Abstract

Parallel computing has been quite successful in solving problems that have very regular structure, because the structure naturally leads to a balanced allocation of data and computations across the processors and to efficient communication between them. Examples of such problems can be found in matrix computations, signal/image processing, and the natural sciences. However, in many important mathematical, scientific, and industrial problems, data access patterns are irregular and evolve during the computation. The parallelization of these problems poses difficulties related to data partitioning, data communication, and load balancing.

This dissertation examines particular types of irregular problems, known as hierarchical clustering applications, and identifies the language and runtime support needed for their efficient execution on distributed memory machines.

Hierarchical clustering applications are found in a variety of disciplines, including physics, image processing, document retrieval, biology, and the social sciences. These applications start with a set of objects and a set of variables describing these objects. The aim is to divide the objects into groups or clusters that satisfy certain constraints. The clusters are built gradually by putting the most similar objects together and producing a nested sequence or hierarchy of partitions that can be represented by trees or dendrograms.

The language and runtime support needed for the execution of hierarchical clustering applications on distributed memory machines is determined as follows. Starting with a definition of hierarchical clustering, a graph-theoretic formulation of hierarchical clustering, using the multigraph concept, is presented. This formulation is used to describe a programming paradigm for hierarchical clustering. The paradigm is applied to a representative clustering application, region growing, and is implemented on a distributed memory machine in both the data parallel and message passing models. The parallel implementations provide insight into the language and runtime support needed for region growing in particular, and for hierarchical clustering applications in general.

The language and runtime support needed for the execution of hierarchical clustering applications is described using High Performance Fortran (HPF) and is divided into the following categories: data distribution, runtime data re-distribution, unstructured communication, processors, data structures, and library routines. The results of the study are useful for the application programmer, parallel language designer, as well as the compile-time and runtime systems developer.

LANGUAGE AND RUNTIME SUPPORT FOR THE EXECUTION OF HIERARCHICAL CLUSTERING APPLICATIONS ON DISTRIBUTED MEMORY MACHINES

by

NAWAL COPTY

B.S., American University of Beirut, 1980
M.Sc., University of Aston in Birmingham, 1982
M.S., Syracuse University, 1991

DISSERTATION

Submitted in partial fulfilment of the requirements for the degree of
Doctor of Philisophy in Computer and Information Science
in the Graduate School of Syracuse University

December 1995

Approved ________________________
Professor Geoffrey C. Fox

Date ________________________

Committee Approval Page

To Hassoun

Contents

List of Figures

List of Tables

Preface

I would like to thank Professor Geoffrey Fox, my thesis advisor, for his help, support, and kindness throughout this research project. He has continually provided me with guidance, "vision", and inspiration. It has been a great pleasure to know him and work with him.

Special thanks to my committee members, Dr. Tomasz Haupt and Dr. Paul Coddington for their time, help, and guidance. I owe much to their contributions, their direction, and feedback which made this work possible. Thanks to Professor Ernest Sibert for his guidance and careful reading of the dissertation. I especially appreciate his interesting and insightful comments. I am grateful to Professor Carlos Hartmann for his kindness, help, and support throughout my years at Syracuse University.

Many others have helped and inspired me. Thanks to Professor Brinch Hansen for introducing me to parallel programming. Special thanks to Professor Sanjay Ranka and my colleague Ravi Shankar for their direct contributions to the work on region growing. I am grateful to Elaine Weinman for her careful editing of this dissertation.

I would also like to thank my parents, Kamel and Samira, for their continuous love, patience, and support that have made possible my accomplishments and successes. I am grateful to my brothers, Nader and Nadim, for their love and for believing that I could complete this dissertation. Many thanks to my friend Donald Pratt for his patience and support.

Finally, special thanks to (Dr.) Hassan Diab, my companion and my friend throughout this long journey.

Chapter 1

Introduction and Motivation

Since the 1980's there has been a significant growth in the use of parallel computers to solve a variety of scientific problems. The advent of parallel computing has not only provided faster solutions to computationally intensive problems, but has also provided new answers to scientific questions [44].

There are two main motivations for using parallel computers. The first motivation is speed. By using more than one processor, more processing power is available to solve a given problem, and so it may be solved in a shorter length of time. This is especially important for the so-called *Grand Challenge* applications [61] such as climate modeling, quantum chromo-dynamics, structural aerodynamic calculations, and other applications that require vast computational resources. The second motivation for using parallel computing is the lower *price per unit performance* of a parallel computer in comparison to that of a very fast sequential machine. It is less costly to obtain a certain level of performance by using a number of moderately fast processors connected together than by using a very fast sequential machine.

Parallel computing has been quite successful in solving problems that have very regular structure, because the structure naturally leads to a balanced allocation of data and computations across the processors and to efficient communication between them. Examples of such problems can be found in matrix computations, signal/image processing, and the natural sciences.

However, in many important mathematical, scientific, and industrial problems, data access patterns (or data dependencies) are irregular and evolve during the computation. Examples of these applications include molecular dynamics simulations, N-body simulations, partial differential equation solvers using unstructured or adaptive meshes, and sparse matrix problems. The parallelization of these and other irregular problems poses further difficulties related to data distribution, data communication, and load balancing.

There have been many approaches that aim to simplify the task of developing correct, efficient parallel programs. These approaches include:

1. The development of parallelizing compilers such as Fortran D [40, 58], FORGE Explorer/DMP [37], and SUIF [108] that transform a sequential program into message-passing code that runs on a distributed memory machine.

2. The design of high-level programming languages that hide the details of the architecture from the programmer. Examples of data parallel languages include *Lisp [95], CM Fortran [94], C* [97], High Performance Fortran (HPF) [43, 62, 63, 68], Vienna Fortran [26], and pC++ [15, 16]. Examples of task or control parallel languages include Joyce [19], SuperPascal [21], Fortran M [38], and Compositional C++ [24].

3. The development of runtime support libraries. The implementation of a high-level language, like HPF, on a distributed memory machine requires both a compile-time and a runtime system. The compile-time system generates the

low-level code appropriate for the target machine. The runtime system is a collection of library routines that perform address translations, data movements, and computation partitioning, as well as other useful functions. In cases when these functions are data dependent and cannot be completely specified at compile-time, the compiler inserts calls to the library routines at appropriate points in the object code. Examples of runtime support libraries include the CHAOS library [81, 80] which provides primitives for distributing irregular data and computations and for optimizing communication, and the NICE library [83, 104] which provides primitives for collective communication. A major research goal is to identify common compile-time and runtime functions that can be shared by a variety of languages [77].

4. The design of message passing systems such as Express [78], PVM [48], and MPI [51] that offer portable communication routines.

5. The identification of parallel programming paradigms [20, 22, 47], skeletons [30], templates [9], archetypes [4, 25], or programming models that the programmer can use as building blocks in solving certain classes of problems.

This dissertation contributes to the efforts aimed at simplifying the task of developing parallel programs. It examines particular types of irregular problems known as hierarchical clustering applications and identifies the language and runtime support needed for their efficient execution on distributed memory machines. The results of this study are useful for the application programmer and the parallel language designer, as well as the compile-time and runtime systems developer.

Hierarchical clustering applications are representative of types of irregular problems, known as *loosely synchronous* problems, whose data objects evolve during computation in a time synchronized manner [42]. Examples of these clustering applications are found in a variety of disciplines, including physics, image processing, document retrieval, biology, and the social sciences. These applications start with a set of objects and a set of variables describing them. The aim is to divide the objects into groups or clusters that satisfy certain constraints. The clusters are built gradually by putting the most similar objects together and producing a nested sequence or hierarchy of partitions that can be represented by trees or dendrograms.

