MM1,MM2,MM3,MM4:Chan1;
  INSTANCE S21,S22,S23,S24,S31,S32:Chan1;
  INSTANCE S33,S34,S41,S42,S43,S44:Chan1;
  MONITOR Chan2 = CHANNEL IN LIBRARY(WHERE TYPE BUFFTYPE = MTRREC);
  INSTANCE FA1,FA2,FA3,FA4:Chan2;
  MONITOR Chan3 = CHANNEL IN LIBRARY(WHERE TYPE BUFFTYPE = RECTYPE);
  INSTANCE MM1,R1,R2,R3,R4,S11,S12,S13,S14:Chan3;
  PROCESS FTRANS3 IN LIBRARY;
  INSTANCE XT3:FTRANS3;
  PROCESS FTRANS4 IN LIBRARY;
  INSTANCE XT4:FTRANS4;
  PROCESS FTRANS1 IN LIBRARY;
  INSTANCE XT1:FTRANS1;
  PROCESS FTRANS2 IN LIBRARY;
  INSTANCE XT2:FTRANS2;
  PROCESS FMERGER1 IN LIBRARY;
  INSTANCE XF1:FFMERGER1;
  PROCESS FMERGER2 IN LIBRARY;
  INSTANCE XF2:FFMERGER2;
  PROCESS FMERGER3 IN LIBRARY;
  INSTANCE XF3:FFMERGER3;
  PROCESS FMERGER4 IN LIBRARY;
  INSTANCE XF4:FFMERGER4;
  PROCESS FRES IN LIBRARY;
  INSTANCEXF: FRES;
  PROCESS FDIIVER IN LIBRARY;
  INSTANCE XD1:FDIVER(IMGBFER,IMFSIZE);
  PROCESS FOURIER IN LIBRARY;
  INSTANCE XD4:FOURIER(IMFSIZE);
BEGIN
  RESET(IMGBFER);
  WRITE('THE IMAGE IS OF SIZE:
  WRITELN(IMFSIZE);
  ***
END:
PROCESS FDIVIDER(VAR INF:TEXT; DIM:INTEGER);
VAR
  I,J,A: INTEGER;
  P1,P2,P3,P4: RECTYPE;
  TEMP: REAL;
BEGIN
  RESET(INF);
  WRITELN('THE IMAGE IS THE FOLLOWING');
  WRITELN; WRITELN;
  A:= DIM DIV 2;
  FOR I:= 1 TO DIM DO
  BEGIN
    FOR J:= 1 TO DIM DO
    BEGIN
      READ(INF,TEMP);
      IF (I=A) AND (J<=A) THEN
      BEGIN
        P1.MAT.RECI,J:= TEMP; P1.MAT.IMC,I:= 0.0
      END;
      IF (I=A) AND (J>A) THEN
      BEGIN
        P2.MAT.RECI,J:= TEMP; P2.MAT.IMC,I:= 0.0
      END;
      IF (I<=A) AND (J=A) THEN
      BEGIN
        P3.MAT.RECI,I:= TEMP; P3.MAT.IMC,J:= 0.0
      END;
      IF (I>A) AND (J=A) THEN
      BEGIN
        P4.MAT.RECI,A,J:= TEMP; P4.MAT.IMC,A,J:= 0.0
      END;
    END;
    WRITELN; WRITELN;
    P1.SIZE:= DIM; P2.SIZE:= DIM;
    P3.SIZE:= DIM; P4.SIZE:= DIM;
    R1.SEND(P1);
    R2.SEND(P2);
    R3.SEND(P3);
    R4.SEND(P4);
END;

PROCEDURE MUL(M1,M2,M3,M4:MTRREC; VAR M4:MTRREC; DIM:INTEGER);
VAR
  RES1:MTRREC;
PROCEDURE MULT(P,R:MTRREC; VAR Q:MTRREC);
VAR
  I,J,K: INTEGER;
BEGIN
  FOR I:= 1 TO DIM DIV 2 DO
  BEGIN
    FOR J:= 1 TO DIM DIV 2 DO
    BEGIN
      Q.RECI,J:= I; Q.IMC,I:= 0;
      FOR K:= 1 TO DIM DIV 2 DO
      BEGIN
        Q.RECI,J:= Q.RECI,J+P.RECI,K*J*RECK,J-P.IMC,K*J*IMCK,J;
        Q.IMC,J:= Q.IMC,J+P.IMC,K*J*IMCK,J+P.IMC,K*J*RECK,J;
      END
    END,
  END;
  MULT(M1,M2,RES1);
  MULT(RES1,M3,M4);
END;
PROCESS FOURIER(DIM:INTEGER);
CONST PI=3.1415;
VAR F1,F2,F3,F4:MTREC1;I,J,A,B:INTEGER;
C,D,E: REAL;
BEGIN WRITELN(' THE FOURIER MATRIX (IN MAGNITUDE) IS : ');
WRITELN; WRITELN;
F:= r11n 1iIw ;
FOR I:= 1 TO DIM DO BEGIN
A:= (I-1)*(J-1);
C:= COS(A*2*PI/(DIM));
D:= -SIN(A*2*PI/(DIM));
E:= SORT(CMC + D#);
IF (I=B) AND (J=B) THEN BEGIN
F1.FIRST.REF[I,J]= C; F1.FIRST.IMG[I,J]:= i;
F2.FIRST.REF[I,J]= C; F2.FIRST.IMG[I,J]:= i;
F3.FIRST.REF[I,J]= C; F3.FIRST.IMG[I,J]:= i;
END;
IF (I=B) AND (J=B) THEN BEGIN
F1.SECOND.REF[I,J]= C; F1.SECOND.IMG[I,J]:= i;
F2.SECOND.REF[I,J]= C; F2.SECOND.IMG[I,J]:= i;
F3.SECOND.REF[I,J]= C; F3.SECOND.IMG[I,J]:= i;
END;
IF (I=B) AND (J=B) THEN BEGIN
F1.THIRD.REF[I,J]= C; F1.THIRD.IMG[I,J]:= i;
F2.THIRD.REF[I,J]= C; F2.THIRD.IMG[I,J]:= i;
F3.THIRD.REF[I,J]= C; F3.THIRD.IMG[I,J]:= i;
END;
WRITE(TRUNC(E):15,'');
END;
WRITE;
END;
FA1.SEND(F1); FA2.SEND(F2);
FA3.SEND(F3); FA4.SEND(F4);
END;

PROCESS FTRANS1;
VAR H: MTREC1; HA,HB,HC,HD,P: RECTYPE;
DIM: INTEGER;
PROCEDURE MUL IN LIBRARY;
BEGIN R1.RECEIVE(P); FA1.RECEIVE(H);
DIM:= P.SIZE;
MUL(H,FIRST+P.MAT,HA.MAT,DIM);
MUL(H,SECOND+P.MAT,HB.MAT,DIM);
MUL(H,THIRD+P.MAT,HC.MAT,DIM);
MUL(H,FOURTH+P.MAT,HD.MAT,DIM);
HA.SIZE:= DIM DIV 2;
HB.SIZE:= DIM DIV 2;
HC.SIZE:= DIM DIV 2;
HD.SIZE:= DIM DIV 2;
S11.SEND(HA);S12.SEND(HB);
S13.SEND(HC);S14.SEND(HD);
END;
PROCEDURE MUL IN LIBRARY:
BEGIN
    R0.RECEIVE(P); FA1.RECEIVE(H);
    DIM:= P.SIZE;
    MUL(H.FIRST,H.SECOND,P.MAT,HA+DIM);
    MUL(H.FIRST,H.THIRD,P.MAT,HB+DIM);
    MUL(H.SECOND,H.SECOND,P.MAT,HC+DIM);
    MUL(H.SECOND,H.THIRD,P.MAT,HD+DIM);
    S21.SEND(HA); S22.SEND(HB);
    S23.SEND(HC); S24.SEND(HD);
END;

PROCEDURE MUL IN LIBRARY:
BEGIN
    R0.RECEIVE(P); FA2.RECEIVE(H);
    DIM:= P.SIZE;
    MUL(H.FIRST,H.SECOND,P.MAT,HA+DIM);
    MUL(H.FIRST,H.THIRD,P.MAT,HB+DIM);
    MUL(H.SECOND,H.SECOND,P.MAT,HC+DIM);
    MUL(H.SECOND,H.THIRD,P.MAT,HD+DIM);
    S31.SEND(HA); S32.SEND(HB);
    S33.SEND(HC); S34.SEND(HD);
END;

PROCEDURE MUL IN LIBRARY:
BEGIN
    R0.RECEIVE(P); FA3.RECEIVE(H);
    DIM:= P.SIZE;
    MUL(H.FIRST,H.SECOND,P.MAT,HA+DIM);
    MUL(H.FIRST,H.THIRD,P.MAT,HB+DIM);
    MUL(H.SECOND,H. SECOND,P.MAT,HC+DIM);
    MUL(H.SECOND,H.THIRD,P.MAT,HD+DIM);
    S41.SEND(HA); S42.SEND(HB);
    S43.SEND(HC); S44.SEND(HD);
END;
PROCESS FFMERGER1;
VAR
  I,J,DIM: INTEGER;
  B,C,D:E: MTREC;
  A,E: RECTYPE;
BEGIN
  S11.RECEIVE(A); S21.RECEIVE(B);
  S31.RECEIVE(C); S41.RECEIVE(D);
  DIM:= A.SIZE;
  FOR I:= 1 TO DIM DO
    FOR J:= 1 TO DIM DO
      BEGIN
      END;
  E.SIZE:= 2*DIM;
  HM1.SEND(E);
END;

