Real Time Texture Analysis from the Parallel Computation of Fractal Dimension

Halford I. Hayes Jr.
Old Dominion University

Follow this and additional works at: https://digitalcommons.odu.edu/computerscience_etds

Part of the Artificial Intelligence and Robotics Commons, Numerical Analysis and Scientific Computing Commons, and the Systems Architecture Commons

Recommended Citation
Hayes, Halford I.. "Real Time Texture Analysis from the Parallel Computation of Fractal Dimension" (1993). Master of Science (MS), Thesis, Computer Science, Old Dominion University, DOI: 10.25777/gkm8-3v43
https://digitalcommons.odu.edu/computerscience_etds/155

This Thesis is brought to you for free and open access by the Computer Science at ODU Digital Commons. It has been accepted for inclusion in Computer Science Theses & Dissertations by an authorized administrator of ODU Digital Commons. For more information, please contact digitalcommons@odu.edu.
Real Time Texture Analysis From the Parallel Computation of Fractal Dimension

by

Halford I. Hayes, Jr.

July 26, 1993

A Thesis submitted to the Faculty of Old Dominion University in Partial Fulfillment of the Requirement for the Degree of

MASTERS OF SCIENCE

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
REAL TIME TEXTURE ANALYSIS FROM THE PARALLEL COMputation OF FRACTAL DIMENSION

by

HALFORD IRVIN HAYES, JR.

B.S.A.E June 1980, University of Virginia
B.S.C.S December 1991, Old Dominion University

A Thesis submitted to the Faculty of Old Dominion University in Partial Fulfillment of the Requirement for the Degree of

MASTERS OF SCIENCE
COMPUTER SCIENCE
OLD DOMINION UNIVERSITY
July 26, 1993

Approved by:

Dr. James L. Schwing (Committee Chairman)

Dr. C. Michael Overstreet (Director)

Dr. Stephen Olariu (Committee Member)
The discrimination of texture features in an image has many important applications: from detection of man-made objects from a surrounding natural background to identification of cancerous from healthy tissue in X-ray imagery. The fractal structure in an image has been used with success to identify these features but requires unacceptable processing time if executed sequentially.

The paradigm of data parallelism is presented as the best method for applying massively parallel processing to the computation of fractal dimension of an image. With this methodology, and sufficient numbers of processors, this computation can reach real time speeds necessary for many applications. A model is analyzed and evaluated on several architectures: workstation, vectorizing supercomputer, shared-memory MIMD, and massively parallel SIMD computers. Per expectations, results in the subsecond range are attained on the massively parallel SIMD computer.
Acknowledgment

I first wish to thank my family, Maureen, Joshua, Jeremy, and Jonathan for their support during the long summer days when I could not be with them.

There is a long list of people who have provided me with encouragement and support in writing this. The following is thanks for those associated with the Advanced Computation Technology Group at NSWC. I thank Carey Priebe for suggesting this particular idea for research, and his prodding and encouragement. He sets a higher standard. Jeff Solka has been invaluable as my guide and confidant. His discussion on both technical and semantic issues vastly improved my unpolished writing. George Rogers deserves thanks for his challenging of my methods and observations. I appreciate the opportunity to have collaborated with him. John Ellis provided the original source code and many tools for these applications.

Robert Whaley, of Thinking Machines, Inc., at the Naval Research Laboratory, was invaluable in aiding me in programming my applications on the Connection Machine.

I thank my mentors at Old Dominion University: Dennis Ray, Dr. Michael Overstreet, and Dr. James Schwing. Your advice and counseling have been a great help to me over the years and kept me pointed in the direction I intended when I started attending classes over six years ago.

Finally, I am forever indebted to all the people at the Naval Surface Warfare Center without whose support none of this would have been possible.

I dedicate this work to my grandfathers: William Howard Hayes and James Theodore Tate. From them I learned inquisitiveness, humor, and love of learning.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
</tbody>
</table>

## CHAPTER

1. Introduction ........................................ 1

2. Describing Texture ..................................... 7
   2.1 Statistical ........................................ 8
   2.2 Structural ......................................... 9
   2.3 Spectral ........................................... 9
   2.4 Empirical Description of Texture ................ 10

3. Describing Texture with Fractals .................... 12
   3.1 Fractals ........................................... 12
   3.2 Measurement of a Coastline .................... 13
   3.3 Inferring Texture ............................... 17

4. Fractal Dimension Measurement ....................... 19
   4.1 Covering Method .................................. 20
   4.2 Application and Results ....................... 21
   4.3 Use of Results from Feature Extraction ....... 28
4.4 Other Approaches to Fractal Dimension Estimation Methods ................. 31

5. Parallel Approach ........................................ 33
  5.1 Complexity Analysis. ................................. 35
  5.2 Problem Size ........................................... 36
  5.3 Speedup Potential ..................................... 38
  5.4 Data Parallelism ....................................... 39
  5.5 Architecture Mapping ................................. 44
  5.6 Supposition on Implementation for Real Time Processing. ................. 46

6. Performance Results ....................................... 47
  6.1 Program Process ....................................... 44
  6.2 Performance Measurement .............................. 49
  6.3 Architecture Sampling ................................. 51
  6.4 Comparison on Different Computer Architectures ......................... 53
  6.5 Results for Varying Numbers of Processors on MIMD ..................... 56
  6.6 Results for Varying Numbers of Processors on the Connection Machine 60
  6.7 Speed Up Revisited .................................... 65

7. Texture Analysis Model for Computer Vision .............. 68
  7.1 Boundary Information ................................ 68
  7.2 Pre-attentive Vision .................................. 69
  7.3 Attentive Vision ...................................... 72
8. Conclusion ........................................ 75

BIBLIOGRAPHY ........................................ 77

APPENDICES

A. Definitions ........................................ A.1

B. Source Code

  Single Processor .............................. B.1

  Connection Machine .......................... B.24
**LIST OF TABLES**

<table>
<thead>
<tr>
<th>TABLE</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Examples of uses of Texture Analysis</td>
<td>4</td>
</tr>
<tr>
<td>2. Comparison of number of computations</td>
<td>37</td>
</tr>
<tr>
<td>3. Comparison Run Results for Fractal Dimension Computation</td>
<td>55</td>
</tr>
<tr>
<td>4. Comparative Run Results for Fractal Dimension Single vs. Multiple Processor</td>
<td>59</td>
</tr>
<tr>
<td>5. Compararative Run Results for Connection Machine</td>
<td>64</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Fractal Dimension in Pattern Recognition</td>
<td>3</td>
</tr>
<tr>
<td>2. Characteristics of rectifiable and fractal curves</td>
<td>14</td>
</tr>
<tr>
<td>3. Measurement of a Coastline</td>
<td>15</td>
</tr>
<tr>
<td>4. Texture Quilt</td>
<td>23</td>
</tr>
<tr>
<td>5. Texture Quilt Scale 2</td>
<td>24</td>
</tr>
<tr>
<td>6. Texture Quilt Scale 4</td>
<td>25</td>
</tr>
<tr>
<td>7. Texture Quilt With Selected Points</td>
<td>26</td>
</tr>
<tr>
<td>8. Plot of Surface Area Values for Selected Points</td>
<td>27</td>
</tr>
<tr>
<td>9. Class conditional probability distribution for quilt patch number 4</td>
<td>29</td>
</tr>
<tr>
<td>10. Class conditional probability distribution for quilt patch number 8</td>
<td>30</td>
</tr>
<tr>
<td>11. Methods for Parallel Processing of an Image</td>
<td>42</td>
</tr>
<tr>
<td>12. Image Processing Using Data Parallelism</td>
<td>43</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>13. Original Program Process for the Covering Method</td>
<td>48</td>
</tr>
<tr>
<td>14. Processing Time versus Image Size using the Covering Method</td>
<td>50</td>
</tr>
<tr>
<td>15. Comparison Runs: Single versus Multiple Processors</td>
<td>58</td>
</tr>
<tr>
<td>16. Connection Machine Time Results: 8k vs. 16k Processors</td>
<td>63</td>
</tr>
<tr>
<td>17. Speedup Comparison between 8k and 16k processors</td>
<td>67</td>
</tr>
<tr>
<td>18. Pre-attentive Textural Vision Model</td>
<td>71</td>
</tr>
<tr>
<td>19. Attentive Textural Vision Model</td>
<td>74</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

Machine or Computer Vision has become a strong area of research over the past decade. This is due primarily to the maturity and success in the development of algorithms for extracting information out of an image, and the increasing power of computing devices involved with implementing those algorithms.

An area of application of machine vision is Automatic Target Recognition (ATR). In ATR the user predetermines a target of interest and uses a computer vision system to find the target within an image. Though ominous sounding, this application is being researched for use in such areas as the identification of cancerous versus healthy tissue in X-ray imagery, and the identification of barnacles on the hull of ships. An ATR system requires a combination of feature extractor and discriminant analyzer to identify a target from surrounding background in an image.

I had the opportunity to work in this area of research with Jeffrey Solka, Dr. Carey Priebe, and Dr. George Rogers, at the Advanced Computation Technology Group of the Naval Surface Warfare Center at Dahlgren, Virginia, in the second half of 1992. The focus of their work has
been in the low-level area of machine vision. Their continuing interest has been in the formulation, design, and testing of algorithms which perform image segmentation based on textural clues.

Figure 1 illustrates the method for implementing texture analysis. An image is processed by a feature extraction (covering) method. The resulting output of the Covering method is a series of data points. Linear regression is performed on these data points for each pixel yielding three (or more) features. The pattern recognition subsystem performs discriminant analysis on these features, yielding classifications for each texture in the image.

This research has resulted in a capability to distinguish man-made objects from a natural background by using the measurement of fractal dimension of textures in an image to provide features for discriminant analysis [SOLKA 92]. This feature extraction method resulted in excessive processing times. The researchers standardized their images to a 480x512 pixel, 8-bit grayscale format. A typical time to process a standard image was over 33 minutes on a high-end graphical workstation.

This group was interested in determining if there were a way in which the processing time could be reduced, thus paving the way for actual applications of this technology. To be useful, the feature extraction process must be completed in sub-minute times for desktop applications (desktop), and sub-second times for real time applications (real time).
FIGURE 1. Fractal Dimension in Texture Analysis

Grayscale Image

Local Feature Extraction

Covering Method

Linear Regression

Discriminant Analysis & Classification
<table>
<thead>
<tr>
<th>Application</th>
<th>Purpose</th>
<th>Time Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smart Automobiles</td>
<td>Sensor system differentiates road surface</td>
<td>Real Time</td>
</tr>
<tr>
<td>Biomedical and X-ray Imagery Analysis</td>
<td>Discriminate between healthy and cancerous tissues</td>
<td>Desktop</td>
</tr>
<tr>
<td>Automatic Target Recognition</td>
<td>Find man-made object hiding or obscured by natural background</td>
<td>Real Time</td>
</tr>
<tr>
<td>Earth Sciences</td>
<td>Analysis of LANDSAT imagery for geological and biological patterns and formations</td>
<td>Desktop</td>
</tr>
<tr>
<td>Automated Inspection</td>
<td>Ship hull inspection for damage or barnacles</td>
<td>Real Time</td>
</tr>
</tbody>
</table>

This is the focus of my research. Analysis of the original method’s throughput time indicated that to meet the requirements of some possible applications, the feature extraction method needed to be, as in nature, steered to a parallel systems design. Using nature as our design guide, the best model is that of human vision.

The Human Visual System (HVS) provides one of the few reliable realizations of an automatic target recognition system. The HVS can perform multispectral segmentation followed by position, scale, and rotation invariant recognition in real time. Its high performance rate of processing can be attributed to its inherently parallel design [MEAD 89].
Many investigators have expressed that computer vision tasks require a considerable amount of computation, especially low-level or early vision. As Chaudhary, et al. [CHAUD 90], notes,

"Low-level vision is characterized by the local nature of its computations. There are few conceptual difficulties in designing parallel algorithm[s] for these tasks and several existing systems do many of these tasks in parallel. Intermediate level vision involves segmentation - a reduction of the incoming visual information to a form that will be effective for the recognition step of high-level vision. High-level vision involves the cognitive use of knowledge. In general, parallelism is not immediately evident in both intermediate and high-level vision."

**Problem Statement:** To do texture analysis on an image requires a texture feature extractor (the Covering method), as seen in Figure 1, which can be characterized as a low-level vision function. As such, the computational operations over the data set should be highly regular and local in nature. Image problems, as in this case, involve large data sets. Based upon these assumptions, what is the best method for producing real time throughput for this feature extractor and what is the most advantageous computing platform to implement it on?

The term real time can have many definitions depending upon the subject matter. For those applications listed in Table 1, it is intended to mean processing times in the range of (or less than) one second for a 512x512 pixel image. This would be an optimum speed for most applications. Some, such as vehicle navigation, would require processing times of much less than one second, while others could be satisfied with times in the subminute range.
The orientation of this thesis is in two parts. Chapters 2, 3, and 4 discuss the perception of texture, and means to extract it from grayscale images. Chapters 5, 6, and 7 discuss the implementation of the covering method for texture analysis with results from different hardware platforms.
CHAPTER 2

Describing Texture

The discrimination of texture in images is an active area of computer vision research. Texture plays an important part in scene analysis and corresponds to the physical surface of an object. In an image, it is characterized by a non-uniform spatial distribution of intensities. In a gray-scale picture, these intensities would be represented by pixels varying in shades of gray ranging from black (lowest) to white (highest).

Using texture and contrast information, the HVS can segregate objects, determine shape, and even the spatial orientation among objects. It has even been linked to the determination of motion[LANDY 91]. As mentioned previously, texture analysis has been used successfully to discriminate man-made from natural objects [SOLKA 92][PELI 90].

Texture discrimination is considered a low-level vision function. In computational vision, we treat texture as a two-dimensional pattern of intensities, and do not consider the physical basis of its underlying form. Modeling texture by its physical variation is in practice difficult to do. Rather, we would wish to determine a technique which would identify textures in a manner
similar to the HVS, allowing us to rapidly compute and discriminate texture regions in an image.