The language and runtime support needed for the execution of hierarchical clustering applications on distributed memory machines is determined as follows. Starting with a definition of hierarchical clustering, a graph-theoretic formulation of hierarchical clustering using the multigraph concept is presented. This formulation is used to describe a programming paradigm for hierarchical clustering. The paradigm is applied to a representative clustering application, region growing, and is implemented on a distributed memory machine in both the data parallel and message passing models. The parallel implementations provide insight into the language and runtime support needed for region growing in particular, and for hierarchical applications in general.

The language and runtime support needed for the execution of hierarchical clustering applications is described using High Performance Fortran (HPF) and is divided into the categories of data distribution, runtime data re-distribution, unstructured communication, processors, data structures, and library routines.

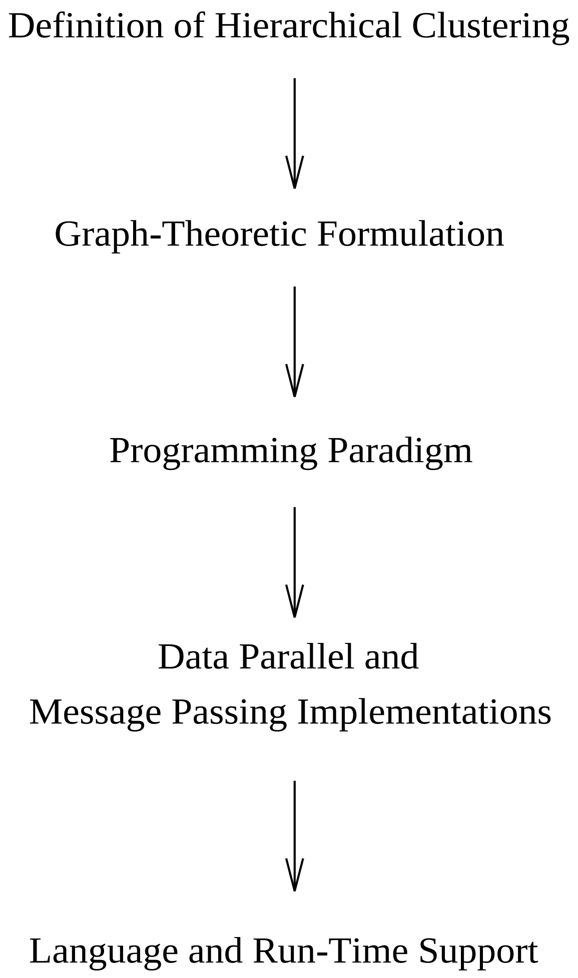

Definition of Hierarchical Clustering
Graph-Theoretic Formulation
Programming Paradigm
Data Parallel and
Message Passing Implementations
Language and Run-Time Support

Figure 1.1: Steps in Identifying the Language and Runtime Support Needed for Clustering Applications

Chapter 2

Overview of Parallel Processing Concepts

A parallel computer is comprised of a set of processors that are able to work cooperatively to solve a *single* computational problem. This definition is broad enough to include parallel supercomputers, networks of workstations, multiple-processor workstations, and embedded systems [39].

2.1 Models of Parallel Programming

Many models have been proposed for parallel programming. These models provide various ways of viewing and representing a parallel computation. They do not do not necessarily reflect the underlying physical machine architectures. Foster [?] presents the following models of parallel programming:

- **Shared Memory:** In the shared memory programming model, processes share a common, single address space which they can read and write to asynchronously. Mechanisms, such as locks and semaphores may be used to control access to the data.

- **Message Passing:** In the message passing programming model (also known as control or task parallel model), a computation comprises one or more processes, each with its own private address space. Two processes share data by sending and receiving messages.

  If all processes execute the same program, then the model is referred to as Single Program Multiple Data (SPMD). If the processes execute different programs, then the model is referred to as Multiple Program Multiple Data (MPMD). CMMD [94] and MPI [51] are two communication libraries that support the SPMD and MPMD models, respectively.

- **Data Parallel:** In the data parallel programming model, concurrency is derived from the application of the same operation on many elements of a data structure. A data parallel program consists of a sequence of such tasks. Each operation on each element can be viewed as a single process, and hence the granularity of a data parallel computation is small. Examples of data parallel languages include CM Fortran [94], High Performance Fortran (HPF) [43, 62, 63, 68].

2.2 Parallel Machines

Two distributed memory machines, the Connection Machine model CM-5 and a cluster of DEC Alpha workstations, were used to implement hierarchical clustering. The machines are described below.

2.2.1 The Connection Machine Model CM-5

The Connection Machine model CM-5 [93] is an MIMD machine from Thinking Machines Corporation (TMC). The machine is composed of a control processor (also known as partition manager) and tens or hundreds of processing nodes connected together in the form of a *fat tree* [71]. The CM-5 at the Northeast Parallel Architectures Center (NPAC) at Syracuse University that was used in this study has 32 such processing nodes.

Every processing node is a general-purpose computer that can fetch and interpret its own instruction stream, execute arithmetic and logical instructions, calculate memory addresses, and perform inter-processor communication.

The control processor (also known as partition manager) has the same general capability as a processing node, but is specialized to perform administrative functions rather than computational ones. It manages a partition composed of a number of processing nodes and is responsible for scheduling user tasks, allocating resources, and servicing I/O requests for that partition.

The CM-5 supports both the data parallel and message passing models of programming. For the data parallel model, TMC provides the CM Fortran [94], C* [97], and *Lisp [93, 95] programming languages. In the case of CM Fortran, the partition manager executes all the Fortran 77 statements, while the nodes execute all the array extensions drawn from Fortran 90.

For the message passing model, TMC provides the CMMD library [96]. This library is a collection of routines that permit cooperative message passing among the processing nodes. CMMD supports a version of message passing known as *host/node* programming, where a *host program* runs on the partition manager and independent copies of a *node program* run on each of the processing nodes. The programmer must handle all the load balancing, the distribution of data, the division of work, and the communication among the nodes.

2.2.2 The DEC Alpha Cluster

The second distributed memory machine used in this study is a cluster of DEC Alpha workstations. The cluster at the Northeast Parallel Architectures Center (NPAC) at Syracuse University consists of eight DEC 3000/400 compute servers from Digital Equipment Corporation, each with 62 MB of memory and one DEC DEFTA FDDI interface. The FDDI interfaces connect the servers over fiber to a DEC Gigaswitch, which connects via fiber to a DEC Network Integration Server (DECNIS). This networking provides full FDDI bandwidth and low-latency switching to every processor in the cluster.

2.3 Problem Classes

Problems can be classified into the following broad categories [42]:

1. **Synchronous Problems:** The data associated with these problems have regular geometries, and values of the data evolve synchronously during the computation. An example is the finite difference problem using the Jacobi update strategy.

2. **Loosely Synchronous Problems:** The data associated with these problems have irregular geometries. The computation synchronizes at regular time intervals (after every iteration, for example). Examples include hierarchical N-body simulation and region growing.

3. **Asynchronous Problems:** The data associated with these problems have irregular geometries and the computation does not synchronize at regular time intervals (computation is event-driven). An example is melting in two dimensions.

2.4 High Performance Fortran

High Performance Fortran (HPF) [43, 62, 63, 68] is Fortran 90 augmented by a set of extensions intended to facilitate data parallel programming on a wide range of parallel architectures. The extensions include:

1. *Data Distribution and Alignment Directives*

   These directives advise the compiler on how to distribute data across multiple processors.

2. *New Language Syntax*

   These are extensions to Fortran 90 to better express parallelism in a program. They include the FORALL statement which is a more flexible form of array assignment.