PROCESS FFMERGER2;
VAR
  I,J,DIM: INTEGER;
  B,C,D:E: MTREC;
  A: RECTYPE;
BEGIN
  S12.RECEIVE(A); S22.RECEIVE(B);
  S32.RECEIVE(C); S42.RECEIVE(D);
  DIM:= A.SIZE;
  FOR I:= 1 TO DIM DO
    FOR J:= 1 TO DIM DO
      BEGIN
      END;
  HM2.SEND(E);
END;

PROCESS FFMERGER3;
VAR
  I,J,DIM: INTEGER;
  B,C,D:E: MTREC;
  A: RECTYPE;
BEGIN
  S13.RECEIVE(A); S23.RECEIVE(B);
  S33.RECEIVE(C); S43.RECEIVE(D);
  DIM:= A.SIZE;
  FOR I:= 1 TO DIM DO
    FOR J:= 1 TO DIM DO
      BEGIN
      END;
  HM3.SEND(E);
END;
PROCESS FFMERGER4;
VAR
  I,J,DIM: INTEGER;
  A: MTRREC;
BEGIN
  S14.RECEIVE(A); S24.RECEIVE(B);
  S34.RECEIVE(C); S44.RECEIVE(D);
  DIM:= A.SIZE;
  FOR I:= 1 TO DIM DO
    FOR J:= 1 TO DIM DO
      BEGIN
        E.RECEIVE(JI):= A.MAT.JI + B.RECEIVE(JI) + C.RECEIVE(JI) + D.RECEIVE(JI);
      END;
      MM4.SEND(E);
END;

PROCESS FRES;
VAR
  H1,H2,H3,H4: MTRREC;
  I,J,DIM: INTEGER; A: REAL;
  TEMP: REAL; H1: RECTYPE;
BEGIN
  MM1.RECEIVE(H1);
  MM2.RECEIVE(H2);
  MM3.RECEIVE(H3);
  MM4.RECEIVE(H4);
  DIM:= H1.SIZE;
  WRITELN; WRITELN;
  WRITELN(' THE IMAGE TRANSFORM (IN MAGNITUDE) IS: ');
  WRITELN; WRITELN;
  B:= DIM DIV 2;
  FOR I:= 1 TO DIM DO
    BEGIN
      FOR J:= 1 TO DIM DO
        BEGIN
          CASE I:=DIM DIV 2 OF
            TRUE: IF J= DIM DIV 2 THEN
            FALSE: IF J= DIM DIV 2 THEN
          END;
        END;
      END;
    END;
    WRITE(TRUNC(A):S,' '); END;
END;
THE IMAGE IS OF SIZE: 16

THE IMAGE IS THE FOLLOWING:

```
1 1 1 1 1 5 5 5 5 5 1 1 1 1 1 1
1 1 1 1 1 5 5 5 5 5 1 1 1 1 1 1
1 1 1 1 1 5 5 5 5 5 1 1 1 1 1 1
1 1 1 1 1 5 5 5 5 5 1 1 1 1 1 1
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
1 1 1 1 1 5 5 5 5 5 1 1 1 1 1 1
1 1 1 1 1 5 5 5 5 5 1 1 1 1 1 1
1 1 1 1 1 5 5 5 5 5 1 1 1 1 1 1
1 1 1 1 1 5 5 5 5 5 1 1 1 1 1 1
```

THE FOURIER MATRIX (IN MAGNITUDE) IS:

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

THE IMAGE TRANSFORM (IN MAGNITUDE) IS:

```
704 173 125 61 0 40 51 34 0 34 51 40 0 60 125 173
173 52 37 18 0 12 15 10 0 10 15 12 0 18 37 52
125 37 27 13 0 8 11 7 0 7 11 8 0 13 37 52
41 18 13 4 0 4 5 3 0 3 5 4 0 6 13 18
41 18 13 4 0 4 5 3 0 3 5 4 0 6 13 18
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
40 12 8 4 0 2 3 2 0 2 3 2 0 4 8 12
51 15 11 5 0 3 4 3 0 3 4 3 0 7 11 15
34 10 7 3 0 2 3 2 0 2 3 2 0 7 11 15
34 10 7 3 0 2 3 2 0 2 3 2 0 7 11 15
51 15 11 5 0 3 4 3 0 3 4 3 0 7 11 15
40 12 8 4 0 2 3 2 0 2 3 2 0 4 8 12
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
60 18 13 6 0 4 5 3 0 3 5 4 0 6 13 18
125 37 27 13 0 8 11 7 0 7 11 8 0 13 37 52
173 52 37 18 0 12 15 10 0 10 15 12 0 18 37 52
```
APPENDIX E

This appendix has a computer program written in PASCAL PLUS to simulate the data-flow system of Figure 5.4.21. Its purpose is to do concurrent Hadamard transformation on the four quadrants $P_1$, $P_2$, $P_3$ and $P_4$ of an image. The image data is read from an external file named IMGBFER. The size of the image is specified by the constant IMSIZE. Each process is equivalent to a processor in Figure 5.4.21. This equivalence is shown below.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Equivalent Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMAGE SUBDIVIDER</td>
<td>DIVIDER</td>
</tr>
<tr>
<td>HADAMARD GENERATOR</td>
<td>HADMARD</td>
</tr>
<tr>
<td>HADAMARD TRANSFORM1</td>
<td>HADTRANS1</td>
</tr>
<tr>
<td>HADAMARD TRANSFORM2</td>
<td>HADTRANS2</td>
</tr>
<tr>
<td>HADAMARD TRANSFORM3</td>
<td>HADTRANS3</td>
</tr>
<tr>
<td>HADAMARD TRANSFORM4</td>
<td>HADTRANS4</td>
</tr>
<tr>
<td>RESULT</td>
<td>RESULTw</td>
</tr>
</tbody>
</table>

An example is also given. The variable TRANSMIT is set true for Hadamard transformation and false for inverse Hadamard transformation. This transformation is named the non-merging case since each quadrant of the image is transform-coded independently without merging those quadrants to obtain the overall image transform. A similar program that does Fourier transformation (non-merging) follows the same principle.
PROGRAM PROJ(IMGBFER=OUTPUT);
CONST IMSIZE = 16;
TRANSMIT = TRUE;

TYPE
M2=ARRAY[1..IMSIZE,1..IMSIZE] OF REAL;
RECTYPE = RECORD
  MAT:M2;
  SIZE:INTEGER
END;
VAR IMGBFER:TEXT;

MONITOR CHAN1=CHANNEL IN LIBRARY(WHERE TYPE BUFFTYPE=M2);
INSTANCE R1,R2,R3,R4,M2,M3,M4:CHAN1;
MONITOR CHAN2=CHANNEL IN LIBRARY(WHERE TYPE BUFFTYPE=RECTYPE);
INSTANCE M1,M2,M3,M4,M1:CHAN2;
PROCESS HADTRANS3 IN LIBRARY;
INSTANCE XH3: HADTRANS3;
PROCESS HADTRANS4 IN LIBRARY;
INSTANCE XH4: HADTRANS4;
PROCESS HADTRANS1 IN LIBRARY;
INSTANCE XH1: HADTRANS1;
PROCESS HADTRANS2 IN LIBRARY;
INSTANCE XH2: HADTRANS2;
PROCESS RESULTw IN LIBRARY;
INSTANCE XRES: RESULTw;
PROCESS HADMARD IN LIBRARY;
INSTANCE XHADMARD: HADMARD(IMSIZE DIV 2);
PROCESS DIVIDER IN LIBRARY;
INSTANCE XDIV: DIVIDER(IMGBFER,IMSIZE);
BEGIN
  RESET(IMGBFER);
  WRITE('THE IMAGE IS OF SIZE : ');
  WRITELN(IMSIZE:3);
  WRITELN(IMSIZE:3);WRITELN;WRITELN;
  ***
END;

BEGIN
  FULL:= FALSE;
  ***
END;