The ability to describe textures until recently has been based upon empirical analysis. A brief survey of such texture analysis techniques indicates the difficulty in describing texture. We can classify these empirical methods by the principle approaches for texture analysis identified by Gonzalez and Wintz: "Statistical, Structural, and Spectral" [Gonzalez 87].

2.1 Statistical

Statistical approaches attempt to describe textures by analysis of the distribution of gray level intensities. It is derived from the observation that certain properties occur repeatedly throughout a texture region. These properties can be analyzed through such statistical techniques as histogramming and the measurement of the variance of intensities. Additionally, spatial relationships among nearest-neighbor pixels, such as co-occurrence analysis, have been used to model textures. For a more in-depth survey of co-occurrence statistical techniques, Davis provides an excellent description of pixel-based texture modeling through the use of co-occurrence operators [Davis 81]. The work by Haralick, et al., also includes intensive research in this method [Haralick 73].

The limitations of such analysis can occur when trying to match possibly similar textures in different images under different lighting conditions. The HVS has little difficulty matching the texture from two pictures even with different shadings due to light levels. As an example, consider a picture of a grass lawn. Though one picture maybe "light" and the other "dark", we
can still discern the texture of the grass in the different pictures. This difference in variations in shading will have a severe impact on such statistical approaches.

2.2 Structural

Structural analysis involves the use of texture primitives which can be developed into more complex generalized patterns by means of some rule base that then can be used to describe real textures. An example of a formal mathematical image model can be seen in the work by Zucker, who develops primitive and ideal descriptors of real textures [ZUCK 76].

In a comprehensive approach to identifying textures through structural analysis, Vilnrotter, et al., defines a method requiring the following steps: the isolation of texture primitives, description of primitives, and the description of the interrelationship of these primitives [VILN 86]. The authors claim that the model has proven successful in discriminating a variety of textures. However, the complexity of the method is apparent by the necessity of the various features which must be extracted from the image. Each feature results in an additional matrix of data, (i.e.: edge and direction images). Each additional feature extraction applied to an image results in an increase in the overall computational complexity. It is doubtful that such methods would be invariant to both shading and scale, as is the HVS.

2.3 Spectral

This involves the use of the Fourier spectrum to describe the directionality of periodic two-dimension patterns in an image. Gonzalez and
Wintz provide a good summation of such techniques. They describe the following three features from the Fourier spectrum as useful in texture description [GONZ 87]:

“(1) prominent peaks in the spectrum give the principal direction of the texture patterns; (2) the location of the peaks in the frequency plane gives the fundamental spatial period of the patterns; and (3) by eliminating any periodic components via filtering we are left with nonperiodic image elements, which can then be described by statistical techniques.”

From this author’s perspective, the limitations of spectral techniques lies (1) in there inability to extract information from non-periodic texture patterns; and (2) in the resulting complex descriptors that are derived, such as spectrum plots, which must be correlated between the target texture and a library of texture features.

2.4 Empirical Description of Texture

This discussion of empirical texture techniques is not meant as an exhaustive survey, but rather demonstrates the various methods available as a comparison to the use of fractal dimensional. For further information the reader is invited to investigate the references mentioned within.

It can be argued that each of the three approaches has been shown to be successful in deriving textures from images. Why then is texture analysis still an ongoing area of research in computer vision? I believe that the reason lies in the lack of a robust description approach which can be implemented without an ad hoc modification to the feature extractor either because of the
conditions under which the images were taken (light versus dark), or because of the fine, and often unclassifiable structure of the texture itself. What is required for real time applications is a system for calculating texture that is both fast and reliable, that produces simple features that can be extracted both locally and globally in an image.

It is the opinion of this and other authors that the best method is one that models nature. That method is the derivation of the fractal dimension for a texture region.
CHAPTER 3

Describing Texture with Fractals

The concept of employing fractal geometry to extract texture-related information from a gray-scale image has been studied extensively [PELEG 84][PENT 84] [PELI 90] [SOLKA 92]. The following discussion is meant to provide the reader with a general background in this area. The concept of fractals is developed, and its application to textures is discussed.

3.1 Fractals

The geometry of nature has recently undergone a paradigm shift. No longer do we consider most objects in nature to be of Euclidean geometrical design. Many are so complex and erratic that such methods are hopeless. Indeed, the natural world disdains from defining itself in terms of Euclidean geometry.

Self-similarity, or invariance to changes in scale, is an attribute of many laws of nature and phenomena in the world around us. “Self-similarity is, in fact, one of the decisive symmetries that shape our universe and our efforts to comprehend it...Self-similarity often engenders highly fractured structures, called fractals by Benoit B. Mandelbrot, that abound in the world in and around us [SCHROED 89].”
However, fractal geometry has shown great success in the characterization of such plants, landscapes, and clouds. One only has to consider the “Black Spleenwort fern” image produced by fractal methods in Barnsley’s book, Fractals Everywhere, to see the power of this method [BARN 88].

Mandelbrot, considered one of the founding fathers of this field defines a fractal as any object whose fractal dimension exceeds its topological dimension [MAND 83]. We will see from the following example the concepts of invariance to scale and non-integer dimension.

3.2 Measurement of a Coastline

Let us see how it is possible for an object to have a non-integer dimension. Consider a curve in Euclidean space. The rectifiability of this curve, or the measurement of the arc length, \( \lambda \), by straight line segments, \( \eta \), approaches a finite limit as \( \eta \) approaches zero, or as \( p \), the number of line segments, goes to infinity. Rectifiability can be understood simply by graphing \( \lambda \) as a function of \( \eta \), such that

\[ \lambda \to L \]

as

\[ \eta \to 0 . \]

(Note: Mehaute gives a more detailed analysis of rectifiability in Fractal Geometries: Theory and Applications for Figure 2 [MEHAUTE 90].)
FIGURE 2. Characteristics of rectifiable (R) and fractal (F) curves. The concept of non-integral dimension is related to the slope of a fractal curve. [MEHAUTE 90]
FIGURE 3. Measurement of a Coastline
Yet, if we consider the measuring of a coastline, no length $\eta$ will be able to circumscribe its length, since as the size of $\eta$ diminishes, our measurement of the coastline grows without bound. For example, consider the distance one would compute for two harbor towns in figure 3, A and B, along a coast if you were to measure first with an instrument that measures in miles, then one that measures in feet, and finally one that measures in inches. For the fractal coastline, $F$, Figure 2 says that the slope is invariant to the scale of our measurement. This invariance to scale of measurement is one of the descriptors of fractals. If we characterize the slope of the fractal $F$ as $1 - \Delta$, we can describe the fractal dimension as $\Delta$. Given that a Euclidean line is described as one dimensional, the $\Delta$ for a fractal line will always be $1 < \Delta < 2$. 
3.3 Inferring Texture

This description can be applied to two dimensional objects as well. We can define a surface as a two-dimensional grid of values, such as a grayscale image. This image can be thought of as a surface in three space. The intensity of pixels would provide the measure of the height of the surface. Under our present description, the surface would have a topological dimension of 2 and a fractal dimension of $2 \leq D < 3$.

For such surfaces, the fractal dimension can be shown to be correlated with our perception of roughness. Pentland first reported that the variation of fractal dimension corresponds to our intuitive notion of the degree of roughness [PENT 84].

Falconer defines fractals as objects with some or all of the following properties: fine structure (i.e. detail on arbitrarily small scales) too irregular to be described with Euclidean geometry; self-similar structure, and recursively defined [FALC 90]. These are exactly the kinds of properties we observe in textures.

Peli's research in the characterization of man-made and natural objects from two-dimensional data, [PELI 90], noted that textures “...are various gray level surfaces that are self-similar over a limited range of scales, and therefore their corresponding fractal dimensions are stable ... over a small number of scales.”
Additionally, Peli notes from her experimentation that the fractal signature of natural objects has a relatively constant signature, while man-made objects change rapidly with scale. This differentiation is exploited by Solka, et al., to provide attributes for feature space in the development of a method for discriminant analysis of textures regions in an image [SOLKA 92].

In Pentland’s landmark discussion of fractals and the imaging process, “Fractal-Based Description of Natural Scenes” [PENT 84], he states,

“Fractal functions seem to provide a good model for describing the rough, crenulated, and crumpled 3-D surfaces typical of natural scenes. The evidence in support of this assertion is as follows:

1. Many basic physical processes produce fractal surfaces.

2. Fractal surfaces look like natural surfaces, and thus appear to capture all of the shape structure relevant to human perception.

3. We have conducted a survey of natural imagery and found that a fractal model of imaged 3-D surfaces, when transformed by the image formation process, furnishes an accurate description of both textured and shaded regions in most natural imagery.”

The concept of fractal geometry therefore, we can conclude, is useful in describing 2-D images of 3-D surfaces. The fractal signature gives us a useful description of the texture of a region that is invariant both in orientation and over a limited range of scales. (See the case study of the successful discrimination of texture regions from different orientations in aerial images by Priebe, et al., entitled, “Discrimination Analysis in Aerial Images Using Fractal Based Features” [PRIEBE 93]).
CHAPTER 4

Fractal Dimension Measurement

Virtually all schemes to estimate the fractal dimension use Richardson's law. Richardson's law tells us the manner in which a quantity measured on a fractal, \( M(\varepsilon) \), varies as a function of the scale of the measuring device, \( \varepsilon \), the topological dimension \( d \), and the fractal dimension \( D \). Richardson's law is given below as

\[
M(\varepsilon) = K \varepsilon^{(d-D)}.
\]

(EQ 1)

where \( K \) is a constant of proportionality. If we take the log of both sides of this equation, we can demonstrate the log-linear relationship between \( M(\varepsilon) \) and \( D \),

\[
\log[M(\varepsilon)] = \log(K) + (d-D) \log(\varepsilon).
\]

(EQ 2)

So if we were to measure the length of our hypothetical coast line using rulers with different scales we could then estimate \( D \) using a simple linear regression scheme.
4.1 Covering Method

At first the subject of grayscale texture feature extraction may seem far removed from our previous discussion. However, it seems at least intuitively pleasing to suppose that there would be a relationship between the texture embodied in a grayscale image and the fractal dimension of the image. The fractal dimension of a grayscale image can best be understood by realizing that the image can be thought of as a surface in three space. This surface would have a topological dimension of 2 and a fractal dimension $D \geq 2$.

In order to estimate the fractal dimension of the image, one needs to measure some quantity at various scales so that Richardson's law can be exploited. Our work has used the surface area in a window about a pixel as the quantity of interest. The method that is used to estimate the surface area at a given scale is known as the covering method. The covering method originated in the work of Peleg [PELEG 84], however the version of the method that we use was first described by Peli [PELI 90]. This method makes extensive use of grayscale morphological processing. The first step in the process is the computation of the dilation and erosion of our original image $G$. First the dilation and erosion of $G$ at scale 0 and pixel location $(i, j)$ is defined naturally as

$$U(i, j, 0) = L(i, j, 0) = G(i, j),$$  \hspace{1cm} (EQ 3)

where we have used $U$ to represent the dilation and $L$ to represent the erosion. In general, we form the dilation at scale $\varepsilon$ as

$$U(i, j, \varepsilon + 1) = \max_{d[(i, j), (l, m)] \leq 1 \{ U(i, j, \varepsilon) + 1, U(l, m, \varepsilon) \}},$$  \hspace{1cm} (EQ 4)
Similarly we define the erosion as

\[ L(i,j, \varepsilon + 1) = \min_{d[(i,j),(l,m)]} \{ L(i,j, \varepsilon) - 1 \cdot L(l,m, \varepsilon) \} \]  \quad (EQ 5)\]

As can be gleaned from these definitions these morphological computations are inherently parallel in nature, and there parallelization has been previously studied [STERN 82]. Given the upper and lower surfaces, one estimates the surface area at a given pixel \((i,j)\) and scale \(\varepsilon\) as

\[ A(i,j, \varepsilon) = \sum_{(l,m) \in W(i,j)} \frac{(U(l,m, \varepsilon) - L(l,m, \varepsilon))}{2\varepsilon} \]  \quad (EQ 6)\]

where \(W(i,j)\) is a pixel window centered at \((i,j)\). The fractal dimension can be recovered from a plot of \(\log[A(\varepsilon)]\) versus \(\log(\varepsilon)\). This pattern of operations are performed on each pixel in the image.

It is apparent from observation of equations 4-6 that the covering method consists of regular operations performed over the entire image. This is an important consideration in mapping it to parallel architectures.

### 4.2 Application and Results

The best way to see the covering method is through application of it over an image. The output from Equation 6 can be displayed graphically to allow analysis of the effects these calculations. Figure 4 is a texture quilt that has served as a benchmark for general texture discrimination work.

The results of each surface area calculation is visually displayed in Figure 5 and 6. These are the results for the surface area calculations for
scales $\xi = 2$, $\xi = 4$. Notice the myopic blurring effect as the scale increases from the original picture.

A program was developed for use as a tool in the research of the covering method that allows the user to pick points within the image and generate individual pixel plots of the surface area values. Notice the points labeled 1 through 6 in Figure 7. They were selected to give a good representation of the texture up to the boundary of the next texture patch.

Figure 8 is a plot of the points. Observe the separation between plots of the points selected in the two texture patches. This is indicative of the ability to discriminate between these two textures.

Not all textures yield this amount of separation. Additional work in this area is being conducted to use contrast boundary information within the covering method computation to reduce contamination of texture information across boundaries [ROGERS 93].
FIGURE 4. Sixteen texture quilt.
FIGURE 5. Texture Quilt Scale 2
FIGURE 6. Texture Quilt Scale 4
FIGURE 7. Texture Quilt with Selected Points
FIGURE 8. Plot of Surface Area Values for Selected Points
4.3 Use of Results in Feature Extraction

Though not the focus of this work, some cursory discussion of the complete feature extraction process seems appropriate. Instead of the fractal dimension, we have used the closely related slope of the regression line, the y intercept of the line, and the log of the F-test which represents the significance of the regression hypothesis, as our texture features. Given these features sets for various texture types, we have employed parametric and nonparametric probability density estimation techniques to model their posterior probability distribution. These distributions are then used as the building blocks for standard Bayesian classifiers.