3. *Library Procedures*

   HPF provides a standard interface to a library of useful procedures. These include procedures for data reduction, sorting, matrix calculations, etc.

4. *Extrinsic Procedures*

   HPF allows the use of extrinsic procedures written in another language such as Fortran 77 with message passing. These procedures allow the programmer to handle problems that are not efficiently addressed by HPF.

In HPF both the programmer and the compiler are responsible for specifying parallelism in a program. The programmer provides high level directives for distributing the data, while the compiler generates the low-level parallel code appropriate for the target architecture. The strategy behind HPF is that the user writes a data parallel program which is annotated with data distribution and alignment directives. The compiler then generates code that runs on a target parallel machine. In the case of distributed memory machines, the compiler generates a multi-threaded message passing code with local data and optimized send/receive communications.

Chapter 3

Clustering

Clustering is the process of dividing a set of objects into sensible, useful groupings or clusters. Objects may be persons, species, cities, pixels, documents, vertices of a graph, magnetic domains in a ferromagnet, liquid droplets in a gas, monomers or polymers in solution, conducting domains in a superconductor, etc. The goal is to group these objects into clusters such that the objects within a cluster have a high degree of "natural association" among them, while the clusters are "relatively distinct" from each other. Other names used synonymously with clustering are *classification*, *numerical taxonomy*, *botryology*, and *systematics* [2, 1, 7, 13, 11, 12, 14, 32, 33, 36, 49, 50, 54, 52, 59, 64, 65, 70, 73, 84, 86, 103, 89, 102, 109, 111].

3.1 Definition of Clustering

The clustering problem can be stated as follows: Given a data set of N objects and a set of M variables or attributes describing each of the objects, the aim is to partition the N objects into groups or clusters such that the objects within a group are as "similar" to each other as possible, while the "similarity" between objects of different groups is rather low.

The input to the clustering problem can be represented by an $N \times M$ array of values $x_{i,j}$, where $x_{i,j}$ denotes the value of the j'th variable describing the i'th object, as shown in Figure 3.1.

Values of variables for 1st object:	$x_{1,1}$	$x_{1,2}$	$\cdots$	$x_{1,M}$
Values of variables for 2nd object:	$x_{2,1}$	$x_{2,2}$	$\cdots$	$x_{2,M}$
	$\vdots$	$\vdots$		$\vdots$
Values of variables for N'th object:	$x_{N,1}$	$x_{N,2}$	$\cdots$	$x_{N,M}$

Figure 3.1: Input to the Clustering Problem

The clustering problem is an optimization problem, since the aim is to find the "best" partition or classification of a set of N objects that satisfies all the given criteria and constraints. By "best" we mean a clustering that produces the smallest number of clusters possible (or the largest-size clusters possible). However, the solution of a given clustering problem may not not be unique, since more than one partitioning of the objects could satisfy the criteria and constraints and produce the smallest number of clusters possible.

The criteria for identifying clusters depend on how the investigator chooses to give meaning to the terms "natural association" and "relatively distinct". Consequently, clustering requires that a measure of *dissimilarity* (or, equivalently, a measure of *similarity*) be established between pairs of objects.

Suppose $S = \{O_1, O_2, \cdots, O_N\}$ is a set of N objects, each described by a set of M variables or attributes. A mapping $d : S \times S \rightarrow [0, \infty)$ is a *dissimilarity function* if, for any $1 \leq i, j \leq N$, it satisfies the following three conditions:

$$d(O_i, O_i) = 0,$$

$$d(O_i, O_j) = d(O_j, O_i),$$

and

$$d(O_i, O_j) \geq 0.$$

There are many different possible ways of establishing a measure of dissimilarity (or similarity) between objects, and the choice between them is subjective. Sneath and Sokal divided the various measures into four groups: distance measures, association coefficients, correlation coefficients, and probabilistic similarity coefficients [89].

One common measure of dissimilarity between a pair of objects is the *Euclidean distance* between them. Two objects are said to be more similar, the smaller the Euclidean distance between them. They become dissimilar if the distance between them is larger than a given threshold value T. The objects can be represented as points in an M-dimensional metric space, where each dimension represents a variable or attribute and the coordinates of a point represent the values of the M variables or attributes associated with that point. If A and B are two points with coordinates $(a_1, \cdots, a_M)$ and $(b_1, \cdots, b_M)$, respectively, then the Euclidean distance, D, between A and B is given by:

$$D(A, B) \quad = \quad \left[\ \sum_{i=1}^{M} (a_i - b_i)^2 \ \right]^{\frac{1}{2}}.$$

In one, two, or three dimensions, this is just the "straight line" distance between points A and B.

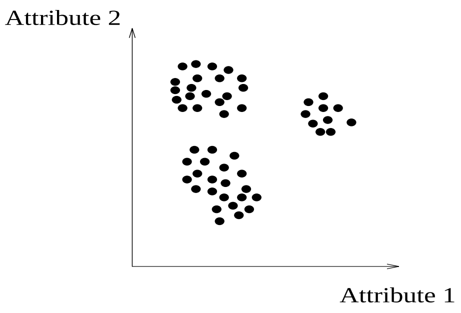
When the variables are measured in different units, it may be necessary to prescale the variables to make their values comparable, or equivalently, to compute a *weighted Euclidean distance*:

$$D_W(A,B) \quad = \quad \left[\sum_{i=1}^{M} W_i \, (a_i - b_i)^2 \right]^{\frac{1}{2}},$$

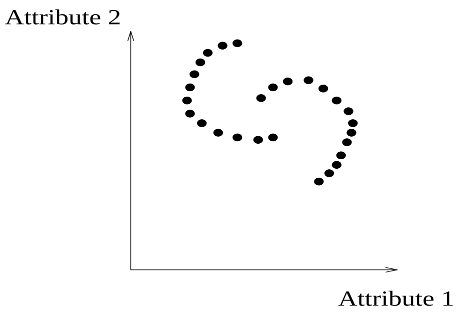
where W_i is the weight applied to variable i. A discussion of the choice of weights W can be found in [52, 65].

Figure 3.2 (a)–(c) gives a geometrical illustration of clustering. Each point in the figure represents an object, and the coordinates of the point represent its attributes. In general, there will be more than two variables describing each object and the clusters will not be visually recognizable as in the configurations shown in Figure 3.2.

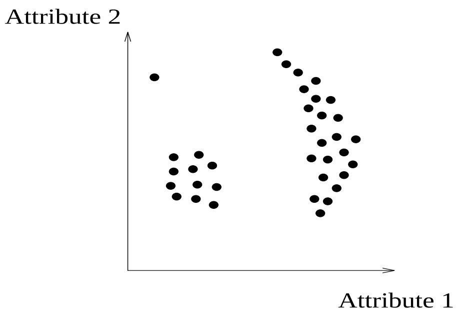
(a)



(b)



(c)



(a) Data set with three clusters; (b) Data set with two clusters; (c) Data set with three clusters, where one of the clusters is composed of a single isolated object

Figure 3.2: A Geometrical Illustration of Clustering

3.2 Example: Sightings of Minor Planets

The classification of minor planet sightings is an example of a typical clustering application [52]. There is a very large number of minor planets (or asteroids) in orbits between Mars and Jupiter. In a photograph against the fixed stars, a minor planet sighting will appear as a curved streak from which its orbital elements may be computed.

There are many thousands of sightings of minor planets. An important problem is deciding which sightings are of the same planet. The objects are the sightings. Two objects are similar if, considering measurement error, the sightings could plausibly be of the same planet. A cluster is a group of sightings of the same planet.