MONITOR CHANNEL;
VAR
  BUFF: BUFFTYPE; FULL: BOOLEAN; INSTANCE QUEUE: CONDITION;
PROCEDURE #SEND(LINE: BUFFTYPE);
BEGIN
  IF FULL THEN QUEUE.WAIT;
  BUFF:= LINE; FULL:= TRUE; QUEUE.SIGNAL
END;
PROCEDURE #RECEIVE(VAR LINE:BUFFTYPE);
BEGIN
  IF NOT FULL THEN QUEUE.WAIT;
  LINE:= BUFF; FULL:= FALSE; QUEUE.SIGNAL
END;
BEGIN
  FULL:= FALSE;
  ***
END;
PROCESS DIVIDER(VAR INF;TEXT;DIM: INTEGER);
VAR
  I,J,A: INTEGER; TEMP: REAL;
  P1,P2,P3,P4: M2;
BEGIN
  RESET(INF);
  IF TRANSMIT THEN
    WRITELN(' THE IMAGE IS THE FOLLOWING :')
  ELSE
    WRITELN(' THE RECEIVED TRANSFORM IS :');
    WRITELN;
    A:= DIM DIV 2;
    FOR I:= 1 TO DIM DO
      BEGIN
        FOR J:= 1 TO DIM DO
          BEGIN
            READ(INF,TEMP);
            WRITE(TRUNC(TEMP):3,'');
            IF I<=DIM DIV 2 AND J<=DIM DIV 2 THEN
              PII[I+J]= TEMP;
            IF I<=DIM DIV 2 AND J>DIM DIV 2 THEN
              PI[I-J+A]= TEMP;
            IF I>DIM DIV 2 AND J<=DIM DIV 2 THEN
              PIJ[I-A,J]= TEMP;
            IF I>DIM DIV 2 AND J>DIM DIV 2 THEN
              PI[I-J-A]= TEMP;
          END;
        WRITELN;
      END;
    SEND(P1);
    SEND(P2);
    SEND(P3);
    SEND(P4);
END;
BEGIN
  PROCESS HADMARD(DIM:INTEGER);
  VAR
    HADM,RECTYPE:
    I,J, NOFTIMES: INTEGER;
  FUNCTION TWO(B:INTEGER):INTEGER:
    BEGIN
      TWO:=TRUNC(EXP(B*LN(2))+.5)
    END;
  PROCEDURE BUILDRECURSIVELY(NUM, START: INTEGER);
  VAR
    I,J,A,B: INTEGER;
  BEGIN
    A:=TWO(NUM-1);
    B:=TWO(NUM);
    FOR I:=1 TO A DO
      FOR J:=START TO B DO
        BEGIN
          HADM.MAT[I,J]= HADM.MAT[I,J-A]
          HADM.MAT[I+A,J-A]= HADM.MAT[I,J-A]
          HADM.MAT[I+J]= -HADM.MAT[I,A,J-A]
        END;
  END;
  BEGIN
    WRITELN(' THIS IS THE HADAMARD MATRIX USED ');
    WRITELN;
    NOFTIMES:= TRUNC(LN(DIM)/LN(2)+.5);
    HADM.MAT[I,J]:= 1;
    FOR I:= 1 TO NOFTIMES DO
      BUILDRECURSIVELY(I*TWO(I-1) + 1);
    FOR I:= 1 TO DIM DO
      BEGIN
        FOR J:= 1 TO DIM DO
          BEGIN
            WRITE(TRUNC(HADM.MAT[I,J]):2,'');
          END;
        WRITELN;
      END;
    HADM.SIZE:= 2*DIM;
    WRITELN;
    m1 := SEND(HADM);
    M22 := SEND(HADM);
    M33 := SEND(HADM);
    M44 := SEND(HADM);
  END;
PROCESS HADTRANS1;
VAR
  IM, RESULT1: M2; RESULT2:HADM: RECTYPE;
  DIM: INTEGER;
PROCEDURE MULTIPLY(F:RM2) VAR G: M2);
VAR
  I, J, K: INTEGER;
BEGIN
  FOR I := 1 TO DIM DIV 2 DO
    FOR J := 1 TO DIM DIV 2 DO
      BEGIN
        QC1[I,J1] := 0;
        FOR K := 1 TO DIM DIV 2 DO
      END;
  END;
BEGIN
  M1.RECEIVE(HADM);
  R1.RECEIVE(IM);
  DIM := HADM.SIZE;
  MULTIPLY(HADM.MAT,IM,RESULT1);
  MULTIPLY(RESULT1,HADM.MAT,RESULT2.MAT);
  RESULT2.SIZE := DIM;
  MM1.SEND(RESULT2)
END;

PROCESS HADTRANS2;
VAR
  IM, RESULT1,RESULT2, M2; HADM:RECTYPE;
  DIM: INTEGER;
PROCEDURE MULTIPLY(F:RM2) VAR G: M2);
VAR
  I, J, K: INTEGER;
BEGIN
  FOR I := 1 TO DIM DIV 2 DO
    FOR J := 1 TO DIM DIV 2 DO
      BEGIN
        QC1[I,J1] := 0;
        FOR K := 1 TO DIM DIV 2 DO
      END;
  END;
BEGIN
  M2.RECEIVE(HADM);
  R2.RECEIVE(IM);
  DIM := HADM.SIZE;
  MULTIPLY(HADM.MAT,IM,RESULT1);
  MULTIPLY(RESULT1,HADM.MAT,RESULT2);
  MM2.SEND(RESULT2)
END;
PROCESS HADTRANS3;
VAR
  IM,RESULT1,RESULT2, M2; HADM:RECTYPE;
  DIM: INTEGER;
PROCEDURE MULTIPLY(p:R:M2; VAR Q: M2);
VAR
  I, J, K: INTEGER;
BEGIN
  FOR I:= 1 TO DIM DIV 2 DO
    FOR J:= 1 TO DIM DIV 2 DO
      BEGIN
        Q[I,J]:= 0;
        FOR K:= 1 TO DIM DIV 2 DO
          Q[I,J]:= Q[I,J]+P[I,K]*R[K,J]
      END
END;
BEGIN
  M3.RECEIVE(HADM);
  R3.RECEIVE(IM);
  DIM:= HADM.SIZE;
  MULTIPLY(HADM,MAT,IM,RESULT1);
  MULTIPLY(RESULT1,HADM,MAT,RESULT2);
  MM3 SEND(RESULT2)
END;

PROCESS HADTRANS4;
VAR
  IM,RESULT1,RESULT2, M2; HADM:RECTYPE;
  DIM: INTEGER;
PROCEDURE MULTIPLY(p:R:M2; VAR Q: M2);
VAR
  I, J, K: INTEGER;
BEGIN
  FOR I:= 1 TO DIM DIV 2 DO
    FOR J:= 1 TO DIM DIV 2 DO
      BEGIN
        Q[I,J]:= 0;
        FOR K:= 1 TO DIM DIV 2 DO
          Q[I,J]:= Q[I,J]+P[I,K]*R[K,J]
      END
END;
BEGIN
  M4.RECEIVE(HADM);
  R4.RECEIVE(IM);
  DIM:= HADM.SIZE;
  MULTIPLY(HADM,MAT,IM,RESULT1);
  MULTIPLY(RESULT1,HADM,MAT,RESULT2);
  MM4 SEND(RESULT2)
END;
PROCESS RESULT;
VAR
  H:2,H3,H4:2; H1: RECTYPE;
  I,J:DIM: INTEGER; AB,CD: REAL;
BEGIN
  MM1.RECEIVE(H1);
  MM2.RECEIVE(H2);
  MM3.RECEIVE(H3);
  MM4.RECEIVE(H4);
  WRITELN(' CONSIDERING EACH QUADRANT INDEPENDENTLY,');
  IF TRANSMIT THEN
    WRITELN(' THE TRANSFORM OF THE IMAGE IS ');
    ELSE
    WRITELN(' THE RECEIVED IMAGE IS ');
  WRITELN;
  DIM:= H1.SIZE;
  A:= DIM DIV 2;
  FOR I:= 1 TO DIM DO
    BEGIN
      FOR J:= 1 TO DIM DO
      BEGIN
        CASE I:=A OF
        END;
        IF HCI,J >= 0 THEN AB:= 0.1
        ELSE AB:= -0.1;
        IF TRANSMIT THEN CD:= 1.0
        ELSE CD:= SQR(DIM/2);
        WRITE( TRUNC(HCI,J)/CD + AB):3,
      END;
    END;
  END;
THE IMAGE IS OF SIZE: 16

THIS IS THE HADAMARD MATRIX USED:

\[
\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & -1 & -1 & -1 & 1 & 1 & -1 & 1 \\
1 & -1 & 1 & 1 & -1 & -1 & -1 & 1 \\
1 & 1 & 1 & -1 & 1 & -1 & -1 & -1 \\
1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\
1 & -1 & -1 & -1 & -1 & 1 & 1 & 1 \\
1 & -1 & -1 & -1 & -1 & -1 & 1 & 1 \\
1 & -1 & 1 & 1 & -1 & 1 & -1 & 1
\end{bmatrix}
\]

THE IMAGE IS THE FOLLOWING:

\[
\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 5 & 5 \\
1 & 1 & 1 & 1 & 1 & 5 & 5 & 5 \\
1 & 1 & 1 & 1 & 1 & 5 & 5 & 5 \\
1 & 1 & 1 & 1 & 1 & 5 & 5 & 5 \\
1 & 1 & 1 & 1 & 1 & 1 & 5 & 5 \\
1 & 1 & 1 & 1 & 1 & 5 & 5 & 5 \\
1 & 1 & 1 & 1 & 1 & 5 & 5 & 5 \\
5 & 5 & 5 & 5 & 5 & 5 & 5 & 5
\end{bmatrix}
\]

CONSIDERING EACH QUADRANT INDEPENDENTLY,
THE TRANSFORM OF THE IMAGE IS:

\[
\begin{bmatrix}
176 & 0 & -48 & 0 & -48 & 0 & 48 & 0 \\
-48 & 0 & -16 & 0 & -16 & 0 & 16 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-48 & 0 & -16 & 0 & -16 & 0 & 16 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
48 & 0 & -16 & 0 & -16 & 0 & 16 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
176 & 0 & -48 & 0 & -48 & 0 & 48 & 0
\end{bmatrix}
\]
APPENDIX F