This approach has been applied to the general texture discrimination problem [SOLKA 93], to aerial image processing problems [PRIEBE 93], and to the general question as to the discrimination between man-made structures and natural backgrounds [SOLKA 92]. For example consider the 16 texture quilt of Figure 4. We number the 16 patches in ascending order from left to right and top to bottom starting at 0. Figure 9 portrays a graph of the class conditional probability distribution of a set of (slope, y-intercepts) pairs obtained from patch number 4. The estimate of the probability density function (pdf) was obtained using the nonparametric probability density estimation procedure of Priebe and Marchette [PRIEBE 91]. Figure 10 presents a plot obtained in an identical manner for patch number 8. It is instructive to notice that even though there is some visual similarity between the two textures, there is still some separation of the two pdfs in feature space.
FIGURE 9. Class conditional probability distribution for quilt patch number 4, modeled with the adaptive mixtures procedure. Feature 2 = y-intercept and Feature 1= slope. Axis labels indicate graphics bins and not feature values.
FIGURE 10. Class conditional probability distribution for quilt patch number 8, modeled with the adaptive mixtures procedure. Feature 2 = y-intercept and Feature 1 = slope. Axis labels indicate graphics bins and not feature values.
4.4 Other Approaches to Fractal Dimension Estimation Methods

Sarkar and Chaudhuri detailed a summary of fractal dimension estimation techniques for texture analysis [SARK 92]. They have proposed a new method derived from the box-counting approaches which is computationally efficient and consistently gives satisfactory results on both synthetic and natural data.

Other methods include previously mentioned Peleg, et al, [PELEG 84]. They derived a set of features using the “covering blanket” method of estimation of fractal dimension suggested by Mandelbrot. These features are used as characteristics to recognize patches of natural texture. Peli’s method [PELI 90] is based upon the covering blanket methodology, but is more robust with regards to image noise. In recent discussions with Dr. Peli, she indicated that this method also shows invariance to lighting. This is a striking similarity to the HVS.

Box counting methods have been developed by Pickover and Khorrasani [PICK 86], to find the fractal dimension to characterize speech wave graphs, and Gangepain and Roques-Carmes [GANGE 86] for the measure of image intensity surfaces. Keller, et al., used fractal dimension as one of the tools for image segmentation based upon this method [KELLER 89].

Pentland developed an approach based upon using the Fourier power spectrum of fractal Brownian motion [PENT 84].

Sarkar and Chaudhuri examine each of these approaches in terms of their computational intensity and accuracy in calculating fractal dimension.
The method developed by Peleg, et al., compares favorably in terms of accuracy, but is by far the most computationally intensive method. The covering method, derived from this approach, is also computationally intensive. However, the covering method, as a low-level vision processing system, should yield considerable speedup when implemented on more than one processor. The next chapter will describe a paradigm which can be applied that can yield real time processing capability.
CHAPTER 5

PARALLEL APPROACH

How can an algorithm be mapped to a parallel implementation that will yield an optimum speedup? Certainly we can assign additional processors to sections of the code after arbitrarily partitioning it, but this may result in little change in throughput times, and may make thing worse due to communication costs between processors. What we intend to show in this chapter is how to efficiently apply processors to the covering method in order to achieve the optimum speedup. Ultimately, what would be desirable is a method that can allow for scalability, in both the data size and the architecture, or the capability of an increase in performance proportional to the increase in size. For the scalability of the data size, we would like to be able to increase the size of the image and have the architecture be able to process it. For the architecture upon which it is applied, we would like to achieve a near linear speedup for each processor that is added.

Analysis of the covering method indicates, as expected, that the algorithm is inherently parallelizable with low computational complexity. The equations that govern the calculation of the covering method are regular and applied in a local fashion to each pixel position in the image. It has been
successfully applied to several computing platforms, indicating the potential for real time speedup [HAYES 93].

Algorithms can be evaluated by a variety of criteria. In this case, we are interested in the rate of growth of the time needed to solve the problem. The time required by an algorithm expressed as a function of size of a problem is called the *time complexity* [AHO 74].

It is important to examine the complexity involved with the task of computing the features rather than any machine dependent factors, such as communication costs, which will vary between computer architecture types. The time complexity of this algorithm can then be used to determine which type of parallelization is most suitable in terms of performance, such as: massively parallel SIMD (Single Instruction Multiple Data) or distributed processing MIMD (Multiple Instruction Multiple Data).

SIMD and MIMD architectures are two distinct types which are of interest to us. The results of studying algorithms for operation in SIMD has shown that they generally can be applied to array computers, pipeline computers, and any other particular type of SIMD computer [STONE 73]. Algorithms for MIMD may be applied to different implementations such as a hypercube machine, such as Intel’s Hypercube and Paragon computers, or to a distributed environment of workstations.
5.1 Complexity Analysis

An analysis of the covering method algorithm can be made by studying the time complexity of the two major parts: the computations of the Upper/Lower surfaces and that of the Area estimate for each pixel. Assuming that the grayscale image is n pixels wide by n pixels long, we can calculate the order of time complexity for computing an n-by-n image.

Both computations for Upper and Lower surfaces involve the selection of a value by comparison from a pixel's nearest neighbors, those that are North, South, East, and West of the pixel. Thus, four comparisons are conducted for each pixel in the image. The cost of processing each surface is

\[ f(n) = 4c_s n^2, \]  

(EQ 7)

where \( c_s \) represents a constant number of instructions and the order of magnitude for each surface is \( \mathcal{O}(n^2) \).

The computation of the Area for each pixel is by far the most intensive and time consuming procedure requiring over 90 per cent of the execution time to complete. The value for each pixel is determined by the differences of the Upper and Lower surfaces of each pixel within a window around the target pixel as defined by equation 6. The cost of computing the Area estimate is

\[ f(n) = c_a w^2 n^2, \]  

(EQ 8)

(with \( c_a \) being the constant number of instructions and \( w \) being the window size) which also has an order of magnitude of \( \mathcal{O}(n^2) \).
The window size is directly related to the scale factor, \( e \). This scale factor, \( e \), also determines the number of Upper/Lower surfaces computed, as well the Area values per pixel. Good results in discrimination between textures can be attained for scale values of \( e = 5 \).

It should be noted that even though the computation of the surfaces and areas are of the same order of magnitude, the time to compute each area estimate is much greater than the surfaces. The surface values can be represented by integers where as the area estimates are of real numbers. Also, due to the summation of values within a window, the resulting computation for the surface area is a magnitude of \( w^2 \) greater than that of the surfaces.

In examining the covering method for implementation on multiprocessing platforms, the problem size as measured in the number of operations performed will also be important to the determination of which architectural platform best suits this method.

### 5.2 Problem Size

Along with the time complexity for the covering method algorithm, it can also be beneficial to study the actual number of operations required. These operations are broken down into the numbers of comparisons, additions, subtractions, multiplications, and divisions in computing this method. Based upon the complexity analysis, and using a window size, \( w = 9 \), for each pixel in an image the following numbers of computations are required:
Operations per pixel position for each scale $\varepsilon$

<table>
<thead>
<tr>
<th>Operations</th>
<th>Sarkar &amp; Chaudhuri</th>
<th>Covering Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparison</td>
<td>464,142</td>
<td>655,360</td>
</tr>
<tr>
<td>Addition</td>
<td>37,524</td>
<td>6,717,440</td>
</tr>
<tr>
<td>Subtraction</td>
<td>18,762</td>
<td>6,717,440</td>
</tr>
<tr>
<td>Multiplication</td>
<td>28,143</td>
<td>81,920</td>
</tr>
<tr>
<td>Division</td>
<td>28,143</td>
<td>81,920</td>
</tr>
<tr>
<td>Total</td>
<td>576,714</td>
<td>14,254,080</td>
</tr>
</tbody>
</table>

Comparing this method against that proposed by Sarkar and Chaudhuri should give an indication of the computational complexity with a more efficient mode. We will use an image size of 128x128, with a scale of 5 (all that is required for our method as opposed to 15 for Sarkar and Chaudhuri) [SARK 92]. Table 2 list the values for the two methods.

From the information displayed above concerning the numbers of computations for the covering method, the complexity of this approach yields almost 25 times more operations as does the method by Sarkar and Chaudhuri for an image of size 128x128 pixels. It is also apparent why the sequential calculation of this method resulted in such long throughput times.
The value in the use of the covering method comes from its measurement of the fractal dimension on a local level. Thus heterogeneous texture regions can be evaluated within an image. However, the result is a more computationally complex method. As the size of the image grows, the computation complexity grows at the rate of \( n^2 \) and this behavior limits the efficient calculation for a single processor machine for useful image sizes. This being the case, what is the effect if we use more than one processor to compute the algorithm?

5.3 Speedup Potential

As a model, the eye itself is composed of millions of photo-receptor cells which perform some processing before sending on a signal which reaches the brain. If we extend this model to our low-level processing method, how would the complexity change to improve output? We can use Amdahl's law for parallel processing (equation 9) to evaluate the change in complexity. We are given some number of processors, \( p \), to solve the problem. A given sequence of code may be divided between the portion, \( f \), which can only be solved sequentially, and the portion \((1-f)\) which is parallelizable. The rate of speedup, \( S \), is

\[
S \leq \frac{1}{f + \frac{(1-f)}{p}} \quad \text{(EQ 9)}
\]

The ultimate purpose of speeding up the process is to achieve real time completion of the vision task. In other words, from start of execution to pro-
cess completion, defined as wall clock time, the task should be completed in the subsecond range. Based on equation 9, this portends the use of more processors, and to improve performance, reducing the proportion of sequential instructions. As the image size increases, the proportion of parallelizable code verses that which is sequential increases. This algorithmic problem is comparable to "image smoothing" as described by Siegel, et al. [SIEGEL 92], for which the theoretical maximum speedup is equal to the number of processors, \( p \). In other words, theoretically, doubling the number of processors results in reducing the processing time by half. The speedup would be a factor of two.

5.4 Data Parallelism

Because the covering method involves performing the same series of operations on every pixel in the image, it may be labeled a data parallel algorithm which is described by Hillis and Steele as an algorithm which derives its parallelism by the implementation of simultaneous operations across the data set [HILLIS 86]. Hockney and Jesshope describe the expression of parallelism in two forms: structure parallelism and process parallelism [HOCK 88]. Both data parallelism and structure parallelism can be associated with processes that occur in natural vision. In vision processing, the data is the grayscale image from which we wish to extract the fractal dimension of each pixel.
Using this paradigm, every processor would execute each step of the covering method simultaneously. As an example, to compute the difference, or the Potential (P), between Upper and Lower surfaces (U and L) requires only the following syntax;

\[ P = U - L \quad \text{(EQ 10)} \]

However, this model would require an architecture with 250k processors in order to compute a 500 by 500 pixel image.

Assume that we wish to compute the above instruction on the parallel data structures U and L. We have insufficient numbers of processors to map to each data element. Therefore, we must make use of the concept of virtual processors, where the resident processors are multiplexed as necessary to support the abstract processor requirements [HILLIS 86]. The benefits of the virtual processor abstraction are (1) that the same program can run unchanged on different quantities of processors in the machine, and (2) in terms of programming, the number of processors can be considered as expandable. This allows the programmer to develop code that does not have to deal with the quantities of processors available. Indeed, for the Connection Machine, the programming language allows the user to allocate virtual processors dynamically [CM200].

Using the concept of virtual processors, one can consider the data distributed such that each pixel is operated on by its own processor but that the hardware device would multiplex the loading of available processors to the data until all data had been actively processed for the required instruction
[QUINN 87]. Data parallelism can be implemented on both SIMD and MIMD architectures [BLANK 92]. This paradigm works best on large problems, which does apply to this method, due to the uniformity of the operations applied.

Figure 11 and 12 shows some of the different implementations for image processing. The image processing can be divided among processors by process, or by dividing the image into subimages upon which a processor would be applied, or by use of data parallelism. Data parallelism effectively breaks up an algorithm/task into concurrent subtasks to a degree that their exist one virtual processor for each pixel position in the image. Siegel, et al., identified that subtask parallelism in image processing can have an impact on performance [SIEGEL 92]. This paradigm, coupled with effectively applying as many processors to the problem as possible (remembering that this algorithm approaches near linear speedup with an increase in numbers of processors) should give the necessary methodology for achieving real time speeds.
FIGURE 11. Methods for Parallel Processing of an Image

By Process

Grayscale Image

\( P \) - processors

\( T \) - tasks

By Subimages
FIGURE 12. Image Processing Using Data Parallelism

Data Parallelism

Grayscale Image

Simultaneous Operations on Each Pixel

T₀

T₁

T₂
5.5 Architecture Mapping

Parallel processing architectures can be divided into two broad categories: Single Instruction Multiple Data (SIMD), and Multiple Instruction Multiple Data (MIMD). The differences in these two architectural types is in terms of control. Each processor in an MIMD system executes its own separate set of instructions. In the SIMD system, a single instruction is executed by all processors in the system.

- SIMD

The basic model of a shared-memory class of SIMD, the Parallel Random Access Machine (or PRAM), allows all processors simultaneous access to memory as long as no two processors attempt to read the same location at the same time [AKL 89]. Since simultaneous reads from identical memory locations cannot be ruled out with this algorithm, a more useful model is the Concurrent Read Exclusive Write (CREW) PRAM model. The processing of information for each pixel produces a unique value such that a processor would write to an exclusive location in memory. Therefore, there is no write conflict problem. While other PRAM models may be closer to both real machines and natural analogues of vision, the CREW is the best model for demonstrating the parallelism capability of the covering method.