Table 3.1 consists of a number of observations on minor planets. Each sighting is labeled by the year of the sighting and the initials of the astronomer, and is described by the two variables *Node* and *Inclination*. *Node* denotes the angle in the plane of the earth's orbit at which the minor planet crosses the plane of the earth's orbit, while *Inclination* denotes the angle between the plane of the earth's orbit and the plane of the planet's orbit.

Figure 3.3 illustrates the clustering of sightings of minor planets. Each point in the figure represents a sighting, and the coordinates of the point represent its two variables, *Node* and *Inclination*. All sightings that are close enough are clustered together and are assumed to be of the same planet. The clusters are delineated by dashed lines.

Table 3.1: Sightings of Minor Planets

Minor Planet Sighting	**Node** (degrees)	**Inclination**(degrees)
1924TZ	59.9	5.7
1931DQ	69.6	4.7
1940YL	338.333	16.773
1941FD	132.2	4.7
1949HM	339.625	16.076
1952DA	55.144	4.542
1955QT	130.07	4.79

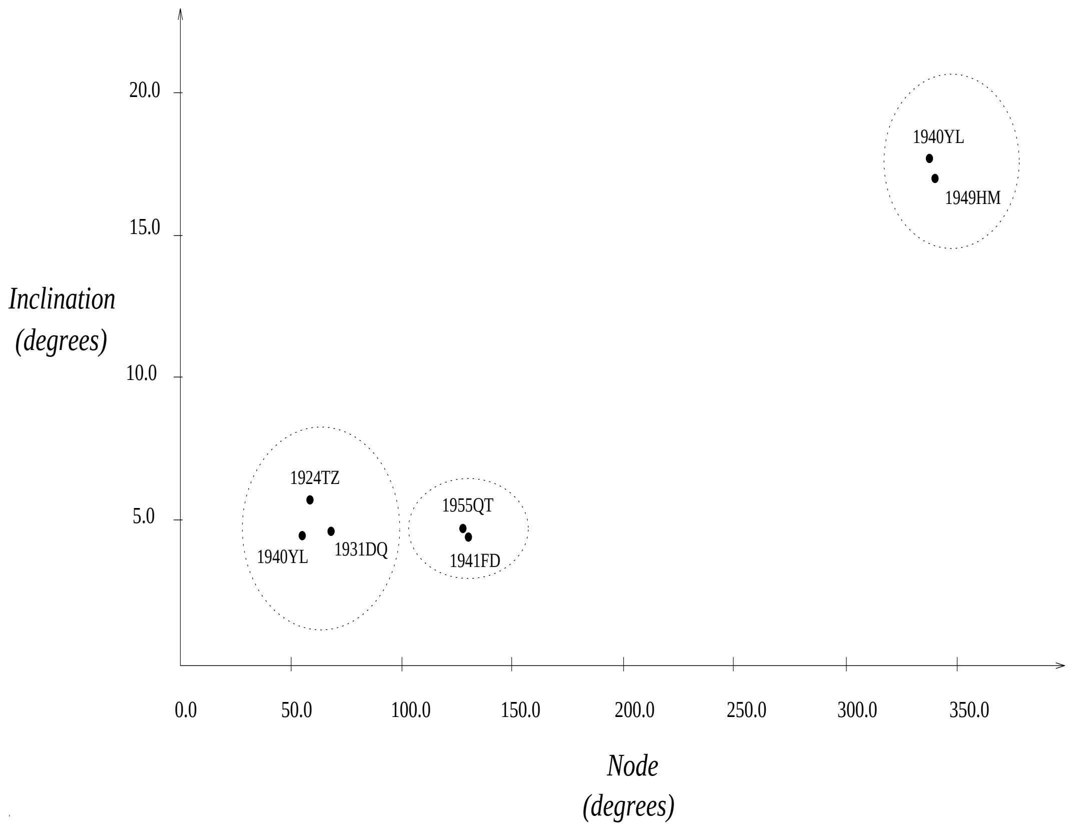


Figure 3.3: Clusters of Minor Planet Sightings

3.3 Methods of Clustering

There are many competing philosophies as to how groups should be constructed and how they should be defined. Consequently, a wide variety of logical, statistical, mathematical, and heuristic methods have been applied to the problem of creating groups. According to Blashfield [12], most clustering methods fall into two categories: (a) hierarchical methods, and (b) partitional methods.

3.3.1 Hierarchical Methods

Hierarchical methods of clustering have been by far the most commonly used ones [12], and are the focus of this dissertation. These methods proceed to build clusters gradually by putting the most similar entities together and producing a nested sequence or hierarchy of partitions that can be represented by trees or dendrograms. Figure 3.4 shows an example of a dendrogram.

Hierarchical methods are either *agglomerative* or *divisive*. Agglomerative hierarchical clustering places each object in its own cluster and gradually merges these atomic clusters into larger and larger clusters. A divisive hierarchical clustering, on the other hand, reverses the process by starting with all objects in one cluster and subdividing into smaller pieces.

When using an agglomerative hierarchical method, different criteria can be used to merge objects together into the same group. The two most commonly used criteria are single-linkage and complete-linkage:

1. **Single-linkage criterion.** In single-linkage, two objects are merged into the same cluster if the dissimilarity between them is smaller than or equal to a certain threshold value T that is defined by the researcher. A single object O_i

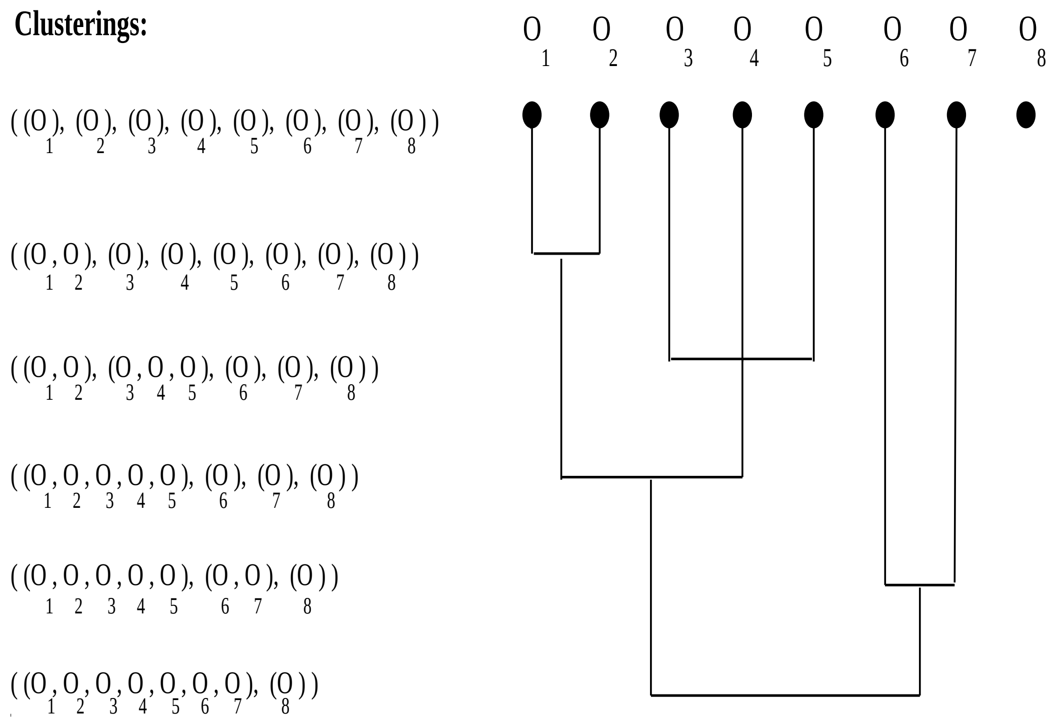


Figure 3.4: An Example of a Dendrogram

is attached to an already existing cluster C_k if there exists an object $O_j \in C_k$ such that the dissimilarity $d(O_i, O_j)$ between O_i and O_j is $\leq T$.