This appendix has a computer program written in PASCAL PLUS to simulate the data-flow system of Figure 5.8.2. The purpose of it is to concurrently decompose the image and transform-code its information quadrants (using the segmentation scheme) once detected. The input data is read from an external file named TREEFILE. DECOMPOSE process specifies the threshold values TETA1 and TETA2, the levels of decomposition MINLEVEL and MAXLEVEL, and the image size MAXSIZE. Each process is equivalent to a processor in Figure 5.4.21. This equivalence is shown below.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Equivalent Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGULAR DECOMPOSITION</td>
<td>DECOMPOSE</td>
</tr>
<tr>
<td>QUADRANT SUBDIVIDER</td>
<td>DIVIMAGE</td>
</tr>
<tr>
<td>HADAMARD GENERATOR</td>
<td>HADAMARD</td>
</tr>
<tr>
<td>HADAMARD TRANSFORM1</td>
<td>HADPR1</td>
</tr>
<tr>
<td>HADAMARD TRANSFORM2</td>
<td>HADPR2</td>
</tr>
<tr>
<td>HADAMARD TRANSFORM3</td>
<td>HADPR3</td>
</tr>
<tr>
<td>HADAMARD TRANSFORM4</td>
<td>HADPR4</td>
</tr>
<tr>
<td>RESULT</td>
<td>RESPR</td>
</tr>
</tbody>
</table>

In DECOMPOSE process (explained in Appendix G), procedure BUILDER is to build the quadrant (have it as a two-dimensional array from its one-dimensional array at the last level of the tree) of a corresponding informative node. This program is highly concurrent and could be easily transformed to do Fourier transformation.
PROGRAM FRUJO(TREEFILE, OUTPUT);
TYPE
  RECTYPE2 = RECORD
    LEV: INTEGER;
    NOD: INTEGER;
    INFO: BOOLEAN;
  END;
  M2 = ARRAY[1..25, 1..25] OF INTEGER;
  RECTYPE = RECORD
    MAT: M2;
    SIZE: INTEGER;
  END;
VAR
  TREEFILE: TEXT;

MONITOR CHAN1 = CHANNEL IN LIBRARY(WHERE TYPE BUFFTYPE = M2);
INSTANCE R1, R2, R3, R4, M2, M3, M4: CHAN1;
MONITOR CHAN2 = CHANNEL IN LIBRARY(WHERE TYPE BUFFTYPE = RECTYPE);
INSTANCE M1, M2, M3, M4, M5: CHAN2;
MONITOR CHAN3 = CHANNEL IN LIBRARY(WHERE TYPE BUFFTYPE = RECTYPE2);
INSTANCE KK: CHAN3;
MONITOR CHAN4 = CHANNEL IN LIBRARY(WHERE TYPE BUFFTYPE = INTEGER);
INSTANCE RC: CHAN4;
PROCESS DECOMPOSE IN LIBRARY;
INSTANCE XDEC: DECOMPOSE(TREEFILE);
PROCESS HADPR3 IN LIBRARY;
INSTANCE XI3: HADPR3;
PROCESS HADPR4 IN LIBRARY;
INSTANCE XI4: HADPR4;
PROCESS HADPR1 IN LIBRARY;
INSTANCE XI1: HADPR1;
PROCESS HADPR2 IN LIBRARY;
INSTANCE XI2: HADPR2;
PROCESS RESPR IN LIBRARY;
INSTANCE XRES: RESPR;
PROCESS HADAMARD IN LIBRARY;
INSTANCE XHADAMARD: HADAMARD;
PROCESS DIVIMAGE IN LIBRARY;
INSTANCE XDIV: DIVIMAGE;
BEGIN
  RESCT(TREEFILE);
  ***
END.

PROCESS DECOMPOSE(VAR INF: TEXT);
CONST TETA1 = 0.25; TETA2 = 0.75;
  MINLEVEL = 0; MAXLEVEL = 3;
  MAXSIZE = 8;
TYPE
  ATYPE = ARRAY[1..500] OF INTEGER;
  PTRTONODE = @NODE;
  NODE = RECORD
    INDEX: INTEGER;
    LINK: PTRTONODE;
  END;
VAR
  TREE: ATYPE;
  P, FIRST, TEMPFIRST, SAVEFIRST: PTRTONODE;
  LEVEL, FATHER, I, TEMP: INTEGER;
  FIRSTELEMENT, INFORMATIVE: BOOLEAN;
FUNCTION FOUR(B: INTEGER): INTEGER
BEGIN
  FOUR: = TRUNC(EXP(B * LN(4)) + 0.5)
END;
PROCEDURE BUILDER IN LIBRARY;
PROCEDURE FLUSHSTACK;
VAR
  Q: PTRTONODE; J: INTEGER;
BEGIN
  INFORMATIVE: = TRUE;
  P: = FIRST;
  WHILE (P<>NIL) DO...
BEGIN
  J := P0.INDEX;
  BUILDER(LEVEL, J, INFORMATIVE);
  G := P;
  F := P0.LINK;
  DISPOSE(Q)
END
PROCEDURE PUSHONSTACK(LEVEL:INTEGER);
VAR
  START, UPPERMOST: INTEGER;
BEGIN
  START := (FOUR(LEVEL) + 2) DIV 3;
  UPPERMOST := START + FOUR(LEVEL);
  FIRST := NIL;
  WHILE (START <> UPPERMOST) DO
    BEGIN
      NEW(P);
      P0.INDEX := START;
      P0.LINK := FIRST;
      FIRST := P;
      START := START + 1
    END;
END;
PROCEDURE CHECKSONS(FATHER, LEVEL:INTEGER);
VAR
  J: INTEGER; AVG: REAL;
FUNCTION FIRSTSON(FATHER:INTEGER): INTEGER;
VAR
  RANK: INTEGER;
BEGIN
  RANK := FATHER - (FOUR(LEVEL) + 2) DIV 3;
  FIRSTSON := (FOUR(LEVEL + 1) + 2) DIV 3 + 4*RANK
END;
BEGIN
  TEMPFIRST := FIRST;
  FOR J := FIRSTSON(FATHER) + 3 DOWNTO FIRSTSON(FATHER) DO
    BEGIN
      INFORMATIVE := TRUE;
      AVG := TREE(J)/TREE(FATHER);
      IF (AVG >= 0) AND (AVG < TETA1) THEN
        BEGIN
          INFORMATIVE := FALSE;
          BUILDER(LEVEL + 1, J, INFORMATIVE)
        END;
      ELSE
        IF (AVG >= TETA2) AND (AVG < TETA3) THEN
          BUILDER(LEVEL + 1, J, INFORMATIVE)
        ELSE
          BEGIN
            NEW(P);
            P0.INDEX := J;
            P0.LINK := TEMPFIRST;
            IF FIRSTELEMENT = TRUE THEN
              BEGIN
                P0.LINK := NIL;
                FIRSTELEMENT := FALSE
              END;
            TEMPFIRST := P
          END;
    END;
  FIRST := TEMPFIRST
END;
BEGIN
  CLEAR(INF);
  WRITE('THE STARTING LEVEL IS: ', MINLEVEL:4);
  WRITE; WRITELN;
  WRITE('THE MAXIMUM LEVEL IS: ', MAXLEVEL:4);
  WRITE; WRITELN;
  FOR I := 1 TO (4*MAXSIZE-MAXSIZE - 1) DIV 3 DO
    BEGIN
      READ(INF, TEMP);
      TREE[I] := TEMP
    END;
END;
FSIICUSTACK(MINLEVEL)
LEVEL := MINLEVEL;
REPEAT
  FIRSTELEMENT := TRUE;
  SAVEFIRST := FIRST;
  REPEAT
    FATHER := SAVEFIRST@INDEX;
    F := SAVEFIRST;
    SAVEFIRST := SAVEFIRST@LINK;
    DISPOSE(F);
  CHECKSONS(FATHER@LEVEL);
UNTIL SAVEFIRST = NIL;
LEVEL := LEVEL + 1;
UNTIL (FIRSTELEMENT = TRUE) OR (LEVEL = MAXLEVEL);
FLUSHSTACK;

PROCEDURE BUILDER(LEVEL, NODE: INTEGER; INFORMATIVE: BOOLEAN);
VAR
  I, J, BETA, RANK, STR, QUADSIZE: INTEGER;
  IND, IND+1, VAL, RECTYPE;
  HEADING: RECTYPE;
FUNCTION TWO(A: INTEGER): INTEGER;
BEGIN
  TWO := TRUNC((EXP(A*LN(2)) + 0.5));
END;