Though the elements of the covering method are themselves simple algorithms involving comparisons and summations, the cost of parallelizing these elements will involve some communication traffic control to limit simultaneous reads and, more importantly, the loading of information into the
shared-memory resource. This means that there should be some loss in speedup due to these operations as the problem grows.

Ideally, for a large processing array with \( n^2 \) elements, the computations of the area estimate and surfaces would take place in some constant number of instructions as the number of pixels and the number of processors converge.

- **MIMD**

Quinn divides MIMD algorithms into three categories: Pipelined, Partitioned, and Relaxed. Using the systems available, a partitioning approach is feasible if we preschedule those sections of the covering method which can be implemented in parallel [QUINN 87].

Certainly the simplest method for partitioning the algorithm is at the computation of the surface area arrays. This is because each surface area array is independent of the other. However, this still requires a processor to perform \( O(n^2) \) calculations. Obviously, with faster processors a good speedup can be achieved with this method over the original workstation model. However, it would not efficiently allocate the workload equally to all processors if there is an odd matchup between processor nodes and surface area arrays to be computed. Limitations would occur for both load balancing and partitioning methods. This would result in additional computational overhead.

Another more practical scheduling scheme is to allow a series of processors to be scheduled to compute each summation over each pixel. In
effect, this would be a manual implementation of data parallelism. As with
the SIMD model, as the number of processors available approaches \( n^2 \), the
problem approaches linearity with the problem size. Of more importance for
this method is how the values are passed for such operations as area summa-
tions over a window. If some form of shared memory is used, then this sys-
tem would behave in accordance with the SIMD model. If distributed
memory is used, some procedure must be implemented to output the entire
surface area arrays upon completion of the computation. If this procedure
involves asynchronous communication between processors, this imposes fur-
ther complexity and overhead, especially if is not transparent to the program-
mer/user.

5.6 Supposition on Implementation for Real Time Processing

The covering method represents a subset of a large class of low-level
image processing operators. Research by authors, such as Dew and Manning
[DEW 86], and Siegel, et al.,[SIEGEL 92] resulted in analysis of implementing
such operators on various parallel machines. From the results, it can be sur-
mised that if these operators can be decomposed into small enough subtasks,
with a processor assigned to each, it is possible that the computations could
approach the optimum speedup, linear with the number of processors. The
data parallelism paradigm represents the best fit for reducing this class of
problems into the smallest decomposable subtask for efficient execution in a
parallel environment.

The next chapter will show the results of implementing this paradigm
on across several architectures.
As previously mentioned, the covering method has been successfully demonstrated on a high-end graphical workstation [SOLKA 92]. In order to properly model the original algorithm on several diverse architectures, the original program was modified to extract out extraneous I/O routines and library calls which would impact the extraction of performance data.

The language of implementation was C, available on all architectures, with the Connection Machine having its own version called C*. The multiprocessing platform, a Silicon Graphics (SGI) 4D/440, used the C optimizer, Power C. This was an early version which had some drawbacks which will be discussed later.

6.1 Program Process

The flowchart in Figure 13 diagrams the original program process for calculating the covering method. This initial scheme was designed to produce both a data set for later processing (i.e. linear regression), and to produce graphical images like those in Figures 5 and 6.
FIGURE 13. Original Program Process for the Covering Method
Note that the Erosion/Dilation operators are initially applied to the image. Successive iterations involve using the previously computed Erosion/Dilation surfaces as inputs for these operators.

For the measurement of wall clock time, the loading of the image and off-loading of the results of the covering method are excluded. This is done for two reasons. First, different computers will have varying capabilities in disk access and it is not our intent to measure how fast the disk drive works. Second, if this method were to be implemented for real time applications, the output would be processed by the next set of operations rather than store the results off-line. It is reasonable to expect that the output would be held somewhere in memory with its much faster access time than disk storage devices. Therefore, the measurement area consists of the looping operations of computing the Erosion and Dilation surfaces, the computation of the Surface area, and the computation of the minimum and maximum Surface area values, only, as identified in Figure 13. The minimum/maximum computations are highlighted in the figure because it was later determined that these computations added considerably to the processing time.

6.2 Performance Measurement

Performance was measured as a function of wall clock time. This would include both user and system processing time. The output displayed in Figure 14 gives the statistical means for program completion in seconds of execution time. Sufficient numbers of runs (25 per image size) were made to give a good statistical sample (i.e., for the Cray Y-MP2E and n=500 the average processing time was 27.69635 seconds, and the variance was 0.025258).
FIGURE 14. Processing Time versus Image Size using the Covering Method
The algorithm was modelled against varying image sizes. The scale of sizes varied from 50 to 500 pixels in width and length. The original program used as a standard 480 by 512 pixel images.

6.3 Architecture Sampling

The program was modeled on several computer architectures. Initially implemented on a graphical workstation, the performance generated from this machine was used as the standard for all other devices. Other architectures used were a course-grained, tightly-coupled, multi-processor workstation (4 processors), a vectorizing supercomputer, and a massively parallel supercomputer.

Workstations

Both the single and multi-processor workstations were made by Silicon Graphics (Personal Iris 4D/35 and Power Series 4D/440). The single processor ran code compiled from the standard C compiler. The program code for the multi-processor workstation (MIMD) was preprocessed through a multiprocessor optimizer which inserted compiler directives at critical paths to create macrotasking of inner loops in iterative statements. Thus, each processor would be tasked with a separate thread containing that iterative step. The operating system was responsible for loading the individual processors as each became available to run the next iteration. The executable code produced by this older version of the multiprocessing compiler, incurred significant overhead on start-up. Table 3 shows that it was an order of magnitude slower than a single processor model for the smallest image size. It did not
overcome this limitation until n was equal to or greater than 300 for the image size.

The SGI 4D/440 is a shared-memory coarse-grained MIMD machine. It contains 4 R3000 MIPS processors, the same as in the SGI 4D/35 workstation. Through software, the user can control the number of processors accessible to the application.

**Vector Supercomputer**

The Cray Y-MP2E used for this model is a single processor vectorizing supercomputer. Vector optimization was used with limited success for this code. It had been noted that the C vector-optimization was less capable than that of the FORTRAN model. The performance displayed in Figure 14 demonstrates the significant speed up over both workstation models. However, in terms of cost effectiveness, the workstations gave a better accounting of performance to system cost.

**Massively Parallel Supercomputer**

The CM200 is the latest implementation of Thinking Machines, Inc. SIMD architecture. It represented the best chance to demonstrate the potential speed up achieved by representing the algorithm as a *data parallel algorithm*. Parallel array variables are spread across up to 64k processors with each element stored in each processor’s memory. The hardware allows each physical processor to behave as many virtual processors and is invoked automatically when parallel variables are declared [CM200]. This virtual processor implementation is relatively transparent. The C* language uses the
concept of data parallelization in the format of parallel variables which are
used to store and manipulate arrays of data.

6.4 Comparison of Different Computer Architectures

Figure 14 is a plot of throughput times for each of the architectures
based upon image size. The performance can be determined by examination
of the apparent slopes for each type of device used.

Notice that the performance of the single processor workstation appar­
ently follows an $n^2$ path as the image size increases. Application of four pro­
cessors for the same family of workstations yields a slower growth. The slope
for the four processor workstation and that of the vectorizing supercomputer
are similar, from which it could be implied that for some problems the more
cost efficient computing device is the multiprocessing workstation.

The most interesting result is that for the Connection Machine. Test
case design for the Connection Machine was dictated by the constraints of its
programming language. It required that the number of elements in the vari­
able be powers of two and be equal to or exceed the number of processors
available. As a result a stepping pattern occurred. For 8k processors, the
smallest parallel variable possible was 128x128 cells for a symmetric array.
Image sizes of 50 and 100 were applied to variables of size 128x128, sizes of
150, 200, and 250 to variables of size 256x256, and the remaining image
sizes were applied to variables of 512x512. The steps occur at these transi­
tions.
Flynn first described this pattern, calling it the "fitting problem" which is due to the incompatibility of source vector size and numbers of available processors [FLYNN 72]. Denning and Tichy delineate these discontinuities as the result of the simulation of virtual processors mapped on to the real processors [DEN 90]. These discontinuities are visible around the points of 128 and 256. The differences can be viewed as the cost of applying the problem to a larger number of virtual processors multiplexed to the actual processors.

Additionally, the SIMD performance does not appear to follow the increasing growth of throughput times of the other architectures. The difference between steps may be accounted for in the quadruple growth of the parallel variable each time it is increased. A profile of the execution of the program on the Connection Machine indicated that over seventy percent of processing time was used to write the results stored in the surface area parallel variable to an associated array on the front end machine.
<table>
<thead>
<tr>
<th>Image Size (NxN)</th>
<th>CPUs</th>
<th>SGI 4D/35</th>
<th>SGI 4D/440</th>
<th>Cray Y-MP</th>
<th>TMC CM-200</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>1</td>
<td>0.3700</td>
<td>4.181200</td>
<td>0.193593</td>
<td>0.3215</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
<td>2.0248</td>
<td>8.190000</td>
<td>0.952274</td>
<td>0.3415</td>
</tr>
<tr>
<td>150</td>
<td>1</td>
<td>5.1860</td>
<td>12.78960</td>
<td>2.517519</td>
<td>1.0770</td>
</tr>
<tr>
<td>200</td>
<td>4</td>
<td>8.8279</td>
<td>17.51920</td>
<td>4.771344</td>
<td>1.1030</td>
</tr>
<tr>
<td>250</td>
<td>1</td>
<td>13.8428</td>
<td>22.33400</td>
<td>7.591169</td>
<td>1.1334</td>
</tr>
<tr>
<td>300</td>
<td>4</td>
<td>19.9363</td>
<td>27.21440</td>
<td>9.831461</td>
<td>3.7948</td>
</tr>
<tr>
<td>350</td>
<td>1</td>
<td>39.7668</td>
<td>32.22240</td>
<td>13.36696</td>
<td>3.8436</td>
</tr>
<tr>
<td>400</td>
<td>4</td>
<td>55.0179</td>
<td>37.34880</td>
<td>17.57627</td>
<td>3.9120</td>
</tr>
<tr>
<td>450</td>
<td>1</td>
<td>107.9367</td>
<td>42.53440</td>
<td>22.35932</td>
<td>3.9656</td>
</tr>
<tr>
<td>500</td>
<td>4</td>
<td>130.3232</td>
<td>47.84800</td>
<td>27.69635</td>
<td>3.4616</td>
</tr>
</tbody>
</table>

**TABLE 3. Comparative Run Results for Fractal Dimension Computation (in seconds).**
6.5 Results for Varying Numbers of Processors on MIMD

As mentioned previously, the operation of code on the SGI 4D/440 workstation showed significant overhead induced by the old version of the compiler. Regardless, observation of the results varying the number of processors indicated interesting results.

Notice the plot of results for figure 15 based on the data provided in table 4. Results were obtained by running the code on the multiprocessing platform withholding various numbers of processors from participating. The additional overhead produced by the multiprocessing compiler severally limits the execution of the code in comparison to the single processor performance of the SGI 4D/35. However, the implementation does overcome these limitations as the number of processors are increased.

It is expected that as the C compiler becomes a more mature product, that this overhead will be significantly reduced. The performance, despite the overhead, demonstrates that multiprocessing workstations and computer servers are showing increased capability to compete with the mainframe/supercomputer platforms, especially on a cost/performance ratio.

Though the times at 500x500 pixel size were above the real time speeds needed for even a desktop application (table 1), the performance is significant. As additional processors are added, the throughput times decrease proportionately. The following supposition could be made from evaluation of figures 14 and 15; the key to improved performance for this type of algorithm is not in improving the computational power of the processor (as in the Cray
Y-MP2E or the single processor workstation with its MIPS R3000), but in the number of processors applied to the computation.
FIGURE 15. Comparison Runs: Single versus Multiple Processors  
SGI 4D/35 (1 cpu) vs. 4D/440 (4 cpu)
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0.3700</td>
<td>16.58280</td>
<td>8.04800</td>
<td>5.59160</td>
<td>4.18120</td>
</tr>
<tr>
<td>100</td>
<td>2.0248</td>
<td>32.76520</td>
<td>16.55600</td>
<td>11.14960</td>
<td>8.19000</td>
</tr>
<tr>
<td>150</td>
<td>5.1860</td>
<td>51.25720</td>
<td>25.90399</td>
<td>17.42960</td>
<td>12.78960</td>
</tr>
<tr>
<td>200</td>
<td>8.8279</td>
<td>69.87919</td>
<td>35.40559</td>
<td>23.88000</td>
<td>17.51920</td>
</tr>
<tr>
<td>250</td>
<td>13.8428</td>
<td>89.01158</td>
<td>45.06120</td>
<td>30.54360</td>
<td>22.33400</td>
</tr>
<tr>
<td>300</td>
<td>19.9363</td>
<td>108.31040</td>
<td>54.91359</td>
<td>37.11280</td>
<td>27.21440</td>
</tr>
<tr>
<td>350</td>
<td>39.7668</td>
<td>128.18318</td>
<td>64.81120</td>
<td>43.82840</td>
<td>32.22240</td>
</tr>
<tr>
<td>400</td>
<td>55.0179</td>
<td>148.11560</td>
<td>75.01480</td>
<td>50.62999</td>
<td>37.34880</td>
</tr>
<tr>
<td>450</td>
<td>107.9367</td>
<td>168.78559</td>
<td>85.27999</td>
<td>57.69721</td>
<td>42.53440</td>
</tr>
<tr>
<td>500</td>
<td>130.3232</td>
<td>189.31718</td>
<td>95.70759</td>
<td>64.68160</td>
<td>47.84800</td>
</tr>
</tbody>
</table>

**TABLE 4. Comparative Run Results for Fractal Dimension**  
Single vs. Multiple Processors  
(SGI 4D/35 vs. SGI 4D/440 running 1, 2, 3, and 4 processors)
6.6 Results for Varying Number of Processors on the Connection Machine

The CM200 used in this research had 16k processors. Initial performance runs were conducted on just half of the machine's processors, and the results in the form of surface area arrays were downloaded to the front end processor, which was a workstation.