2. **Complete-linkage criterion**. In complete-linkage, two objects are merged into the same cluster if the dissimilarity between them is smaller than or equal to the threshold value T. A single object O_i is attached to an already existing cluster C_k if for every object $O_j \in C_k$, $d(O_i, O_j) \leq T$.

An algorithm applying the single-linkage criterion may proceed by joining together the two closest objects to form a cluster (the dissimilarity between the objects must be $\leq T$). It then joins together the next two closest objects to form a cluster, and so on. If the two objects to be joined lie in different clusters obtained in previous steps, then the two clusters are joined instead. The algorithm terminates when no more objects can be joined together, or when all the objects are in the same cluster. In complete-linkage, on the other hand, it is not sufficient to examine the dissimilarity between two objects to determine whether they can be in the same cluster, but all the objects in the two clusters to which these two objects belong must be examined. Two clusters are joined if the dissimilarity between *every* pair of objects in the two clusters is $\leq T$.

The single-linkage criterion produces clusters that are isolated from each other, paying no attention to their internal cohesion, while the complete-linkage criterion produces clusters that have strong internal cohesion. Figure 3.5 (a)–(c) compares clusters obtained using the single-linkage and complete-linkage criteria for a given threshold T. The dissimilarity measure between two points is taken to be the Euclidean distance between them. Clusters in the figure are delineated by dashed lines.

Many other clustering criteria take a middle road between single-linkage and complete-linkage. For example, the *average-linkage* criterion characterizes a cluster by the average of all dissimilarities (or similarities) within it; that average must be $\leq$ a given threshold T [89].

3.3.2 Partitional Methods

Partitional methods of clustering generally start with an initial partition of the data set of objects into some specified number of clusters. Then the boundaries of these clusters are iteratively refined and modified until the required clustering is obtained. No nested sequence or hierarchy of partitions is produced in the process. Several clustering methods belong to this category; three of them are listed below [49].

1. **Optimization methods**. These methods require a *clustering criterion* to be optimized. A commonly used criterion is the within-groups sums of squares. If $S = \{O_1, O_2, \cdots, O_N\}$ is the set of objects, $C = \{C_1, C_2, \cdots, C_P\}$ is a partition of S into P clusters, and $d(O_i, O_j)$ is the dissimilarity between objects O_i and O_j, then the quality function $g(C)$ is defined by

   $$g(C) = \frac{1}{2} \sum_{k=1}^{P} \frac{1}{n_k} \sum_{O_i \in C_k} \sum_{O_j \in C_k} d(O_i, O_j)^2,$$

   where n_k denotes the number of objects in cluster k.

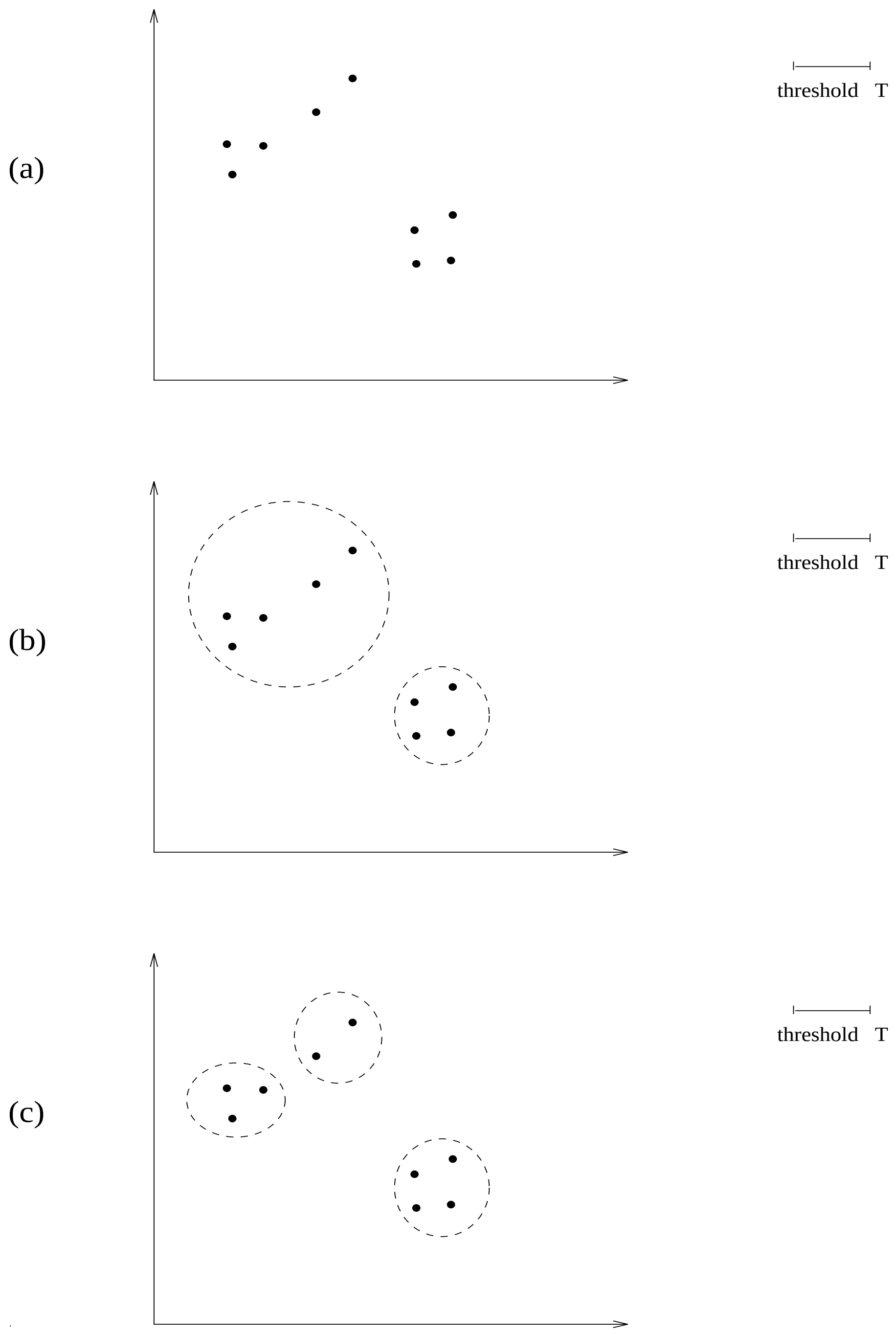
   A partition C that minimizes $g(C)$ is taken as optimum. However, every quality function would take its optimum when the objects are classified into N one-element clusters, since $g(C) = 0$ always holds for $C = \{\{O_1\}, \{O_2\}, \cdots \{O_N\}\}$. Therefore, to prevent these methods from producing such trivial results, the number of clusters must be defined in advance or some constraints must be imposed on the clusters.

   The most straightforward way to discover the optimum partition C is to form all possible partitions and choose the one that minimizes the quality function. Unfortunately, the number of partitions that must be examined by this simple approach may be enormous. Heuristic procedures, which sample a small subset of the possible partitions, are therefore commonly used. Examples of this method include [85, 86].

2. **Constructive Methods**. These methods start from an arbitrarily chosen object O_i of the data set, which is believed to be a representative of a group.

This object is the so-called *kernel* of the cluster. All elements nearest to the kernel are attached to its group. This process stops if the cluster becomes too heterogeneous. Another element from the remainder of the data set is chosen as the kernel of the next cluster, and the process is repeated until all objects are classified.