PROCEDURE BUILD(NUM, START: INTEGER);
VAR
  I, J, A: INTEGER;
BEGIN
  A := TWO(NUM-1);
  FOR I := 1 TO TWO(NUM-1) DO
    FOR J := START TO TWO(NUM) DO
      BEGIN
        IND[I, J] := IND[I, J-A] + SQRT(TWO(NUM-1));
        VAL[I, MAT[I, J]] := TREECIND[I, J];
        IND[I+1, A, J-A] := IND[I, J-A] + TWO(2*NUM-1);
        VAL[I+1, A, J] := IND[I+1, A, J-A] + SQRT(TWO(NUM-1));
        VAL[I+1, J] := TREECIND[I+1, A, J];
      END;
  END;
  BETA := TRUNC(LN(MAXSIZE)/LN(2) + 0.5);
  RANK := NODE - ((FOUR(LEVEL) + 2) DIV 3);
  QUADSIZE := TRUNC(SORT(FOUR(BETA - LEVEL))) DIV 2;
  VAL.SIZE := QUADSIZE*2;
  HEADING.INFO := TRUE;
  HEADING.LEVL := LEVEL;
  HEADING.NODE := NODE;
  IF INFORMATIVE THEN
    BEGIN
      WRITE(‘NODE’,NODE:3,’ AT LEVEL’,LEVEL:3);
      WRITE(‘ IS INFORMATIVE (DETECTED)’);
      WRITE(‘ THIS IS ITS CORRESPONDING QUADRANT’);
      IND[I, J] := STR;
      VAL.MAT[I, J] := TREE(STR);
      FOR I := 1 TO (BETA - LEVEL) DO
        BUILD(1, TWO(I-I) + I);
      FOR I := 1 TO TWO(BETA - LEVEL) DO
        BEGIN
          FOR J := 1 TO TWO(BETA - LEVEL) DO
            WRITE(VAL.MAT[I, J]);
        END;
    END;
  ELSE
    BEGIN
      WRITE(‘NODE’,NODE:3,’ AT LEVEL’,LEVEL:3);
      WRITE(‘ IS NOT INFORMATIVE (DETECTED)’);
      HEADING.INFO := FALSE;
    END;
  BEGIN
    KK.SEND(HEADING);
    RR.SEND(QUADSIZE);
    GG.SEND(VAL);
  END;
PROCESS DIVIMAGE;
VAR
  I,J,A,DIM: INTEGER;
  F1,F2,P3,P4: M2;
  TEMP: RECTYPE;
BEGIN
  GG.RECEIVE(TEMP);
  DIM:= TEMP.SIZE;
  WHILE (DIM >= 2) DO
BEGIN
  A:= DIM DIV 2;
  FOR I:= 1 TO DIM DO
BEGIN
    FOR J:= 1 TO DIM DO
BEGIN
      IF (I=DIM DIV 2) AND (J=DIM DIV 2) THEN
        F1[I,J]:= TEMP.MAT[I,J];
      IF (I=DIM DIV 2) AND (J<DIM DIV 2) THEN
        F2[I,J-A]:= TEMP.MAT[I,J];
      IF (I<DIM DIV 2) AND (J=DIM DIV 2) THEN
        P3[I-A,J]:= TEMP.MAT[I,J];
      IF (I<DIM DIV 2) AND (J<DIM DIV 2) THEN
        P4[I-A,J-A]:= TEMP.MAT[I,J];
    END;
END;
  R1.SEND(F1);
  R2.SEND(F2);
  R3.SEND(P3);
  R4.SEND(P4);
  GG.RECEIVE(TEMP);
END;
PROCESS HADM1;
VAR
  I, RESULT1: M2; RESULT2, HADM: RECTYPE;
  DIM: INTEGER;
PROCEDURE MULTIPLY(F: R; M: M2; VAR Q: M2);
VAR
  I, J, K: INTEGER;
BEGIN
  FOR I := 1 TO DIM DIV 2 DO
    FOR J := 1 TO DIM DIV 2 DO
      BEGIN
        FOR K := 1 TO DIM DIV 2 DO
          Q[I, J] := 0;
        FOR K := 1 TO DIM DIV 2 DO
      END
END;
BEGIN
  HADM, RESULT1, RESULT2: M2;
  HADM: RECTYPE;
  DIM: INTEGER;
PROCEDURE MULTIPLY(F: R; M: M2; VAR Q: M2);
VAR
  I, J, K: INTEGER;
BEGIN
  FOR I := 1 TO DIM DIV 2 DO
    FOR J := 1 TO DIM DIV 2 DO
      BEGIN
        FOR K := 1 TO DIM DIV 2 DO
          Q[I, J] := 0;
        FOR K := 1 TO DIM DIV 2 DO
      END
END;
BEGIN
  M22, RECEIVE(HADM);
  R2, RECEIVE(IM);
  DIM: HADM, SIZE;
  WHILE (DIM >= 2) DO
    BEGIN
      MULTIPLY(HADM, MAT, IM, RESULT1);
      MULTIPLY(RESULT1, HADM, MAT, RESULT2);
      RESULT2, SIZE := DIM;
      M22: SEND(RESULT2);
      M1: RECEIVE(HADM);
      R1: RECEIVE(IM);
      DIM := HADM, SIZE;
    END;
END;
END;

PROCESS HADM2;
VAR
  I, RESULT1, RESULT2: M2; HADM: RECTYPE;
  DIM: INTEGER;
PROCEDURE MULTIPLY(F: R; M: M2; VAR Q: M2);
VAR
  I, J, K: INTEGER;
BEGIN
  FOR I := 1 TO DIM DIV 2 DO
    FOR J := 1 TO DIM DIV 2 DO
      BEGIN
        FOR K := 1 TO DIM DIV 2 DO
          Q[I, J] := 0;
        FOR K := 1 TO DIM DIV 2 DO
      END
END;
BEGIN
  M22: RECEIVE(HADM);
  R2: RECEIVE(IM);
  DIM: HADM, SIZE;
  WHILE (DIM >= 2) DO
    BEGIN
      MULTIPLY(HADM, MAT, IM, RESULT1);
      MULTIPLY(RESULT1, HADM, MAT, RESULT2);
      M22: SEND(RESULT2);
      M22: RECEIVE(HADM);
      R2: RECEIVE(IM);
      DIM := HADM, SIZE;
    END;
END;
END;
PROCESS HADR3;
VAR
IM:RESULT1,RESULT2: M2; HADM:RECTYPE;
DIM: INTEGER;
PROCEDURE MULTIPLY(F, R: M2; VAR Q: M2);
VAR
I, J, K: INTEGER;
BEGIN
FOR I:= 1 TO DIM DIV 2 DO
FOR J:= 1 TO DIM DIV 2 DO
BEGIN
Q[I,J]:= 0;
FOR K:= 1 TO DIM DIV 2 DO
Q[I,J]:= Q[I,J]+F[I,K]*R[K,J]
END
END;
BEGIN
M3.RECEIVE(HADM);
R3.RECEIVE(IM);
DIM:= HADM.SIZE;
WHILE (DIM >= 2) DO
BEGIN
MULTIPLY(HADM,MATvIMvRESULT1);
MULTIPLY(RESULT1,HADM,MATvRESULT2);
MM3 SEND(RESULT2);
M3.RECEIVE(HADM);
R3.RECEIVE(IM);
DIM:= HADM.SIZE;
END;
END;

PROCESS HADR4;
VAR
IM:RESULT1,RESULT2: M2; HADM:RECTYPE;
DIM: INTEGER;
PROCEDURE MULTIPLY(F, R: M2; VAR Q: M2);
VAR
I, J, K: INTEGER;
BEGIN
FOR I:= 1 TO DIM DIV 2 DO
FOR J:= 1 TO DIM DIV 2 DO
BEGIN
Q[I,J]:= 0;
FOR K:= 1 TO DIM DIV 2 DO
Q[I,J]:= Q[I,J]+F[I,K]*R[K,J]
END
END;
BEGIN
M4.RECEIVE(HADM);
R4.RECEIVE(IM);
DIM:= HADM.SIZE;
WHILE (DIM >= 2) DO
BEGIN
MULTIPLY(HADM,MATvIMvRESULT1);
MULTIPLY(RESULT1,HADM,MATvRESULT2);
MM4 SEND(RESULT2);
M4.RECEIVE(HADM);
R4.RECEIVE(IM);
DIM:= HADM.SIZE;
END;
END;
PROGRAM RESFR;
VAR
H1,H2,H3,H4:REC; H1: RECTYPE;
I,J,X,DIM: INTEGER;
HEADING: RECTYPE2;
BEGIN
MM1.RECEIVE(H1);
MM2.RECEIVE(H2);
MM3.RECEIVE(H3);
MM4.RECEIVE(H4);
MM.RECEIVE(HEADING);
DIM:= H1.SIZE;
WHILE (DIM >= 2) DO
BEGIN
WRITE(' NODE:HEADING,NOD:3);
WRITE(' AT LEVEL:HEADING.LEV:3);
IF (HEADING.INFO = FALSE) THEN
  WRITELN(' IS NOT INFORMATIVE (TO TRANSMITTER)');
ELSE
  BEGIN
    WRITELN(' IS INFORMATIVE (TO TRANSMITTER)');
    WRITELN(' THE CORRESPONDING TRANSFORMATION MATRIX IS');
    WRITELN;
    A:= DIM DIV 2;
    FOR I:= 1 TO DIM DO
      BEGIN
        FOR J:= 1 TO DIM DO
          BEGIN
            CASE I=A OF
              TRUE: IF J=A THEN HI[I,J]:= H1.MAT[I,J]
                ELSE HI[I,J]:= H2[I,J-A];
              FALSE:IF J=A THEN HI[I,J]:= H3[I-A,J]
                ELSE HI[I,J]:= H4[I-A,J-A];
            END;
            WRITE(HI[I,J]:7)
          END;
        WRITELN
      END;
    MM1.RECEIVE(H1);
    MM2.RECEIVE(H2);
    MM3.RECEIVE(H3);
    MM4.RECEIVE(H4);
    MM.RECEIVE(HEADING);
    DIM:= H1.SIZE;
  END;
END;
END;