A second phase of investigation was warranted to explore the potential speedup from 8k to 16k processors. As mentioned earlier, there was a severe cost extracted when passing the data back out to the front end workstation. During this phase, that operation was eliminated to better explore the potential for performance improvement without drowning out the results due to the conversion of parallel variables into array variables at the front end.

Figure 16 depicts the various processing times for different image sizes for both the 8k and 16k processor runs. Notice that without the requirement for sending data immediately to the front end, that all runs were accomplished in under one second!

As far as I can discern, no other research has been attempted to benchmark fractal dimension code on highly parallel computing devices. However, research on computer vision algorithms by James Little at MIT was performed on the first version of the Connection Machine. A series of algorithms were tested for benchmark performances with the following parameters:

(1) images were 8-bit digital of size 512x512 pixels,

(2) system configuration used 64k processors.
The surface area computation of the covering method is a type of convolution. We can compare it to Little's "Convolution with Laplacian" which used a mask diameter of 11 pixels. The dilation/erosion operators can be compared to the "Zero-Crossing" calculation which examined the sign bits of each pixel's nearest neighbor [LITTLE 86]. The following time were reported for the implementation of these algorithms:

- Convolution 3.0 ms
- Zero-Crossing 0.5 ms

We can use these results to extrapolate a comparison value for the covering method runs of 500x500 pixel image sizes. Each convolution would need be run five times and each Zero-Crossing run ten times (five times each for the dilation and erosion operators).

Times total (For \( \varepsilon = 5 \)):

- Convolution 15 ms
- Zero-Crossing 5 ms
- Total 20 ms

Since these were calculated on a 64k machine with 4 virtual processors for every real processor (4:1), we must normalize the values above to reflect the performance on an 8k (32:1) and 16k (16:1) Connection Machine. Extrapolating, we get 0.16 and 0.08 seconds processing time, respectively. From table 5 the values calculated on the CM200 for these parameters are 0.781389 and 0.625638 seconds.
The actual covering method times certainly approach Little’s performance benchmarks. Further investigation in improving performance of the covering method is warranted, especially if used in texture analysis. However, the values reported herein can be considered “good enough” to demonstrate that sub-second times are actually achievable. To achieve processing this algorithm at Little’s benchmark speeds would make possible such applications as visual processing for the smart car sensor system eluded to earlier in table 1.
FIGURE 16. Connection Machine Time Results: 8k vs. 16k Processors
<table>
<thead>
<tr>
<th>Image Size (NxN)</th>
<th>CM200</th>
<th>CM200</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUs</td>
<td>8k</td>
<td>16k</td>
</tr>
<tr>
<td>150</td>
<td>0.154555</td>
<td>0.098952</td>
</tr>
<tr>
<td>200</td>
<td>0.180047</td>
<td>0.138920</td>
</tr>
<tr>
<td>250</td>
<td>0.215399</td>
<td>0.167252</td>
</tr>
<tr>
<td>300</td>
<td>0.514292</td>
<td>0.344134</td>
</tr>
<tr>
<td>350</td>
<td>0.569407</td>
<td>0.415731</td>
</tr>
<tr>
<td>400</td>
<td>0.657654</td>
<td>0.512917</td>
</tr>
<tr>
<td>450</td>
<td>0.706991</td>
<td>0.532282</td>
</tr>
<tr>
<td>500</td>
<td>0.781389</td>
<td>0.625638</td>
</tr>
</tbody>
</table>

**TABLE 5. Comparative Run Results for Connection Machine 8k vs. 16k processors**
6.7 Speedup Revisited

The Connection Machine results provide us with a vehicle to probe the potential for speedup with additional processors. The upper-bounds on the parallel processing of covering method-like algorithms is linear with the number of processors. How close, then, is the method, as presently implemented, in reaching this goal.

Figure 17 is a plot of speedup between 8k and 16k processors. The differences in variable sizes, 256x256 and 512x512, result in the discontinuity between the two lines. As the image size grows, the speedup decreases for both lines. If we are using true data parallelism throughout, how is it possible that image sizes less than the parallel variables impact the processing times.

The code for the Connection Machine model is presented in Appendix B.2. The bulk of the computation is performed in the function entitled "Compute_Area". This function is taken from the original code, where a pointer to the surface area array was passed to this function which filled in the array as part of its execution. Additionally, the function determined the maximum and minimum values for each array, which was necessary for providing a visual output.

At best, the maximum and minimum values can be computed on the Connection Machine in $O(\log n^2)$ [HILLIS 86]. The operations on non-parallel variable array certainly adds to the cost of computation, and as these variables increase in size, that cost grows.
Further modification of the original code can be implemented to produce more efficient execution. There is considerable saving to be made if the code is optimized. Such code optimized for the Connection Machine with the full 64k processors should approach Little’s benchmark of 20 milliseconds.

With the current code, the best speedup is attained when both image size and parallel variables are small, and the virtual processor to actual processor is small. At 150x150 image size, the speedup is 1.56, drawing close to the theoretical potential of 2.

The conclusion of the results is that with the proper methodology for implementing the covering method (i.e. data parallelism), and an architecture that allows for large numbers of processors, it is realistic to expect that this method can be implemented in subsecond times. Though broad based, this study was not exhaustive in terms of parallel computers. A comparison of other architectures, and numbers of processors should validate this approach.
FIGURE 17. Speedup Comparison between 8k and 16k processors
By demonstrating the capability to compute the fractal dimension in real time, thereby providing a means for extracting texture features from an image, we now have a method for addressing the needs of texture analysis computer vision. The next step lies in producing a model based upon this method.

However, there are still some issues that need to be addressed in the development of such a model. Specifically, how can boundary information be included in this model, and at what point? What are the other computation problems in developing a real time model? How can a large spatial array of pixels be transformed into compact forms of information usable by higher vision levels? What aspects of ATR can be addressed by differences in this model?

7.1 Boundary Information

The presence of a boundary between textures in this neighborhood can result in a greatly perturbed set of texture features, especially if there is a large contrast between the two textures. Segmentation schemes are designed with the goal of detecting boundaries between texture types. This raises the
question of how to incorporate any segmentation information that may be available into the texture feature computation so as to avoid basing features on both textures (as well as a contrast difference) near a boundary.

Rogers, et al., have developed a methodology involving only nearest neighbor computations to include boundary information in a modified covering method [ROGERS 93]. Peli states that a preponderance of the pixels within the surface area window should contain mostly the target texture [PELI 90]. Other methods are currently being discussed by the researchers at the Naval Surface Warfare Center's Advanced Computation Technology Group to solve this problem.

An alternative method to the one proposed by Rogers, et al., would limit texture contamination across boundaries by modifying both the dilation/erosion operators and the surface area computation by use of a boundary map. This map would be produced, initially, by some segmentation scheme.

7.2 Pre-attentive Vision

Pre-attentive vision can be considered as a wide area scan, where the viewer is interested in taking in the whole picture without concentrating on any particular part or area. Figure 18 describes a model for pre-attentive vision based upon texture analysis using boundary information.

The information from both the texture analysis feature extractor and boundary map may be further processed by a discriminant analysis/pattern recognition system, such as the one described by Solka, Priebe, and Rogers [SOLKA 93]. The resulting probabilistic classifiers can be grouped to identify
gross areas of similar textures. This information is then channeled to higher level vision functions. The classifiers could be maintained in a data base for future use, providing a more compact means of storing image information.

There are two regions of computer processing in figure 18. The regions involved with segmentation/boundary map creation, texture analysis, and linear regression should be quickly processed by some form of massively parallel computational device. Based upon Little’s benchmarks [LITTLE 86], segmentation and texture analysis should be achievable in subsecond times using a shared-memory SIMD processor like the Connection Machine. Future research will demonstrate that the entire texture analysis feature extraction process, including the linear regression operations, can be processed within real time speeds.

Initial evaluation of the discriminant analysis has shown good processing times with a single processor. A possible MIMD approach should be evaluated if this approach is to used in real time vision processing.
Segmentation

Boundary Map

Texture Analysis (Covering Method)

Linear Regression

Automatic Discriminant Analysis

* Texture Regions
  * Grouping of Similar Regions

High-Level Vision Functions

FIGURE 18. Pre-attentive Textural Vision Model
7.3 Attentive Vision

Once a general mapping is made in the vision system of the “world”, attentive vision techniques may be employed. Rough textural groupings can be refined. Specific high interest areas can be re-evaluated to provide additional discrimination. Motion may be detected against a background. A complex object may be redefined by a composite of textures that reside within a closed boundary, such as a house which has many textures (i.e. roof, windows, boards or brick surfaces of the exterior, etc.)

Figure 19 describes a model for attentive vision. It allows for “active perception” as described by Wechsler [WECHSLER 90]. Active perception allows the system to explore its surroundings. Additionally, by focusing attention on selected areas, the computational complexity may be quite high, but the data size is reduced from that of the whole image, thereby relieving the system from having to process the entire image.

What determines the area of focus may depend upon the intended use of the vision system. In medical imagery, an attentive vision system may roughly scan an X-ray image for signs of cancer. If an area produces a texture on the first pass that roughly falls within that classification, further analysis may be directed at that region to improve the identification of it, and to better localize the area of growth.

This type of process would require the upper-level vision system to modify the boundary map to nullify all but the interest area. Boundary map information currently used by Rogers, et al., consists of a data array of the
same dimensions as the image with values of 1’s and 0’s [ROGERS 93]. The 1’s represent the boundary area. Given a closed bounded area much smaller that the original image, a procedure can be developed which would extract this smaller array for processing with expectations of a faster throughput than the entire image.

The additional advantage of this model is the ability to recursively update the present image model. For a dynamic vision process, a determination would need to be made for starting the whole process over from the beginning, such as when the field of view of the vision system shifts due to movement.

This discussion was meant to describe possible implementations of texture analysis within a vision system for real time applications. Further research is warranted and planned by this author in the foreseeable future to produce a working model based upon this approach.
Segmentation

Texture Analysis (Covering Method)

Linear Regression

Automatic Discriminant Analysis

* Complex Texture Regions
* New Object Identification
* Refined Discrimination

High-Level Vision Functions

FIGURE 19. Attentive Vision Model
CHAPTER 8

CONCLUSION

An effective paradigm for implementing the processing of fractal dimension in parallel has been presented. It has been demonstrated that the capability to compute the fractal dimension of grayscale images within the realm of speeds required for real time computer vision is possible. Additionally, it has been shown that using the covering method to extract fractal dimension data from a grayscale image is computationally intensive. As the size of the image grows the computation complexity grows at the rate of $n^2$. This behavior limits the efficient calculation for a single processor machine for useful image sizes. From our results, an image of 500x500 pixels required over 130 seconds to process this feature extraction method on a single processor workstation. A real time application would require a two order of magnitude reduction in processing time to be even marginally effective.

Significant speedup was attained by applying the concept of data parallelism to this algorithm. Processing times within the range necessary for real time applications were achieved using this method on the Connection Machine. Thus, certain aspects of low-level image processing, such as texture analysis, can be addressed as data parallel algorithms and thus allow the exploitation of spatial parallelism by such massively parallel machines.
These methods were compared against previous execution times based on vision research on the Connection Machine and found to conform with the expected processing times.

With the advent of parallel processing and robust texture analysis through the use of fractal dimension, a model for employing such a texture analyzer was presented. Research in the use of texture in ATR has shown great promise. The ability to compute texture features in real time paves the way for serious applications.

We surveyed different architectures for testing this method. Preliminary results suggest that for real time speeds the massively-parallel SIMD devices, such as the Connection Machine, provide the best architecture for rapid computation of fractal dimension using the covering method.
Bibliography


Appendix A

The following definitions will be used in our discussion of applying the Covering method to parallel architectures.

Definitions

Fractal - objects with structures which cannot be defined by Euclidean geometry, self-similar invariant of scale, and of recursive origin.

Granularity - The unit of computational work allocated to a processor is called a grain. Course grain systems contain many data elements while fine grain systems may contain only a few, or even one, data element.

Memory - Is categorized in parallel systems as either Shared or Distributed. In a shared memory computer, each processor has access to all the computational memory; in a distributed memory computer, each has access only to its local memory and must exchange messages with other processors to obtain nonlocal data.

MIMD - Multiple Instruction Multiple Data. Multiple instruction streams imply the existence of several instruction processing units and therefore necessarily several data streams.

PDF - Probability Density Function. A non-negative function which is piecewise continuous and integrates to one.
**SIMD** - Single Instruction Multiple Data. This a computer that retains a single stream of instructions that initiate many operations. Each element of the vector is regarded as a member of a separate data stream, hence there are multiple data streams.

**Scalable** - capable of being increased in size, or, more exact, capable of an increase in performance proportional to an increase in size.

**Speedup** - the ration of two programs execution times. Speedup is usually discussed as a function of the number of processors used, but it can also be a function of problem size.