3. **Analysis of Density Methods**. If objects are represented as points in space, then "clusters" are the connected areas of high point densities. The maxima of the densities, or *modes*, define cluster centers. Under ideal conditions, clusters are separated by areas of low densities called *valleys* of the distribution. Methods in this category detect the clusters by employing valley seeking techniques or mode seeking techniques.



(a) Objects represented as points in the plane; (b) Clusters determined using single-linkage criterion ; (c) Clusters determined using complete-linkage criterion

Figure 3.5: Single-Linkage vs. Complete-Linkage Clustering Criterion

Chapter 4

A Graph-Theoretic Formulation of Hierarchical Clustering

Many real-world situations and relationships may conveniently be described or modeled by means of a diagram consisting of a set of points, together with lines joining certain pairs of points. The points represent physical objects and the lines represent relations among these objects. For example, the points could represent communication centers and the lines could represent communication links between these centers. Such a diagram, known as a *graph*, has proved to be a valuable tool in modeling these situations [17].

One advantage of using graphs is that the researcher can rely on a large number of mathematical theorems and results to gain insight into the structure of a real system. Moreover, graph theory provides researchers of different disciplines with a single mathematical language.

Many graph-theoretic formulations of clustering have been proposed. These include removing inconsistent edges of a minimum spanning tree of a graph [111]; finding maximal connected components or cliques of a graph [64]; constructing one or more directed trees that correspond to clusters [70]; constructing graphs based on limited neighborhood sets [102]; and removing arcs from a graph such that the largest inter-subgraph flow is minimized [109].

This dissertation presents a graph-theoretic formulation of clustering based on the *multigraph* concept [49]. This formulation can be applied to a wide variety of clustering applications and is useful for cases when the dissimilarity between any two objects of the data set cannot be measured by a single numeric value. Moreover, the formulation allows the researcher to employ different clustering criteria for different attributes of the objects.

4.1 Graphs and Multigraphs

Formally, a graph G (also called an *undirected graph*) is defined to be a pair (V, E), where V is a non-empty finite set of vertices and E is a finite set of *unordered* pairs of elements of V. If pairs of elements in the set of edges E are *ordered*, then the graph G is called a *directed graph*.

Often, it is useful to attach weights to the edges. For example, if the vertices are points in space we can define the weight of the edge between two points to be the Euclidean distance between them. We thus get an *edge-weighted graph* (or simply a *weighted graph*). A *subgraph* of a graph G is a graph whose vertices are a subset of the vertices of G and whose edges are a subset of the edges of G.

A *path* from vertex v_i to vertex v_j in the graph is a list of vertices, starting with v_i and ending with v_j, in which successive vertices are connected by edges in the graph. A graph G is *connected* if there is a path from every vertex to every other vertex in the graph. Intuitively, if the vertices of G were physical objects and the edges were strings connecting them, then a connected graph would stay in one piece if picked up by any vertex. A graph that is not connected can be split up into a number of maximal connected subgraphs called *connected components.* A graph is called *complete* if every pair of its vertices is directly linked by an edge, i.e., if all possible edges are present. A *clique* is a complete subgraph [87]. For a detailed discussion of graph theory, see [17, 35, 92, 99, 100, 101, 107].

Hierarchical clustering can naturally be formulated using graph theory. The objects $O_1, O_2, \cdots, O_N$ of a data set can be represented by vertices $v_1, v_2, \cdots, v_N$ of a graph. An edge links two vertices v_i and v_j of the graph if the dissimilarity $d(O_i, O_j)$ between the objects O_i and O_j does not exceed a given threshold T. The weight of the edge between v_i and v_j is taken to be equal to $d(O_i, O_j)$. The clusters are found by iteratively grouping together the nodes of the graph to form larger and larger clusters. Using this interpretation, the single-linkage clusters are simply the *connected components* of the graph, while the complete-linkage clusters are the *maximal cliques* of the graph.

However, in many cases of multi-dimensional data (i.e., data described by more than one variable) where the scale levels of the attributes vary considerably, the dissimilarity between any two objects O_i and O_j of the data set cannot be measured by a *single* numeric value $d(O_i, O_j)$. Moreover, the researcher may wish to employ different clustering criteria for different attributes. For example, the researcher may wish to use the single-linkage clustering criterion for one attribute, while using the complete-linkage criterion for another attribute.

In such cases, Godehardt [49] proposes using a special kind of graph, called a *multigraph*, to describe clustering. A multigraph is a graph in which a pair of vertices can be connected by *more than one edge.* Multigraphs are useful in describing several distinct relations on the same set of vertices. Pictorially, each relation is represented by a "layer" of edges. Figure 4.1 (a)–(c) illustrates the concept of a multigraph. The two graphs (a) and (b) on the same set of vertices are combined to form the multigraph (c). The vertices of the multigraph are elongated to show clearly the two layers of edges.

The following definitions are derived from those by Godehardt [49]. Formally, *an M-layer multigraph*, Γ_M, is defined to be an $M+1$-tuple $(V,\ E_1, \cdots, E_M)$, where V is a non-empty finite set of vertices and E_l, $1 \leq l \leq M$, is a finite set of elements of the form (v_i, v_j, l), where v_i and v_j are elements of V. So, E_l denotes the set of edges in layer l of the multigraph.

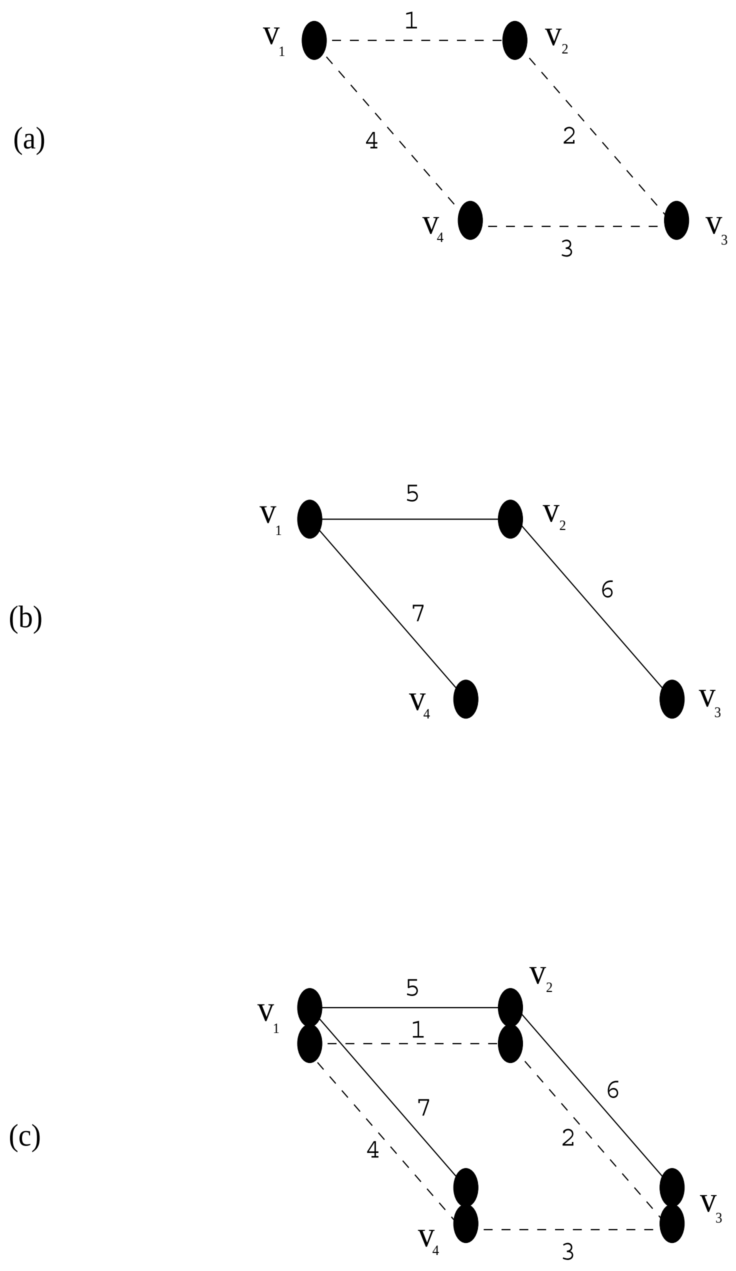
If vertices v_i and v_j of a multigraph are linked by at least k edges (each edge lies in a different layer), then the vertices are said to be *k-fold connected*, and the k edges are called a *k-fold connection* between v_i and v_j. A *sub-multigraph* of a multigraph Γ_M is a multigraph whose vertices are a subset of the vertices of Γ_M and whose edges are a subset of the edges of Γ_M. A *k-sub-multigraph* is a sub-multigraph all of whose vertices have k-fold connections between them.