MONITOR: CHANNEL;
VAR
  BUFF: BUFFTYPE; FULL: BOOLEAN; INSTANCE QUEUE: CONDITION;
PROCEDURE *SEND(LINE: BUFFTYPE);
BEGIN
  IF FULL THEN QUEUE.WAIT;
  BUFF:= LINE; FULL:= TRUE; QUEUE.SIGNAL
END;
PROCEDURE *RECEIVE(VAR LINE:BUFFTYPE);
BEGIN
  IF NOT FULL THEN QUEUE.WAIT;
  LINE:= BUFF; FULL:= FALSE; QUEUE.SIGNAL
END;
BEGIN
  FULL:= FALSE;
  ***
END;
APPENDIX G

This program implements the regular decomposition algorithm of Chapter 4. More specifically, it implements the efficient algorithm described by the flowchart in Figure 4.6.1. Here we use a so-called dynamic stack using Pascal pointers in order to add or delete an item from the stack (using new (p) or dispose (p)). This way we keep track of only "not sure" nodes on the stack. A full explanation is given in Section 4.6. The input is read from an external file called DATAFILE. We specify the threshold values TETA1 and TETA2, the minimum and maximum levels MINLEVEL and MAXLEVEL, and the image size MAXSIZE. The program to build a tree structure is shown in Appendix H.
PROGRAM DECOMP(DATFILE, INPUT, OUTPUT); CONST
  MINLEVEL = 0; MAXLEVEL = 4;
  MAXSIZE = 32;
TYPE
  ATYPE = ARRAY[1..15001] OF INTEGER;
  M2 = ARRAY[1..32,1..32] OF INTEGER;
  RECTYPE = RECORD
    MAT: M2;
    SIZE: INTEGER
  END;
  PTRTONODE = PNODE;
  NODE = RECORD
    INDEX: INTEGER;
    LINK: PTRTONODE
  END;
VAR
  DATFILE: TEXT;
  TETA1, TETA2: REAL;
  TREE: ATYPE;
  P, FIRST, TEMPFIRST, SAVEFIRST: PTRTONODE;
  LEVEL, FATHER, & TEMP: INTEGER;
  FIRSTELEMENT: BOOLEAN;
FUNCTION FOUR(B: INTEGER): INTEGER;
BEGIN
  FOUR := TRUNC(EXP(B*LN(4)) + 0.5);
END;
PROCEDURE BUILDO IN LIBRARY;
PROCEDURE NOTINFORMATIVE(LEVEL, NODE: INTEGER);
BEGIN
  WRITE(' ', NODE:3, ' AT LEVEL ', LEVEL+1:3);
  WRITE(' IS NOT INFORMATIVE'); WRITEln
END;
PROCEDURE PUSHONSTACK(LEVEL: INTEGER);
VAR
  START, UPERMOST: INTEGER;
BEGIN
  START := (FOUR(LEVEL)+2) DIV 3;
  UPERMOST := START + FOUR(LEVEL);
  WHILE (START <= UPERMOST) DO
  BEGIN
    NEW(P);
    P^.INDEX := START;
    P^.LINK := FIRST;
    FIRST := P;
    START := START + 1
  END;
END;
PROCEDURE CHECKSONS(FATHER, LEVEL: INTEGER);
VAR
  J: INTEGER;
  AVG: REAL;
FUNCTION FIRSTSON(FATHER: INTEGER): INTEGER;
VAR
  RANK: INTEGER;
BEGIN
  RANK := FATHER - (FOUR(LEVEL) + 2) DIV 3;
  FIRSTSON := (FOUR(LEVEL+1) + 2) DIV 3 + 4*RANK
END;
BEGIN
1160 TEMPFIRST := FIRST;
1170 FOR J := FIRSTSON(FATHER) + 3 DOWNTO FIRSTSON(FATHER) DO
1185 BEGIN
1195 AVG := TREE[J]/TREE[FATHER];
1215 IF (AVG > 0) AND (AVG < TETA1) THEN
1223 NOTINFORMATIVE(LEVEL, J)
1233 ELSE
1239 IF (AVG = TETA2) AND (AVG = TETA1) THEN
1257 BUILD(LEVEL + 1, J)
1257 ELSE
1264 BEGIN
1264 NEW(P);
1271 P@.INDEX := J;
1278 P0.LINK := TEMPFIRST;
1290 IF FIRSTELEMENT = TRUE THEN
1299 BEGIN
1303 P0.LINK := NIL;
1307 FIRSTELEMENT := FALSE
1307 END;
1313 TEMPFIRST := P
1313 END;
1323 END;
1328 FIRST := TEMPFIRST
132B END;
1460 RESET(DATAFILE);
1465 WRITELN('THE STARTING LEVEL IS ':MINLEVEL:4):
1520 WRITELN;
1528 WRITELN('THE MAXIMUM LEVEL IS ':MAXLEVEL:4):
1580 WRITELN;
1588 READLN(TETA1,TETA2);
1608 WRITE('TETA1 IS ':TETA1:5:2);
1647 WRITE('TETA2 IS ':TETA2:5:2);
1687 WRITELN;
1695 FOR I := 1 TO (MAXSIZE-MAXSIZE REM 2 + 1) DO
1714 BEGIN
1714 READ(DATAFILE, TEMP);
1723 TREE[I] := TEMP;
1723 END;
1757 PUSHONSTACK(MINLEVEL);
1759 LEVEL := MINLEVEL;
1766 REPEAT
1766 FIRSTELEMENT := TRUE;
1772 SAVEFIRST := FIRST;
1782 REPEAT
1792 FATHER := SAVEFIRST@.INDEX;
1793 WRITELN;
1797 WRITELN('FATHER = ':FATHER:1);
1817 WRITELN;
1851 P := SAVEFIRST;
1861 SAVEFIRST := SAVEFIRST@.LINK;
1872 DISPOSE(P);
1879 CHECKSONS(FATHER, LEVEL);
1891 UNTIL SAVEFIRST = NIL;
1900 LEVEL := LEVEL + 1;
1911 P := FIRST;
1921 WRITELN;
1929 WRITELN('NOT SURE NODES OF LEVEL 'LEVEL:2 ARE :');
1905 WRITELN;
1909 WHILE (P != NIL) DO
1918 BEGIN
1923 WRITE(P@.INDEX:1 :1'); P := P@.LINK
1936 END;
1949 WRITELN;
1961 WRITELN;
1961 UNTIL (FIRSTELEMENT = TRUE) OR (LEVEL = MAXLEVEL);
1979 END.
PROCEDURE BUILD(LEVEL, NODE: INTEGER):
VAR
  I, J, BETA, RANK, STR, QUADSIZE: INTEGER;
  IND: M2; VAL: RECTYPE;
FUNCTION TWO(A: INTEGER): INTEGER;
BEGIN
  TWO := TRUNC((EXP(A*LN(2)) + 0.5));
END;

PROCEDURE BUILD(NUM, START: INTEGER):
VAR
  I, J, A: INTEGER;
BEGIN
  A := TWO(NUM-1);
  FOR I := 1 TO TWO(NUM-1) DO
    FOR J := START TO TWO(NUM) DO
      BEGIN
        IND[I, J] := IND[I, J-A] + SORT(TWO(NUM-1));
        VAL[I, J] := TREE[IND[I, J]];
      END
END;