**Texture** - defined as the non-uniform spatial distribution of intensities in grayscale images.

**Wall Clock time** - the time required to actually complete a process or program.
Appendix B
FRACTAL COMPUTATION CODE

B.1 Single Processor Code for SGI 4D/35

/*****************************************************************
* Program: Fractal_area
* by: Hal Hayes
* purpose: Computes the fractal dimension (delta) at each
* pixel for a NxN image for several choices of the
* scale parameter (epsilon).
* date: November 10, 1992
* version: 3.3
*
* files (in): image file ( )
* N_pixel_pos.c <include>
* N_readfractal.c <include>
* N_writeout.c <include>
* fractal.h <include>
* files (out): image file enhanced by covering method for
* features, name as specified by user when asked.
*******************************************************************/

#include "fractal.h"

/*
 * EXTERNAL FUNCTION CALLS
 */

extern void pixel_pos (); /* performs covering method */
/*** from N_pixel_pos.c ***/
extern void read_input_2 (); /* reads in image as binary*/
/*** from N_readfractal.c  ***/

extern void A_out (); /* output data in ascii  */
/*** from N_writeout.c  ***/

/*@ 
* DATA STRUCTURES 
*/

/*@ 
* Area_data structure used to hold results in a lined 
* list.
*/

struct Area_data {
    float d_max,
    d_min;
    float scale_f;
    int area_num;
    struct Area_data *next_ptr;
};

/*@ 
* first_ptr points to head of linked list of results.
* new_ptr points to a newly created node.
*/

struct Area_data *first_ptr = NULL;
struct Area_data *new_ptr;

/*@ 
* declare time variables for accounting 
*/

struct tms tim;
float stime, eltime;

/*@ 
* usurface holds the upper surface values for A_size layers.
* lsurface holds the lower surface values for A_size layers.
*/
static int usurface[A_size][PicRow_Size][PicCol_Size],
    lsurface[A_size][PicRow_Size][PicCol_Size];

/*
 * area holds the real values of area computed from
 * the upper and lower surfaces for up to A_size iterations.
 */

static float area[A_size][PicRow_Size][PicCol_Size];

;/* delta_max - maximum area value */
float delta_max = -100000.0,
/* delta_min - minimum area value */
delta_min = 100000.0;

FILE *datafile, *fractalfile, *result;
float Pic_Scale = 255;

/****************************************************************************
* Main Program Section
****************************************************************************/

main(argc,argv)
    int argc;
    char *argv[];
{

/****************************************************************************
* Variable Section
****************************************************************************/

    float scale_factor,
        e2;
    int st_epsilon,
        user_n,
        new_int,
        subtotal;
    int flag, e = 0,
        window,
a,
num,
n;

int surface[PicRow_Size][PicCol_Size];

register int j, i, l, k;

double dval;

char fract_file[100];

int Size;

int print_Flag;

int iterate;

/*
* Ask user for data file. Should be supplied at command line.
* (ie: program_name file_name <return>)
*/
if (argc < 2)
{
    fprintf(stderr, "Usage: fractal_dim data_file_name \n");
    exit(-1);
}
if ((datafile = fopen(argv[1], "r") == 0)
{
    fprintf(stderr, "!%s did not open",argv[1]);
    exit(-1);
}

/*
* Ask user for output file names. Will be supplied internally
* in this version.
*/

/*
* Change this to extract output file under new name
*/
strcpy(fract_file, "resultboard.new");

if ( (fractalfile = fopen(fract_file,"w")) == 0 )
{
    fprintf (stderr," !!!%s file did not open", fract_file);
    exit(-1);
}

strcpy(fract_file, "result.out");

if ( (result = fopen(fract_file,"a")) == 0 )
{
    fprintf (stderr,"!!!** result.out file did not open\n");
    exit(-1);
}

if (user_n >= A_size)
{
    fprintf (stderr,"ERROR::Scale size too large\n", fract_file);
    exit(-1);
}

/*
 * Ask user for scale values. These are the values used to
 * determine the number of upper and lower layers of the
 * covering method. [See Simulation, May 1992,
 * "An Initial Assessment of the Discriminant Surface
 * Complexity for Power Law Features"
 */

user_n = PrefSet_S;
num = user_n - 1;
printf ("scale_values = 2 through %d \n",num);
num = (user_n == 2) ? user_n - 1 : user_n - 2;

/*
 * Compute the window size, done only once
 */

/****************************
 * Change this to extract output file with variable window *
 * 
 *************************************************************/
window = Win_Set;

/****************************
 * Print Scale size to output file *
 *************************************************************/
#if def BIN_FORMAT
fwrite(&num,NBYTES,1,fractalfile);
#else
fprintf (fractalfile," %d\n",num);
#endif

/****************************
 * Compute Upper/Lower Surface values *
 *************************************************************/
/*** Initialize Area surfaces */
Initialize_Area (area);
print_Flag = FALSE;

B.6
/**<<<<<< Read data file into first upper surface matrix >>>>>>*/

    read_input_2 (surface,datafile);

/%%%%%%%%%%%%%%%%%%%%%%% Multiple Interation Section %%%%%%%%%%%%%%%%%%%%%%
*    Wallclock throughput results in file results.out
*    NxN sizes: 100x100 200x200 300x300 400x400 500x500
* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

for (iterate = 0;iterate < PrefRun_S;iterate++)
{

    Size = N_start + iterate * N_start;

    printf("Start %d size ::",Size);

    /******** ITERATIVE SECTION ********/
    if ((stime = times(&tim)) == -1) { /* start timing here */
        printf("Error collecting time\n");
        exit(-1);
    }

    /******** Make upper and lower surface matrixes ********/

    flag = 0; /* flag for upper surface */
    pixel_pos (surface, usurface[0], flag, Size);
    flag = 1; /* flag for lower surface */
    pixel_pos (surface, lsurface[0], flag, Size);

    st_epsilon = 0; /* initial scale factor */

    while (st_epsilon < user_n)
    {

    }
/* Make upper surface */
flag = 0; /* flag for upper surface */
pixel_pos (usurface[st_epsilon],
usurface[st_epsilon+1], flag, Size);

/* Make lower surface */
flag = 1; /* flag for lower surface */
pixel_pos (lsurface[st_epsilon],
lsurface[st_epsilon+1], flag, Size);

st_epsilon++;

/******************************
* Compute scale values
******************************/

/* Using classical least squares theory along with the
* definition of fractal dimension, we compute the
* following.
*/

st_epsilon = 0; /* initial scale factor */

while (st_epsilon < user_n)
{
    e++;  
e2 = 2.*e;

    Compute_Area (window, e2,
    usurface[st_epsilon],
    lsurface[st_epsilon],
    area[st_epsilon], Size);

    Scale Max to Min for Picture Production package
    * ras_2_13 by John Ellis, K14.
if (abs(delta_max - delta_min) < Size_epsilon) {
    scale_factor = Pic_Scale;
}
else
    scale_factor = Pic_Scale / (delta_max - delta_min);

/*
 * Place new data at the end of the linked list structure
 * ...but only for first iteration 100x100, ignore otherwise
 */

if (print_Flag == FALSE)
    Add_to_List( delta_min, delta_max,
    scale_factor, st_epsilon);

delta_min = 100000.;
delta_max = -100000.;

st_epsilon++;

} /* end of while */

print_Flag = TRUE;

time = times(&tim) - stime; /* stop timing*/

printf("Completed for %d:%d image\n", Size, Size);

printf("TIME::%g\n", eltime/HZ);

fprintf(result,"%d%f\n", Size, eltime/HZ);

} /*From for (Size)*/

/*
 * Produce Output
 */

Print_list(first_ptr);
printf("Run Completed...\n");
printf("____________________\n");
}

/*********************************************************/
/* Function Section */
/***********************************************************/

/************
* Adds output of iteration to linked list
* -------------------------------------
* /

Add_to_List (d_mn, d_mx, scale, area_num)
    float    d_mn;
    float    d_mx;
    float    scale;
    int      area_num;
{
    new_ptr = (struct Area_data *) malloc(sizeof(struct Area_data));
    (*new_ptr).d_min = d_mn;
    (*new_ptr).d_max = d_mx;
    (*new_ptr).scale_f = scale;
    (*new_ptr).area_num = area_num;
    (*new_ptr).next_ptr = first_ptr;
    first_ptr = new_ptr;
}

/************
* Computes sum over window, w, for each pixel in area
* --------------------------------------
* /

Compute_Area(w, e2, s_u, s_l, area, N_Size)
    int         w;
float e2;

int s_u[PicRow_Size][PicCol_Size],
    s_l[PicRow_Size][PicCol_Size];

float area[PicRow_Size][PicCol_Size];

int N_Size;

{
    int k, l;
    int subtotal;
    double dval;
    int Row_Size, Col_Size;

    Row_Size = Col_Size = N_Size;

    /*
     * This next section requires the most computing resources
     * 
     * Parallelization should be performed at this point to improve job throughput.
     */

    /*
     * Compute area
     */
    for (k = w; k < (Row_Size-w); k++) {
        for (l = w; l < (Col_Size-w); l++) {
            subtotal = comp_Subtotal(w, s_u, s_l,k,l);
            dval = (double) ((float)subtotal / e2);
            area[k][l] = (float) dval;
        } /* end for loop l */
    } /* end for loop k */

    /*
Find Minimum/Maximum Area values
(Used in rendering grey scale image to screen)

for (k = w; k < (Row_Size-w); k++) {
    for (l = w; l < (Col_Size-w); l++) {
        if (area[k][l] > delta_max)
            delta_max = area[k][l];
        if (area[k][l] < delta_min)
            delta_min = area[k][l];
    } /* end for loop l */
} /* end for loop k */

Summation over the window centered on pixel (k,l)

int comp_Subtotal(w, s_u, s_l,k,l)

int w;
int s_u[PicRow_Size][PicCol_Size],
     s_l[PicRow_Size][PicCol_Size];
int k, l;

int i, j;
int subtotal;
int *ptu, *ptl;

subtotal = 0;
for (i = (k-w); i <= (k+w); i++) {
    ptu = &s_u[i][l-w];
    ptl = &s_l[i][l-w];
    for (j = (l-w); j <= (l+w); j++)
        subtotal += *ptu++ - *ptl++;
}
return(subtotal);

B.12
/* 
* Initialize area array to 0 for all elements 
* ____________________________________________________
*/

Initialize_Area (area)
float area[A_size][PicRow_Size][PicCol_Size];
{
  int i,j,k;

  for (i=0;i<A_size;i++)
    for (j=0;j<PicRow_Size;j++)
      for (k=0;k<PicCol_Size;k++)
        area[i][j][k] = 0.0;
}

/* 
* Print out all data attached to linked list until NULL 
* ____________________________________________________
*/

Print_list(curr_ptr)
struct Area_data *curr_ptr;
{
  if (curr_ptr != NULL) {

    /*Recursive call* 
    Print_list((*curr_ptr).next_ptr);
    A_out ((*curr_ptr).d_min,
    (*curr_ptr).d_max,
    (*curr_ptr).scale_f,
    area[(*curr_ptr).area_num],
    fractalfile);
    }

  } 

} 

/end of program*

B.13
/* Program: pixel_pos */
/* by: Hal Hayes */
/* purpose: Computes the fractal dimension (delta) at each */
/* pixel for a NxN image for several choices of the */
/* scale parameter (epsilon). */
/* Added: N_Size elements in array */
/* date: November 9, 1992 */
/* version: 1.1 */
/* */
/* files (in): none */
/* files (out): none */

#include "fractal.h"

static void upper_inside();
static void lower_inside();

/* pixel_pos */

pixel_pos (surface_in, surface_out, surface_flag, N_Size)
    int surface_in[PicRow_Size][PicCol_Size],
        surface_out[PicRow_Size][PicCol_Size];
    int surface_flag;
    int N_Size;
{
    int n, pcount;
    int pixel[4];
    int temp_plus, temp_minus;
    int v1, v2, v3, v;
    int Row_Size, Col_Size;
    register int k, j, i;

    Row_Size = Col_Size = N_Size;
    for (i = 0; i < Row_Size; i++)
{ /* loop for rows */
   for (j = 0; j < Col_Size; j++)
   { /* loop for columns */
     if (tween(1, i, Row_Size - 2) && tween(1, j, Col_Size - 2))
       /* skip inside */
       continue;
   } /* Border Cases */

   temp_plus = surface_in[i][j] + ONE;
   temp_minus = temp_plus - 2;
   if (i == 0)
     if (j == 0)
       { /* top left corner */
         v1 = surface_in[i][j+1];
         v2 = surface_in[i+1][j];
         v = (surface_flag == UPPER) ? imax(v1,v2) : imin(v1,v2);
       }
     else if (j == Col_Size-1)
       { /* top right corner */
         v1 = surface_in[i][j-1];
         v2 = surface_in[i+1][j];
         v = (surface_flag == UPPER) ? imax(v1,v2) : imin(v1,v2);
       }
     else
       { /* top row */
         v1 = surface_in[i][j-1];
         v2 = surface_in[i][j+1];
         v3 = surface_in[i+1][j];
         v = (surface_flag == UPPER) ? imax3(v1,v2,v3) : imin3(v1,v2,v3);
       }
     else if (i == Row_Size-1)
       {...