A *k-path* from vertex v_i to vertex v_j in a multigraph is a list of vertices starting with v_i and ending with v_j in which successive vertices are k-fold connected. A multigraph is *k-connected* if there is a k-path from every vertex to every other vertex in the multigraph. A multigraph that is not k-connected can be split into a number of maximal k-connected sub-multigraphs called *k-connected components.* A graph is called *k-complete* if every pair of its vertices is directly linked by a k-fold connection. A *k-clique* is a k-complete sub-multigraph.

Because of the flexibility provided by multiple layers of edges, the multigraph concept can easily be applied to clustering when no single measure of dissimilarity can be calculated. Suppose we are given a set of N objects, $O_1, O_2, \cdots, O_N$, and a set of M variables describing each of these objects. Instead of computing a single measure of dissimilarity, $d(O_i, O_j)$, between every pair of objects, O_i and O_j, we compute M "local" dissimilarities, $d_1(O_i, O_j), \cdots, d_M(O_i, O_j)$, corresponding to each of the M variables. Furthermore, we can define M different thresholds for each of the M variables.

Suppose the thresholds defined for the M variables are $T_1, \cdots, T_M$. The clustering problem can be formulated using an M-layer multigraph, where the vertices $v_1, v_2, \cdots, v_N$ of the multigraph correspond to objects $O_1, O_2, \cdots, O_N$ of the data set, and layer l, $1 \leq l \leq M$, of edges describes the dissimilarity between the objects relative to the l'th variable. In a given layer l of the multigraph, an edge links two vertices v_i and v_j if the dissimilarity, $d_l(O_i, O_j)$, between objects O_i and O_j, does not exceed the threshold T_l associated with the l'th variable. The weight of the edge between v_i and v_j in layer l is taken to be equal to $d_l(O_i, O_j)$. Using this multigraph formulation, the single-linkage clusters are the *maximal k-connected components* of the multigraph, while the complete-linkage clusters are the *maximal k-cliques* of the multigraph for a given value k, $1 \leq k \leq M$, that depends on the application and is set by the researcher.

One advantage of the multigraph-based formulation of clustering is that it expresses dissimilarity between objects in a more natural way by allowing a different dissimilarity measure for each variable, instead of using a single measure for all the variables. Moreover, the researcher has an easier task in defining various thresholds for every single variable of the data and in experimenting with them.



(a) First graph on vertices $v_1, \cdots, v_4$;   (b) Second graph on vertices $v_1, \cdots, v_4$;

(c) Multigraph, with "layered" edges, on vertices $v_1, \cdots, v_4$

Figure 4.1: An Example of a Multigraph

4.2 Applicability to Clustering Applications

In this section, we examine the applicability of the multigraph-based formulation to three important clustering applications: cluster labeling in percolation and spin models, binary image segmentation, and region growing. These three applications have been chosen because they are important applications that employ different criteria for clustering.

4.2.1 Cluster Labeling in Percolation and Spin Models

A percolation model (or a "bond" percolation model) in physics is a D-dimensional lattice of sites ($D \geq 1$), where certain pairs of the sites have bonds or links between them. The existence of bonds is governed by a random mechanism, the details of which depend on the context in which the model is used. The word "random" is to be understood in the mathematical sense that a bond between two sites occurs with a given probability p. The sites of the lattice may represent the molecules in a gas, trees in a forest, cells in an organism, etc. [34, 90].

A path is said to exist between two sites A and B of the lattice if a sequence of sites may be found, beginning with A and ending with B, such that successive sites in the sequence have a bond between them. There may be many paths between a given pair of sites; but if there is at least one path, then the sites are said to be connected. The presence of a path may, for example, allow the flow of liquid or electrical charge between the sites it connects, the spread of fire from one tree to another in a forest, or the passage of a telephone message from one point to another.

The sites of the lattice are to be partitioned into clusters such that pairs of sites in the same cluster are connected and there is no path between sites in different clusters. The cluster sizes increase with the number of bonds in the lattice, which in turn increase with the value chosen for the probability p. At some critical probability value p_c, the system is said to be in a *percolating state*. For example, in the case of a forest model, at the critical probability p_c a fire started in one edge of the forest would spread to the opposite edge.

The transition from a non-percolating state to a percolating state is a kind of phase transition. The study of the distribution of the sizes and shapes of the clusters aims to achieve an understanding of this transition as well as the general theory of phase transitions and critical phenomena. A more complete discussion of percolation models can be found in [34, 90].

The identification or labeling of clusters in a percolation model can be expressed as a clustering problem, where the objects are the sites in the lattice and each object is described by the following variables:

1. The position of the site in the lattice.

2. Whether a bond exists between the site and each of its neighbors in the lattice.

The sites are to be grouped into maximal disjoint clusters that satisfy the following connectivity requirement:

> **Connectivity Requirement:** The sites in a cluster must be connected. That is, whenever two sites O_i and O_j belong to the same cluster, then there must exist a sequence of sites

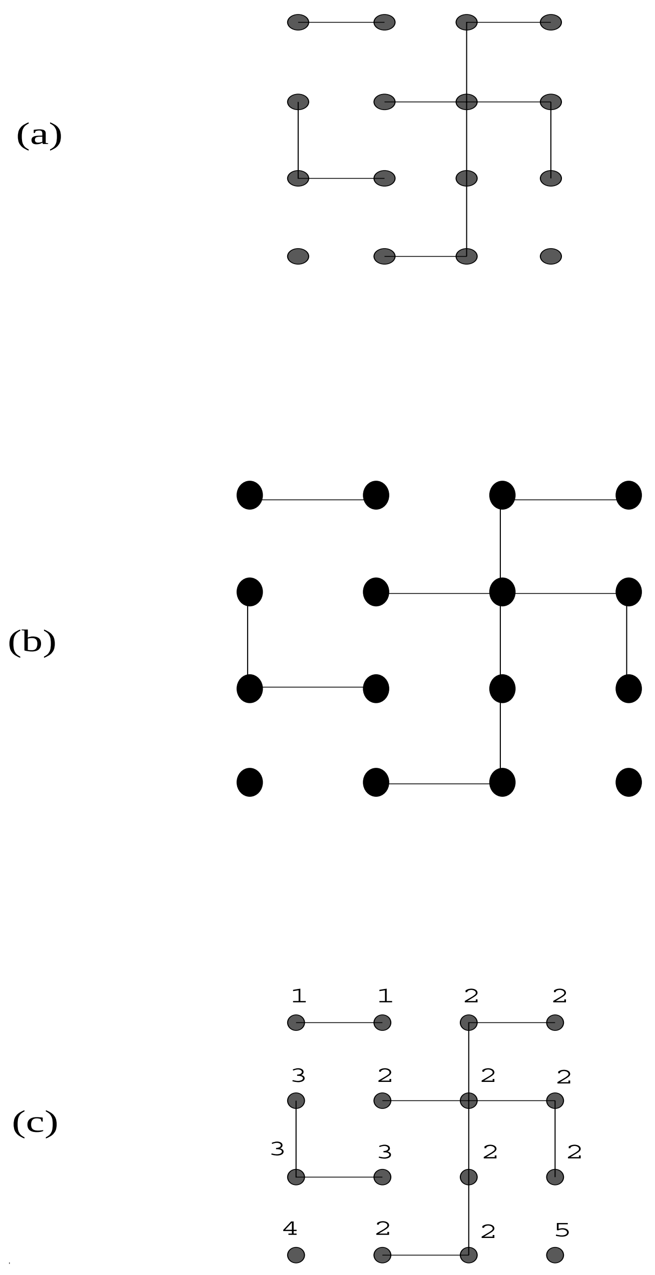
beginning with O_i and ending with O_j such that successive sites in the sequence belong to the cluster and have a bond between them.