BUILD(1, TWO(1-1) + 1);
APPENDIX H

For the regular decomposition algorithm, the input as explained in Chapter 4 is a quadtree that represents the image in hierarchical form. The first program in this appendix (NICK) builds the one-dimensional tree from the corresponding two-dimensional image. First it builds the base of the tree which is the last level of the tree (procedure BUILDBASE). Then, it builds the lower resolution levels recursively (procedure BUILDNEXT). The same program has been implemented concurrently (PARBUILD). This program (second one in this appendix) builds the four branches of the tree concurrently. The image is first subdivided into four quadrants $P_1$, $P_2$, $P_3$ and $P_4$. Then, each process (PROC1, PROC2, PROC3 and PROC4) builds one branch of the tree corresponding to a quadrant of the image. At the end, the father of the whole tree is computed (process TREEMERGE).
PROGRAM NICK(IMGFER,OUTPUT);
CONST
MAXLEVEL = 5;
TYPE
MATRIX = ARRAY[1..32,1..32] OF INTEGER;
RECTYPE = RECORD
  IND: MATRIX;
  VAL: MATRIX;
END;
VAR
IMGFER: TEXT;
TREE: ARRAY[1..1500] OF INTEGER;
I, J, ALPHA, LEVEL: INTEGER;
PIX: RECTYPE;
FUNCTION TWO(A:INTEGER):INTEGER;
BEGIN
  TWO := TRUNC((EXP(A*LN(2)) + 0.5));
END;
PROCEDURE BUILDNEXT(LEVEL:INTEGER);
VAR
I, J, A, DUMMY: INTEGER;
BEGIN
  A := TWO(LEVEL-1); DUMMY := (SQR(TWO(LEVEL)) + 2) DIV 3;
  FOR I:= (SQR(A)+2) DIV 3 TO (SQR(A)+2) DIV 3 + SQR(A) - 1 DO
    BEGIN
      TREE[I] := 0;
      FOR J:= DUMMY TO DUMMY + 3 DO
        TREE[I] := TREE[I] + TREE[J];
      DUMMY := DUMMY + 4;
    END;
END;
PROCEDURE BUILDBASE(NUM, START: INTEGER);
VAR
I, J, A: INTEGER;
BEGIN
  A := TWO(NUM-1);
  FOR I:= 1 TO TWO(NUM-1) DO
    FOR J:= START TO TWO(NUM) DO
      BEGIN
        PIX.IND[I,J] := PIX.IND[I,J-A] + SQR(TWO(NUM-1));
        TREE[PIX.IND[I,J]] := PIX.VAL[I,J];
        PIX.IND[I,J] := PIX.IND[I,J-A] + TWO(2*NUM-1);
        TREE[PIX.IND[I,J]] := PIX.VAL[I+A,J-A];
        PIX.IND[I,J] := PIX.IND[I+A,J] + SQR.TWO(NUM-1);
        TREE[PIX.IND[I+A,J]] := PIX.VAL[I+A,J];
      END;
END;
BEGIN
RESET(IMGFER);
ALPHA := (SQR(TWO(MAXLEVEL)) + 2) DIV 3;
FOR I:= 1 TO TWO(MAXLEVEL) DO
  FOR J:= 1 TO TWO(MAXLEVEL) DO
    READ(IMGFER,PIX.VAL[I,J]);
PIX.IND[1,1] := ALPHA;
TREE[ALPHA] := PIX.VAL[1,1];
FOR I:= 1 TO MAXLEVEL DO
  BEGIN
    BUILDBASE(I,TWO(I-1) + 1);
    LEVEL := MAXLEVEL;
  END;
REPEAT
  BUILDNEXT(LEVEL);
  LEVEL := LEVEL - 1
  UNTIL LEVEL = 0;
  I := 1;
  WHILE (I <= (ALPHA + SQR(TWO(MAXLEVEL)) - 1)) DO 
    BEGIN
      WRITE(TREE[I],TREE[I+1],TREE[I+1+5],TREE[I+23],TREE[I+33],TREE[I+43],
            TREE[43],TREE[I+53],TREE[I+63]);
      I := I + 7;
    END;
END;
END.
PROGRAM PARBUILD(IMGBUFFER, OUTPUT);
CONST IMSIZE = 8;

TYPE
  MATRIX = ARRAY[1..32, 1..32] OF INTEGER;
  ONEDIM = ARRAY[1..100] OF INTEGER;

VAR
  IMGBUFFER: TEXT;

MONITOR CHANN=CHANNEL IN LIBRARY(WHERE TYPE BUFFTYPE=MATRIX);
  INSTANCE R1, R2, R3, R4: CHANN;
MONITOR CHAN2=CHANNEL IN LIBRARY(WHERE TYPE BUFFTYPE=ONEDIM);
  INSTANCE G1, G2, G3, G4: CHAN2;
PROCESS PROC3 IN LIBRARY;
  INSTANCE XH3: PROC3;
PROCESS PROC4 IN LIBRARY;
  INSTANCE XH4: PROC4;
PROCESS PROC1 IN LIBRARY;
  INSTANCE XH1: PROC1;
PROCESS PROC2 IN LIBRARY;
  INSTANCE XH2: PROC2;
PROCESS TREEMERGE IN LIBRARY;
  INSTANCE XTREE: TREEMERGE;
PROCESS DIVIDE IN LIBRARY;
  INSTANCE XDIV: DIVIDE(IMGBUFFER, IMSIZE);
BEGIN
  RESET(IMGBUFFER);
  WRITE('THE IMAGE IS OF SIZE: ');
  WRITELN(IMSsize: 3);
  WRITELN(IMSsize: 3);
  ***
END.

PROCESS TREEMERGE(INF: TEXT; DIM: INTEGER);
VAR
  I, J, TEMP, A: INTEGER;
  P1, P2, P3, P4: MATRIX;
BEGIN
  RESET(INF);
  WRITE('THE IMAGE IS THE FOLLOWING: ');
  WRITELN;
  A := DIM DIV 2;
  FOR I := 1 TO DIM DO
  BEGIN
    FOR J := 1 TO DIM DO
    BEGIN
      READ(INF, TEMP);
      WRITE(TEMP: 6, ' ');
      IF (I <= DIM DIV 2) AND (J < DIM DIV 2) THEN
        P1[I, J] := TEMP;
      IF (I <= DIM DIV 2) AND (J > DIM DIV 2) THEN
        P2[I, J-A] := TEMP;
      IF (I > DIM DIV 2) AND (J <= DIM DIV 2) THEN
        P3[I-A, J] := TEMP;
      IF (I > DIM DIV 2) AND (J > DIM DIV 2) THEN
        P4[I-A, J-A] := TEMP;
    END;
  END;
END.
IF (I>DIM DIV 2) AND (J<=DIM DIV 2) THEN
P3[I-A,J]= TEMP;
IF (I>DIM DIV 2) AND (J>DIM DIV 2) THEN
P4[I-A,J-A]= TEMP;

END;
WRITEln
END;
WRITEln; WRITEln;
R1.SEND(P1);
R2.SEND(P2);
R3.SEND(P3);
R4.SEND(P4);
END;
PROCESS PROC1;
CONST
MAXLEVEL = 3;

TYPE
RECTYPE = RECORD
  IND: MATRIX;
  VAL: MATRIX;
END;

VAR
  TREE: ONEIDIM;
  I, J, BB, ALPHA, BETA, LEVEL: INTEGER;
  PIX: RECTYPE;
FUNCTION TWO(A:INTEGER):INTEGER
BEGIN
  TWO:= TRUNC((EXP(A*LN(2))+0.5));
END;
PROCEDURE BUILDNEXT(LEVEL:INTEGER);
VAR
  I, J, A, DUMMY, START, OFFSET1, OFFSET2: INTEGER;
BEGIN
  A:= TWO(LEVEL-1);
  DUMMY:= (SQR(TWO(A)) DIV 3);
  OFFSET1:= (BB*(SQR(TWO(LEVEL))) DIV 4);
  START:= (SQR(A) + 2) DIV 3 + OFFSET1;
  DUMMY:= DUMMY + OFFSET2;
  FOR I:= START TO START + (SQR(A) DIV 4) - 1 DO
    BEGIN
      TREE[I]:= 0;
      FOR J:= DUMMY TO DUMMY + 3 DO
        TREE[I]:= TREE[I] + TREE[J];
      DUMMY:= DUMMY + 4;
    END;
END;
PROCEDURE BUILDBASE(NUM, START: INTEGER);
VAR
  I, J, A: INTEGER;
BEGIN
  A:= TWO(NUM-1);
  FOR I:= 1 TO TWO(NUM-1) DO
    FOR J:= START TO TWO(NUM) DO
      BEGIN
        PIX.IND[I+J]:= PIX.IND[I+J-A] + SQR(TWO(NUM-1)) +
        TREE[PIX.IND[I+J]]:= PIX.VAL[I+J];
        PIX.IND[I+J+A-J-A]:= PIX.VAL[I+A-J-A];
        PIX.IND[I+A-J]:= PIX.IND[I+A-J-A] + SQR(TWO(NUM-1));
        TREE[PIX.IND[I+A-J]]:= PIX.VAL[I+A-J];
      END;
    END;
BEGIN
  R1.RECEIVE(PIX.VAL);
  BB:= 0; BETA:= BB*2*(SQR(TWO(MAXLEVEL))) DIV 4;
  ALPHA:= (SQR(TWO(MAXLEVEL)) + 2) DIV 3;
  PIX.IND[1,1]:= ALPHA + BETA + TREE[PIX.IND[1,1]]:= PIX.VAL[1,1];
  FOR II:= 1 TO MAXLEVEL - 1 DO
    BUILDBASE(I,TWO(I-1)+1);
  LEVEL:= MAXLEVEL;
  REPEAT
    BUILDNEXT(LEVEL);
    LEVEL:= LEVEL - 1
    UNTIL LEVEL = 1;
  GO1.SEND(TREE);
END;
PROCESS PROC3;
CONST
  MAXLEVEL = 3;
TYPE
  RECTYPE = RECORD
    IND: MATRIX;
    VAL: MATRIX;
  END;
VAR
  TREE: ONEDIM;
  I, J, BB, ALPHA, BETA, LEVEL: INTEGER;
  PIX: RECTYPE;
FUNCTION TWO(A:INTEGER):INTEGER;
BEGIN
  TWO:= TRUNC((EXP(A*LN(2)) + 0.5));
END;
PROCEDURE BUILDNEXT(LEVEL:INTEGER);
VAR
  I, J, A, DUMMY, START, OFFSET1, OFFSET2: INTEGER;
BEGIN
  A:= TWO(LEVEL-1); DUMMY:= (SQR(TWO(LEVEL)) + 2) DIV 3;
  OFFSET1:= (BB*(SQR(A) DIV 4));
  OFFSET2:= (BB*(SQR(TWO(LEVEL)) DIV 4));
  START:= (SQR(A) + 2) DIV 3 + OFFSET1;
  DUMMY:= DUMMY + OFFSET2;
  FOR I:= START TO START + (SQR(A) DIV 4) - 1 DO
  BEGIN
    TREE[I] := 0;
    FOR J:= DUMMY TO DUMMY + 3 DO
      TREE[I] := TREE[I] + TREE[I];
    DUMMY:= DUMMY + 4;
  END;
END;
PROCEDURE BUILDBASE(NUM, START: INTEGER);
VAR
  I, J, A, INTEGER;
BEGIN
  A:= TWO(NUM-1);
  FOR I:= 1 TO TWO(NUM-1) DO
    FOR J:= START TO TWO(NUM) DO
    BEGIN
      FIX.IND[I, J] := FIX.IND[I, J] + SQR(TWO(NUM-1));
      TREE[PIX.IND[I, J]] := TREE[PIX.IND[I, J]+A];
      FIX.IND[I+A, J] := FIX.IND[I+A, J] + TWO(TWO(NUM-1));
      TREE[PIX.IND[I+A, J]] := TREE[PIX.IND[I+A, J]];  
      FIX.IND[I+A, J] := FIX.IND[I+A, J] + SQR(TWO(NUM-1));
      TREE[PIX.IND[I+A, J]] := TREE[PIX.IND[I+A, J]];
    END;
  END;
BEGIN
  X2.RECEIVE(PIX.VAL);
  BB:= 1; BETA:= BB*(SQR(TWO(MAXLEVEL)) DIV 4);
  ALPHA:= (SQR(TWO(MAXLEVEL)) + 2) DIV 3;
  FIX.IND[1, 1] := ALPHA + BETA; TREE[PIX.IND[1, 1]] := PIX.VAL[1, 1];
  FOR I:= 1 TO MAXLEVEL - 1 DO
    BUILDBASE(1, TWO(I-1) + 1);
  LEVEL:= MAXLEVEL;
  REPEAT
    BUILDNEXT(LEVEL);
  LEVEL:= LEVEL - 1
  UNTIL LEVEL = 1;
  X2.SEND(TREE);
END;
PROCESS PROC3;
CONST
  MAXLEVEL = 3;
TYPE
  RECTYPE = RECORD
    IND: MATRIX;
    VAL: MATRIX;
  END;
VAR
  TREE: ONEDIM;
  I, J, BB, ALPHA, BETA, LEVEL: INTEGER;
  PIX: RECTYPE;
FUNCTION TWO(A:INTEGER):INTEGER;
BEGIN
  TWO:= TRUNC((EXP(A#LN(2)) + 0.5));
END;