B.15
if (j == 0)
{
    /* bottom left corner */
    v1 = surface_in[i-1][j];
    v2 = surface_in[i][j+1];
    v = (surface_flag == UPPER) ? imax(v1, v2) : imin(v1, v2);
}
else if (j == Col_Size-1)
{
    /* bottom right corner */
    v1 = surface_in[i-1][j];
    v2 = surface_in[i][j-1];
    v = (surface_flag == UPPER) ? imax(v1, v2) : imin(v1, v2);
}
else
{
    /* bottom row */
    v1 = surface_in[i][j-1];
    v2 = surface_in[i-1][j];
    v3 = surface_in[i][j+1];
    v = (surface_flag == UPPER) ? imax3(v1, v2, v3) : imin3(v1, v2, v3);
}
else
{
    if (j == 0)
    {
        /* left column */
        v1 = surface_in[i-1][j];
        v2 = surface_in[i][j+1];
        v3 = surface_in[i+1][j];
        v = (surface_flag == UPPER) ? imax3(v1, v2, v3) : imin3(v1, v2, v3);
    }
    else if (j == Col_Size-1)
    {
        /* right column */
        v1 = surface_in[i-1][j];
        v2 = surface_in[i][j-1];
        v3 = surface_in[i+1][j];
        v = (surface_flag == UPPER) ? imax3(v1, v2, v3) : imin3(v1, v2, v3);
    }
}
} /* determine maximum pixel to place in upper surface */
if (surface_flag == UPPER)
{
    surface_out[i][j] = imax(v,temp_plus);
}
else
{
    /* determine minimum pixel to place in lower surface */
    surface_out[i][j] = imin(v,temp_minus);
}
}
}

if (surface_flag == UPPER)
{
upper_inside (surface_in.surface_out, N_Size);
}
else {
lower_inside (surface_in,surface_out, N_Size);
}
return;
}

/*
  * l o w e r _ i n s i d e
  */

static void
lower_inside (in, out, N_Size)
    int in [PicRow_Size][PicCol_Size],
    out[PicRow_Size][PicCol_Size];
    int N_Size;
{
    int col, row, *pout;
    int v, center, vtb, vlr;
    int Row_Size, Col_Size;
    /*

* Pass value-by-reference
* increments pointer pout rerencing memory position of out
*/

/*
* Increment over picture array for all pixels
*/

Row_Size = Col_Size = N_Size;

for (row = 1; row <= Row_Size-2; row++) {
    for (col = 1; col <= Col_Size-2; col++) {
        center = in[row][col] - 1;
        vtb = imin(in[row-1][col],in[row+1][col]);
        vlr = imin(in[row][col-1],in[row][col+1]);
        v = imin(vtb, vlr);
        out[row][col] = imin(v, center);
    }

return;
}

/*
* upper_inside
*/

static void
upper_inside (in, out, N_Size)

    int in [PicRow_Size][PicCol_Size],
    out[PicRow_Size][PicCol_Size];

    int N_Size;

    {
        int col, row;
        int v, center, vtb, vlr;
        int Row_Size, Col_Size;

    /*
* Pass value-by-reference
* increments pointer out referencing memory position of out
*/

/*
 * Increment over picture array for all pixels
*/

Row_Size = Col_Size = N_Size;

for (row = 1; row <= Row_Size-2; row++) {
    for (col = 1; col <= Col_Size-2; col++) {
    
        center = in[row][col] + 1;
        vtb = imax(in[row-1][col], in[row+1][col]);
        vlr = imax(in[row][col-1], in[row][col+1]);
        v = imax(vtb, vlr);
        out[row][col] = imax(v, center);
    }
}

return;

/**************************************************************************/
* file: N_readfractal.c                                  *
* by:     Hal Hayes                                    *
* purpose:read binary input file of image              *
* date:   November 9, 1992                             *
* version:1.1                                         *
  * added fclose operation to datafile                *
  * files (in): fractal.h datafile                    *
  * files (out):none                                *
/***************************************************************************/

#include “fractal.h”
read_input_2 (image, datafile)
    int image[PicRow_Size][PicCol_Size];
    FILE *datafile;
{
    fread (image, sizeof(int), PicRow_Size*PicCol_Size, datafile);
    fclose (datafile);
    return;
}

A_out (d_mn, d_mx, scale, area, fractalfile)
    float d_mn;
    float d_mx;
    float scale;
    float area[PicRow_Size][PicCol_Size];
    FILE *fractalfile;
{
    int i,j;
    int count;
    int junk;

    printf("scale = %f\n", scale =
    printf("\n%fdelta_max = %f\n", delta_max =
    printf("\n%fdelta_min = %f\n", delta_min =

/* READ_INPUT_2 */

/*******************************************************************/
* Program: N_writeout.c
* by: Hal Hayes
* purpose: Produces binary file from fractal image output
* date: November 9, 1992
* version: 1.0
*
* files (in): none
* files (out): resultboard.new
*******************************************************************/

#include "fractal.h"

#include "fractal.h"

#include "fractal.h"

#include "fractal.h"
scale,d_mx,d_mn);

    fprintf(fractalfile,"%fn\n",d_mn);
    fprintf(fractalfile,"%fn\n",d_mx);
    fprintf(fractalfile,"%fn\n",scale);
#endif
    for (i=0;i<PicRow_Size;i++) {
        count = 0;
        for (j=0;j<PicCol_Size;j++) {
            fprintf(fractalfile,"%g\n",area[i][j]);
            count++;
            if (count == 10) {
                fprintf(fractalfile,"\n");
                count = 0;
            }
        }
        fprintf(fractalfile,"\n");
    }
#endif
/**** >>>>>>>>>>>Binary Format<<<<<<<<< ***/
/************** ^^^^^ ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*/

junk = fwrite (&d_mn,FBYTES,1,fractalfile);
    printf("D_MN fwrite :: %d\n",junk);

fwrite (&d_mx,FBYTES,1,fractalfile);
fwrite (&scale,FBYTES,1,fractalfile);
fwrite (area, FBYTES, PicRow_Size*PicCol_Size, fractalfile);
#endif
}

B.21
B.1.1 Header Files for Single Processor Code

/*********************************************************
* file: fractal.h
* by: Hal Hayes
* purpose: Computes the fractal dimension (delta) at each
* pixel for a NxN image for several choices of the
* scale parameter (epsilon).
* date: November 9, 1992
* version: 1.3
***********************************************************/

/***************************************************************/
* Changes: Added definition Size_epsilon
***************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/types.h>
#include <sys/times.h>
#include <sys/param.h>

#define ONE
#define NBYTES sizeof(int)
#define FBYTES sizeof(float)
#define TRUE
#define FALSE
#define imin(a,b) (((a)< (b))? (a): (b))
#define imax(a,b) (((a)> (b))? (a): (b))
#define UPPER
#define imin3(a,b,c) (imin (imin(a,b),c))
#define imax3(a,b,c) (imax (imax(a,b),c))
#define tween(a,x,b) (((a)<= (x)) && ((x) <= (b)))

/***********************************************************/
* Define the Picture Size *
***********************************************************/

B.22
#define PicRow_Size 600
/* Number of Pixel Rows*/
#define PicCol_Size 600
/* Number of Pixel Columns */

/ *******************************************************/

/* Define the Array Sizes */
/ *******************************************************/

#define A_size 6 /* Allows up to scale of 5 */

/ *******************************************************/

/* Define the File Pointers */
/ *******************************************************/

/** #define BIN_FORMAT ****/

#define Size_epsilon 0.05
#define N_start 50
#define Win_Set 5 /* preset window size*/
#define PrefSet_S 5 /* scale value */
#define PrefRun_S 10 /* >>no. of iterations<<*/
/ * ^^^^^^^^^ ^ ***/

/ *** This variable must be equal to N_Size / 50 ***/
B.2 CM200 CONNECTION MACHINE

Program: Fractal_area
by: Hal Hayes
for: K12, researchers Rogers/Solka/Priebe
purpose: Computes the fractal dimension (delta) at each
pixel for a NxN image for several choices of the
scale parameter (epsilon).
date: December 4, 1992
version: 4.1

* CONNECTION MACHINE VERSION for CM200 at NRL
* ^^^^^^^^^^^^^^ _ ^^^^^ ^^^^^ ^^^^^ ^^^^^
*
* files (in): image file (~)
* N_pixel_pos.c <include>
* N_readfractal.c <include>
* N_writeout.c <include>
* fractal.h <include>
* files (out): image file enhanced by covering method for
* features, name as specified by user when asked.
* [ THIS OPTION DISABLED ]
*
*****************************************************************
CHANGES:
Modified for Connection Machine CM200
*****************************************************************

#include "fractal.h"
#include "cmfile.h"
#include "move_array.h"

pixel_pos (); /* performs covering method */
para_pixel_pos(); /* performs covering on ParaVars */
Compute_Area (); /* computes area from ParaVars */
/*
 * EXTERNAL FUNCTION CALLS
 */

extern void read_input_2 (); /* reads in image as binary */
/*** from N_readfractal.c ***/

extern void A_out (); /* output data in ascii */
/*** from N_writeout.c ***/

extern double get_time(); /* computes total time */
/*** from calc_total_time.c ***/

/*
 * DATA STRUCTURES
 */

/*
 * Area_data structure used to hold results in a lined
 * list.
 */

struct Area_data {
    float d_max,
    d_min;
    float scale_f;
    int area_num;
    struct Area_data *next_ptr;
};

/*
 * first_ptr points to head of linked list of results.
 * new_ptr points to a newly created node.
 */

struct Area_data *first_ptr = NULL;
struct Area_data *new_ptr;

/*
 * declare time variables for accounting

double stime, eltime;

/*
 * area holds the real values of area computed from
 * the upper and lower surfaces for up to A_size iterations.
 */

static float area[A_size][P_varSize][P_varSize];

/**< delta_max - maximum area value */
/**< delta_min - minimum area value */

float delta_max, delta_min;

FILE *datafile, *result;
float Pic_Scale = 255;

 /*******************************************************************************
  * Main Program Section
  *  ******************************************************************************/

main(argc argv)
  int argc;
  char *argv[];
{

  /*******************************************************************************
  * Variable Section
  *  ******************************************************************************/

  float scale_factor,
  c2;
  int st_epsilon,
  user_n;
  int e = 0,
window,
num;

int surface[PicRow_Size][PicCol_Size];

int fname;

char stub[10];
char fract_file[100];

int Size;

int iterate;

/***************************************************************
 * Open Image file and result output file                      *
 ****************************************************************/

/*
 * Ask user for data file. Should be supplied at command line.
 * (ie: program_name file_name <return>)
 */
if (argc < 2)
{
    fprintf (stderr, "Usage: fractal_dim data_file_name \n");
    exit(-1);
}
if ((datafile = fopen(argv[1], "r")) == 0)
{
    fprintf (stderr, "!!%s did not open",argv[1]);
    exit(-1);
}
fname = IMAGE_SIZE;
strcpy(stub, "result.out");
sprintf(fract_file,"%s%d",stub,fname);
if ((result = fopen(fract_file,"a")) == 0)
{  
fprintf (stderr,"!! result.out file did not open\n");  
exit(-1);  
}

/**** End of open file operations ****/

delta_max = -100000.0;  
delta_min = 100000.0;  

user_n = PrefSet_S;  

num = user_n - 1;  
printf ("scale_values = 2 through %d \n",num);  

num = (user_n == 2) ? user_n - 1 : user_n - 2;

/* Change this to extract output file with variable window */

window = Win_Set;

/* Read data file into first upper surface matrix */

read_input_2 (surface.datafile);

/* Multiple Iteration Section */

wallclock throughput results in file results.out

NxN sizes: 100x100 200x200 300x300 400x400 500x500

* REMOVED/MODIFIED MULTIPLE RUNS *

************ Read data file into first upper surface matrix *************/
for (iterate = 0; iterate < PrefRun_S; iterate++)
{

Size = IMAGE_SIZE;

with (ParaVar) {

Lsurf_i = 0;
Usurf_i = 0;
A_para_Surf = 0;
}

printf("Start %d size ::",Size);

/*/ ITERATIVE SECTION */
if ((stime = get_time()) == -1) {/*start timing here*/
printf("Error collecting time\n");
exit(-1);
}

/* Make first upper and lower surface matrixes */
pixel_pos (surface);

/*********************************************************************************/
/* Compute scale values */
*********************************************************************************/

/* Using classical least squares theory along with the
* definition of fractal dimension, we compute the
* following.
*/

st_epsilon = 0; /* initial scale factor */
e = 0;
while (st__epsilon < user_n)
{
    e++;
    e2 = 2. * e;

    Compute_Area ( e2, area[st__epsilon]);

    /***************************************************************************/
    /* Compute next Upper & Lower surfaces                              */
    /***************************************************************************/

    para_pixel_pos();

    /***************************************************************************/
    /* Scale Max to Min for Picture Production package                  */
    /* ras_2_13 by John Ellis, K14.                                       */
    /***************************************************************************/

    if (abs(delta_max - delta_min) < Size_epsilon) {
        scale_factor = Pic_Scale;
    }
    else
        scale_factor = Pic_Scale / (delta_max - delta_min);

    /***************************************************************************/
    /* Place new data at the end of the linked list structure            */
    /* ...but only for first iteration 100x100, ignore otherwise          */
    /***************************************************************************/

    Add_to_List( delta_min, delta_max,
                 scale_factor, st__epsilon);

    delta_min = 100000.;
    delta_max = -100000.;

    st__epsilon++;

} /* end of while */
etime = get_time() - stime; /* stop timing */

printf("Completed for %3d:%3d image\n",Size,Size);

printf("TIME::%g\n",etime);

fprintf(result, "%3d%g\n",Size,(etime));

} /* From for (Size) */

/****************************************************
*   End of Loop for computing different sized images  *
* ****************************************************/


/*
 * Produce Output
 */

Print_list(first_ptr);

printf("Run Completed...\n");
printf("______________\n");
}

/***********************************************
*   Function Section                          *
* ***********************************************/

/
* Adds output of iteration to linked list
* ----------------------------------------
*/

Add_to_List (d_mn, d_mx, scale, area_num)
  float d_mn;
  float d_mx;
  float scale;
  int area_num;
{ 
 new_ptr = (struct Area_data *)
 malloc(sizeof(struct Area_data));
 (*new_ptr).d_min = d_mn;
 (*new_ptr).d_max = d_mx;
 (*new_ptr).scale_f = scale;
 (*new_ptr).area_num = area_num;
 (*new_ptr).next_ptr = first_ptr;
 first_ptr = new_ptr;
 }

Print_list(curr_ptr)
 struct Area_data *curr_ptr;
 {
  if (curr_ptr != NULL) {

  /* ***** _______ Recursive call ________ *****/
  Print_list((*curr_ptr).next_ptr);
  A_out ((*curr_ptr).d_min,
  (*curr_ptr).d_max,
  (*curr_ptr).scale_f,
  (*curr_ptr).area_num);
  }
  }
}
/*****************************************************************
* Program: pixel_pos for the Connection Machine CM200
* by: Hal Hayes
* for: K12, researchers Rogers/Solka/Priebe
* purpose: Computes the fractal dimension (delta) at each
* pixel for a NxN image for several choices of the
* scale parameter (epsilon).
* Added : N_Size elements in array
* date: December 4, 1992
* version: 2.1
*******************************************************************/

int surface_in[PicRow_Size][PicCol_Size];
{

Pixel_pos (surface_in)

B.33
Temp >?= from_torus(&Usurf_i, 1,1);
Usurf_i = Temp;