PROCEDURE BUILDNEXT(LEVEL:INTEGER);
VAR
  I, J, A, DUMMY, START, OFFSET1, OFFSET2: INTEGER;
BEGIN
  A:= TWO(LEVEL-1); DUMMY:= (SQR(TWO(LEVEL)) + 2) DIV 3;
  OFFSET1:= (BB*(SQR(A) DIV 4));
  OFFSET2:= (BB*(SQR(TWO(LEVEL)) DIV 4));
  START:= ((SQR(A) + 2) DIV 3) + OFFSET1;
  DUMMY:= DUMMY + OFFSET2;
  FOR II:= START TO START + (SQR(A) DIV 4) - 1 DO
    BEGIN
      TREE[I][J]:= 0;
      FOR J:= DUMMY TO DUMMY + 3 DO
        TREE[I][J]:= TREE[I][J] + TREE[I][J];
      DUMMY:= DUMMY + 4;
    END;
END;

PROCEDURE BUILDBASE(NUM, START: INTEGER);
VAR
  I, J, A: INTEGER;
BEGIN
  A:= TWO(NUM-1);
  FOR II:= 1 TO TWO(NUM-1) DO
    FOR J:= START TO TWO(NUM) DO
      BEGIN
        PIX.INDC[I][J]:= PIX.INDC[I][J-A] + SQR(TWO(NUM-1));
        TREEPIX.INDC[I][J]:= FIX.INDC[I][J];
        PIX.INDC[I][J-A]:= FIX.INDC[I][J-A] + SQR(TWO(NUM-1));
        TREEPIX.INDC[I][J-A]:= FIX.INDC[I][J-A];
        PIX.INDC[I][J-A]:= FIX.INDC[I][J-A] + SQR(TWO(NUM-1));
        TREEPIX.INDC[I][J-A]:= FIX.INDC[I][J-A];
      END;
END;

BEGIN
  R3.RECEIVE(PIX.VAL);
  BB:= 2; BETA:= BB*((SQR(TWO(MAXLEVEL))) DIV 4);
  ALPHA:= (SQR(TWO(MAXLEVEL)) + 2) DIV 3;
  PIX.INDC[I][J]:= ALPHA + BETA;
  FORII:= 1 TO MAXLEVEL - 1 DO
    BUILDBASE(I:TWO(I-1) +1);
  LEVEL:= MAXLEVEL;
  REPEAT
    BUILDNEXT(LEVEL);
  LEVEL:= LEVEL - 1;
  UNTIL LEVEL = 1;
  GG3.SEND(TREE);
END;

PROCESS PROC4;
CONST
  MAXLEVEL = 3;
TYPE
  RECTYPE = RECORD
    IND: MATRIX;
    VAL: MATRIX;
  END;
VAR
  TREE: ONEDIM;
  I, J, BB, ALPHA, BETA, LEVEL: INTEGER;
  PIX: RECTYPE;

BEGIN
  R4.RECEIVE(PIX.VAL);  
  BB:= 3;  BETA:= BB*((SQR(TWO(MAXLEVEL))) DIV 4);  
  ALPHA:= (SQR(TWO(MAXLEVEL)) + 2) DIV 3;  
  FIX.IND[1,1]:= ALPHA + BETA;  TREE[PIX.IND[1,1]]:= PIX.VAL[1,1];  
  FOR I:= 1 TO MAXLEVEL - 1 DO  
    BUILDBASE(i,TWO(i-1) + 1);  
  LEVEL:= MAXLEVEL;  
  REPEAT  
    BUILDNEXT(LEVEL);  
    LEVEL:= LEVEL - 1  
  UNTIL LEVEL = 1;  
  GG4.SEND(TREE);  
END

PROCESS TREEMERGE;  
CONST  
MAXLEVEL = 3;  
VAR  
H1, H2, H3, H4:ONEDIM;  
LEVEL, TEMP1, TEMP2, I:INTEGER;  
FUNCTION TWO(A:INTEGER): INTEGER;  
BEGIN  
  TWO:= TRUNC((EXP(A*LN(2)) + 0.5))  
END;  
BEGIN  
  GG1.RECEIVE(H1);  GG2.RECEIVE(H2);  
  GG3.RECEIVE(H3);  GG4.RECEIVE(H4);  
  WRITELN('THE TREE REPRESENTING THE ABOVE IMAGE IS');  
  WRITELN('BUILDING ITS FOUR BRANCHES SIMULTANEOUSLY');  
  LEVEL:= 1;  
  WRITELN;  WRITELN;  WRITELN;  
  WRITELN('LEVEL IS: '+(LEVEL-1):4);  WRITELN;  
  WRITELN(LEVEL:=8(H1C2tH2C3tH3C4tH4C5));  
  REPEAT  
    WRITELN;  WRITELN;  WRITELN;  
    WRITELN('LEVEL IS: '+(LEVEL-4):4);  WRITELN;  
    TEMP1:= SOR(TWO(LEVEL))  
    TEMP2:= (TEMP1 + 2) DIV 3;  
    FOR I:= TEMP2 TO TEMP2 + TEMP1 - 1 DO  
      BEGIN  
        IF (I<(TEMP2 + TEMP1 DIV 4)) THEN  
          WRITELN(I:8tH1I13I8)  
        ELSE  
          IF (I<(TEMP2 + 2*(TEMP1 DIV 4))) THEN  
            WRITELN(I:8tH1I13I8)  
          ELSE  
            IF (I<(TEMP2 + 3*(TEMP1 DIV 4))) THEN  
              WRITELN(I:8tH1I13I8)  
            ELSE  
              WRITELN(I:8tH1I13I8)  
            END;  
      END;  
      LEVEL:= LEVEL + 1;  
  UNTIL LEVEL > MAXLEVEL;  
END
THE IMAGE IS OF SIZE: 8

THE IMAGE IS THE FOLLOWING

```
 0 0 4 0 0 6 0 0
 0 0 0 0 0 0 0 2
 2 2 0 2 0 1 3 2
 1 2 2 3 0 1 0 0
 0 0 2 0 2 3 1 0
 1 0 2 0 0 4 3 2
 0 0 0 0 0 0 0 2
 0 0 0 0 0 3 0 0
```

THE TREE REPRESENTING THE ABOVE IMAGE IS
(BUILDING ITS FOUR BRANCHES SIMULTANEOUSLY)

LEVEL IS: 0

```
 1 66
```

LEVEL IS: 1

```
 2 18
 3 20
 4 6
 5 22
```

LEVEL IS: 2

```
 6 0
 7 4
 8 7
 9 7
10 6
11 2
12 6
13 6
14 2
15 4
16 0
17 0
18 9
19 6
20 5
21 2
```

LEVEL IS: 3

```
22 0
23 0
24 0
25 0
26 4
27 0
28 0
29 0
30 2
31 2
32 1
33 2
34 0
35 2
36 2
37 3
```