/*** Computer Lower Surface values ***/
    Temp = Lsurf_i - 1;
    Temp <?= from_torus(&Lsurf_i,-1,0);
    Temp <?= from_torus(&Lsurf_i, 1,0);
    Temp <?= from_torus(&Lsurf_i,-1,1);
    Temp <?= from_torus(&Lsurf_i, 1,1);
    Lsurf_i = Temp;
}
/** Function: Computer_Area
* purpose: Computes the fractal dimension (delta) at each
* pixel for a NxN image for several choices of the
* scale parameter (epsilon).
* Added : N_Size elements in array
* date: December 4, 1992
* version: 2.1
* files (in): none
* files (out): none
*/

Compute_Area(e2, area)

float e2;
float area[P_varSize][P_varSize];

{
/*
* This next section requires the most computing resources
* ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
* Parallelization should be performed at this
* point to improve job throughput.
*/

with (ParaVar) {
    a_Hold = Usurf_i - Lsurf_i;
    Temp = a_Hold + from_torus_dim(&a_Hold, 0, 1)
    + from_torus_dim(&a_Hold, 0, -1);
    Temp = Temp + from_torus_dim(&Temp, 0, 4)
+ from_torus_dim(&Temp, 0, -4);
Temp = Temp + from_torus_dim(&a_Hold, 0, 2)
+ from_torus_dim(&a_Hold, 0, -2);

b_Hold = Temp + from_torus_dim(&Temp, 1, 1)
+ from_torus_dim(&Temp, 1, -1);
b_Hold = b_Hold + from_torus_dim(&b_Hold, 1, 4)
+ from_torus_dim(&b_Hold, 1, -4);
b_Hold = b_Hold + from_torus_dim(&Temp, 1, 2)
+ from_torus_dim(&Temp, 1, -2);
A_para_Surf = (float:ParaVar) b_Hold / e2;

/*** Computer Maximum and Minimum values in Area ***/
delta_max >?= A_para_Surf;
delta_min <? A_para_Surf;

="/************************************************************************
* End of Program *
************************************************************************/

B.36
#include "fractal.h"

/*
read_input_2
*/

read_input_2 (image, datafile)
int image[IMAGE_SIZE][IMAGE_SIZE];
FILE *datafile;
{
    fread (image, sizeof(int), IMAGE_SIZE*IMAGE_SIZE, datafile);
fclose (datafile);
return;
}

} /* READ_INPUT_2 */
#include "fractal.h"

A_out (d_mn, d_mx, scale, area)
float d_mn;
float d_mx;
float scale;
int area;
{
    printf("%d::scale = %delta_max = %f delta_min = %f \n",
        area, scale, d_mx, d_mn);
}

#include <sys/time.h>
#include <sys/param.h>

#define Micron 1000000

double get_time()
{
    struct timeval justime;
    struct timezone *tz;
    double micron_s, seconds, mytime;

tz = NULL;
if (gettimeofday(&justtime, tz) != 0) {
    printf("Timing Error.\n");
    return -1;
}

micron_s = ((double) justime.tv_usec) / Micron;
seconds = (double) (justime.tv_sec);

mytime = seconds + micron_s;

return mytime;
}
File: move_array.cs

Used by the CM200 to move a serially stored array into a parallel variable.

```c
#include <cm/paris.h>
#include "move_array.h"

unsigned move_array_offset[32] =
{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};
unsigned move_array_axes[32] =
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31};

#define DMS1 unsigned d0_len
#define DMS2 unsigned d0_len, unsigned d1_len
#define DMS3 unsigned d0_len, unsigned d1_len, unsigned d2_len
#define DMS4 unsigned d0_len, unsigned d1_len, unsigned d2_len, unsigned d3_len
#define DMS5 unsigned d0_len, unsigned d1_len, unsigned d2_len, unsigned d3_len, unsigned d4_len
#define DMS6 unsigned d0_len, unsigned d1_len, unsigned d2_len, unsigned d3_len, unsigned d4_len, unsigned d5_len
#define DMS7 unsigned d0_len, unsigned d1_len, unsigned d2_len, unsigned d3_len, unsigned d4_len, unsigned d5_len, unsigned d6_len
#define DMS8 unsigned d0_len, unsigned d1_len, unsigned d2_len, unsigned d3_len, unsigned d4_len, unsigned d5_len, unsigned d6_len, unsigned d7_len

#define DCL1 unsigned ds[1]; ds[0]=d0_len;
#define DCL2 unsigned ds[2]; ds[0]=d0_len; ds[1]=d1_len;
#define DCL3 unsigned ds[3]; ds[0]=d0_len; ds[1]=d1_len; ds[2]=d2_len;

#define CM_s_read_array_by_cube_addresses CM_read_array_by_cube_addresses
#define CM_s_write_array_by_cube_addresses CM_write_array_by_cube_addresses
#define MAKE_GET_ARRAY(type.type_abrv,type_size,size)\p void get_array_from_cm_send
(type:current * cm_ptr,\ptype * front_end_ptr,\punsigned length)
{\p CM_s/**/type_abrv/**/read_array_by_cube_addresses(front_end_ptr,\p size, 0, 0, length, cm_ptr,type_size);}
```
#define MAKE_PUT_ARRAY(type,type_abrv,type_size,size)\pvoid put_array_on_cm_send\(type * front_end_ptr,\ptype:current * cm_ptr,\punsigned length)\}\pCM/**/type_abrv/**/_write_array_by_cube_addresses(front_end_ptr,\psize, 0, 0, length, cm_ptr, type_size);\

#define MAKE_GET_ARRAY_GRID(type,type_abrv,type_size,dims,dcls,n_dims)\pvoid get_array_from_cm_grid(type:current * cm_ptr,\ptype * front_end_ptr,\pdims)\{\pdcls \pCM/**/type_abrv/**/_read_from_news_array_L((char *)front_end_ptr,\pmove_array_offsets, move_array_offsets, ds, move_array_axes, cm_ptr, type_size,\pn_dims, ds, sizeof(type));\}

#define MAKE_PUT_ARRAY_GRID(type,type_abrv,type_size,dims,dcls,n_dims)\pvoid put_array_on_cm_grid(type * front_end_ptr,\ptype:current * cm_ptr,\pdims){\pdcls \pCM/**/type_abrv/**/_write_to_news_array_L((char *) front_end_ptr.\pmove_array_offsets, move_array_offsets, ds, move_array_axes, cm_ptr, type_size,\pn_dims, ds, sizeof(type));\}

#define MAKE_ARRAY_MOVES_GRID(type,type_abrv,type_size,dims,dcls,n_dims)\pMAKE_GET_ARRAY_GRID(typemake_array_MOVES_GRID(type,type_abrv,type_size,dims,dcls,n_dims)\pMAKE_GET_ARRAY_GRID(type,type_abrv.type_size,\psize) /* remove this line if you don’t have gcc */\pMAKE_ARRAY_MOVES_GRID(type,type_abrv.type_size,\psize) /* remove this line if you don’t have gcc */\pMAKE_ARRAY_MOVES_GRID(type,type_abrv.type_size,\psize,CM_long_array)\pMAKE_ARRAY_MOVES_GRID(type,type_abrv.type_size,\psize,CM_float_single_array)\pMAKE_ARRAY_MOVES_GRID(type,type_abrv.type_size,\psize,CM_long_array)\pMAKE_ARRAY_MOVES_GRID(type,type_abrv.type_size,\psize,CM_byte_array)

#define S_SIZE 32
#define F_SIZE 23,8
#define D_SIZE 52,11
#define U_SIZE 32
#define C_SIZE 8
#define SH_SIZE 16
#define B_SIZE 1

#define u_char unsigned char
#define u_short unsigned short

MAKE_ARRAY_MOVES(int,s,S_SIZE,CM_long_array)
MAKE_ARRAY_MOVES(float,f,F_SIZE,CM_float_single_array)
MAKE_ARRAY_MOVES(double,f,D_SIZE,CM_float_double_array)
MAKE_ARRAY_MOVES(unsigned,u,U_SIZE,CM_long_array)
MAKE_ARRAY_MOVES(char,s,C_SIZE,CM_byte_array)
MAKE_ARRAY_MOVES(u_char,u,C_SIZE,CM_byte_array)
MAKE_ARRAY_MOVES(short,s,SH_SIZE,CM_short_array)
MAKE_ARRAY_MOVES(u_short,u,SH_SIZE,CM_short_array)
MAKE_ARRAY_MOVES(bool,u,B_SIZE,sizeof(bool))

#undef_MA_DMS_1
#undef_MA_DMS_2
#undef_MA_DMS_3
#undef_MA_DMS_4
#undef_MA_DMS_5
#undef_MA_DMS_6
#undef_MA_DMS_7
#undef_MA_DMS_8

#undef_MAKE_GET_ARRAY
#undef_MAKE_PUT_ARRAY
#undef_MAKE_GET_ARRAY_GRID
#undef_MAKE_PUT_ARRAY_GRID
#undef_MAKE_ARRAY_MOVES_GRID
#undef_MAKE_ARRAY_MOVES

#undef u_char
#undef u_short
B.2.1 Header Files for CM200

/******************************************************************************
* File: move_array.h
* Header file for move_array.cs
* Used by the CM200 to move a serially stored array into a parallel
* Variable.
*******************************************************************************/
#define get_array_from_em get_array_from_em_grid
#define put_array_on_cm put_array_on_cm_grid

overload get_array_from_cm_send, get_array_from_cm_grid,
put_array_on_cm_send, put_array_on_cm_grid;

#define _MA_DMS_1 unsigned d0_len
#define _MA_DMS_2 unsigned d0_len, unsigned d1_len
#define _MA_DMS_3 unsigned d0_len, unsigned d1_len, unsigned d2_len
#define _MA_DMS_4 unsigned d0_len, unsigned d1_len, unsigned d2_len,
    unsigned d3_len
#define _MA_DMS_5 unsigned d0_len, unsigned d1_len, unsigned d2_len,
    unsigned d3_len, unsigned d4_len
#define _MA_DMS_6 unsigned d0_len, unsigned d1_len, unsigned d2_len,
    unsigned d3_len, unsigned d4_len, unsigned d5_len
#define _MA_DMS_7 unsigned d0_len, unsigned d1_len, unsigned d2_len,
    unsigned d3_len, unsigned d4_len, unsigned d5_len, unsigned d6_len
#define _MA_DMS_8 unsigned d0_len, unsigned d1_len, unsigned d2_len,
    unsigned d3_len, unsigned d4_len, unsigned d5_len, unsigned d6_len,
    unsigned d7_len

#define MAKE_GET_ARRAY(type) overload void get_array_from_cm_send(type * cm_ptr,
    type * front_end_ptr, unsigned length);

#define MAKE_PUT_ARRAY(type) overload void put_array_on_cm_send(type * front_end_ptr,
    type * cm_ptr, unsigned length);

#define MAKE_GET_ARRAY_GRID(type, dims) overload void get_array_from_cm_grid(type * cm_ptr,
    type * front_end_ptr, unsigned length);

#define MAKE_PUT_ARRAY_GRID(type, dims) overload void put_array_on_cm_grid(type * front_end_ptr,
    type * cm_ptr, unsigned length);

#define MAKE_ARRAY_MOVES_GRID(type, dims) MAKE_GET_ARRAY_GRID(type, dims)
    MAKE_PUT_ARRAY_GRID(type, dims)

#define MAKE_ARRAY_MOVES(type) MAKE_PUT_ARRAY(type) /* remove this line if you don’t have gcc */
    MAKE_GET_ARRAY(type) /* remove this line if you don’t have gcc */
    MAKE_ARRAY_MOVES_GRID(type, _MA_DMS_1)
    MAKE_ARRAY_MOVES_GRID(type, _MA_DMS_2)
    MAKE_ARRAY_MOVES_GRID(type, _MA_DMS_3)
    MAKE_ARRAY_MOVES_GRID(type, _MA_DMS_4)
    MAKE_ARRAY_MOVES_GRID(type, _MA_DMS_5)
    MAKE_ARRAY_MOVES_GRID(type, _MA_DMS_6)
typedef MA_DMS_7

#define u_char unsigned char
#define u_short unsigned short

MAKE_ARRAY_MOVES(int)
MAKE_ARRAY_MOVES(float)
MAKE_ARRAY_MOVES(double)
MAKE_ARRAY_MOVES(unsigned)
MAKE_ARRAY_MOVES(char)
MAKE_ARRAY_MOVES(u_char)
MAKE_ARRAY_MOVES(short)
MAKE_ARRAY_MOVES(u_short)
MAKE_ARRAY_MOVES(bool)

#undef MA_DMS_1
#undef MA_DMS_2
#undef MA_DMS_3
#undef MA_DMS_4
#undef MA_DMS_5
#undef MA_DMS_6
#undef MA_DMS_7
#undef MA_DMS_8

#undef MAKE_GET_ARRAY
#undef MAKE_PUT_ARRAY
#undef MAKE_GET_ARRAY_GRID
#undef MAKE_PUT_ARRAY_GRID
#undef MAKE_ARRAY_MOVES_GRID
#undef MAKE_ARRAY_MOVES

#undef u_char
#undef u_short

B.44
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/types.h>
#include <sys/times.h>
#include <sys/param.h>

#define ONE 1
#define NBYTES sizeof(int)
#define FBYTES sizeof(float)
#define TRUE 1
#define FALSE 0

#define UPPER 0

/* Define the Picture Size */

/* Define the Array Sizes */
#define A_size 6 /* Allows up to scale of 5 */

/**********************************************************************************
* Define the File Pointers *
***********************************************************************************/

/*** #define BIN_FORMAT ****/

#define Size_epsilon 0.05
#define Win_Set 5 /* preset window size */
#define PrefSet_S 5 /* scale value */
#define PrefRun_S 25 /* >>no. of iterations<<*/

#define IMAGE_SIZE 500
#define N_start 500
#define PicRow_Size 500 /* Number of Pixel Rows */
#define PicCol_Size 500 /* Number of Pixel Columns */