Given the above requirement, the measure of dissimilarity, d, between two objects (or sites), O_i and O_j, can be defined as:

$$d(O_i, O_j) = \begin{cases} 0, \text{ if a bond exists between the two sites } O_i and O_j \\ \\ 1, \text{ otherwise,} \end{cases}$$

and the threshold value T can be taken to be equal to 0.

The problem can be represented by an undirected, weighted graph (or a one-layer multigraph) where the vertices of the graph represent the lattice sites, and an edge exists between two vertices of the graph if the dissimilarity between them is ≤ 0, i.e., if there exists a bond between them in the lattice. All the weights of the edges are identical and are equal to some arbitrary constant, say 1. The single-linkage criterion is used for clustering, since it satisfies the requirements of the problem. Thus the clusters in the lattice are precisely the connected components of the graph, and the clustering problem for percolation models is essentially the problem of labeling connected components of an undirected graph. Figure 4.2 (a)–(c) shows an example of cluster labeling in percolation models where five clusters are identified in the lattice.



(a) Sites in a lattice, with bonds between some sites; (b) Graph representing clustering problem; (c) Clusters identified in the lattice, labeled by the integers 1 – 5

Figure 4.2: Cluster Labeling in Percolation Models

Another closely related clustering application in physics is cluster labeling in spin models of real systems such as magnets. The goal of computer simulations of spin models is to generate configurations of spins typical of statistical equilibrium and to measure physical quantities on this ensemble of configurations. The generation of configurations is traditionally performed by Monte Carlo algorithms such as the one by Metropolis et al [74]. However, improved algorithms have been designed in which clusters of spins, rather than individual spins, are updated at each step of the Monte Carlo algorithm.

In the Swendsen and Wang algorithm [91], clusters of spins are created by introducing bonds between neighboring sites with a certain probability. Sites that are connected together by a sequence of one or more bonds are identified and assigned a unique cluster label. Each cluster is updated by choosing a random new spin value for that entire cluster and assigning it to all the spins in that cluster. In order to implement these algorithms, a method for identifying and labeling the clusters of connected sites is required. The clustering in this application, like that in percolation models, is essentially the problem of labeling connected components of an undirected graph.

A more detailed description of spin models and cluster labeling algorithms can be found in [8, 10, 28, 44, 69, 75].

4.2.2 Binary Image Segmentation

Image segmentation is an important initial step in many image-understanding applications, such as the recognition of objects in robot vision, as well as the study of satellite pictures for the identification of various types of soils, ocean currents, storms, etc. [3, 6, 23, 45, 57, 79, 110].

The problem of image segmentation can be stated as follows: We are given a digital image composed of a two-dimensional array of pixels or pixel elements. Associated with each pixel is an intensity value. For binary images, the intensity value is either 0 (white) or 1 (black); for grey-scale images, the value is an integer commonly between 0 and 255. The goal is to partition the image into disjoint (non-overlapping) regions or segments such that pixels belonging to a region are more similar to each other than pixels belonging to different regions. A region should consist of contiguous or connected pixels.

In this study we assume 4-connectivity of pixels. That is, a pixel in the image has at most four neighbors located to its north, south, east, and west. (If a pixel is on the border of an image, then it has two or three neighbors.) Figure 4.3 shows a 5×5 pixel image, where the four neighbors of the pixel in the center are labeled.

In the case of binary images, the problem of image segmentation simplifies to assigning a unique label to a contiguous collection of pixels that have the *same* intensity value. The problem can be expressed as a clustering problem, where the objects to be clustered are the pixels in the image, and each object is described by

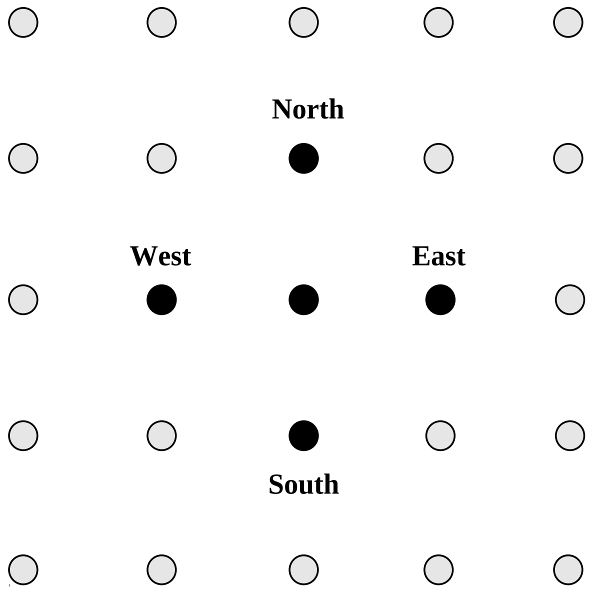


Figure 4.3: Four-Connectivity of Pixels

the following two variables:

1. The position of the pixel (row and column numbers)

2. The intensity value of the pixel

The pixels are to be grouped into maximal disjoint clusters which satisfy both of the following two requirements:

> **Connectivity Requirement:** The pixels in a cluster must be connected. That is, whenever two sites O_i and O_j belong to the same cluster, then there must exist a sequence of sites beginning with O_i and ending with O_j such that successive

sites in the sequence belong to the cluster and are adjacent in the image.

Homogeneity Requirement: All pixels in a cluster must have the same intensity value (0 or 1).

Given the above requirements, the measure of dissimilarity, d, between two objects (or pixels), O_i and O_j, can be defined as follows:

$$d(O_i, O_j) = \begin{cases} 0, \text{ if } O_i \text{ and } O_j \text{ are adjacent in the pixel image} \\ \text{and have the same intensity value} \\ \\ 1, \text{ otherwise} \end{cases}$$

and the threshold value T can be taken to be equal to 0.

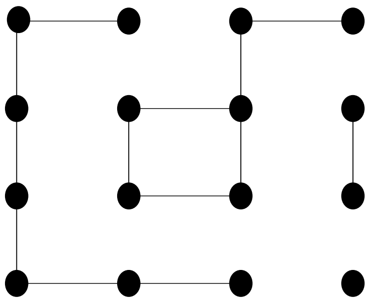
The problem can be represented by an undirected, weighted graph (or a one-layer multigraph) where the vertices of the graph represent the pixels and an edge exists between two vertices of the graph if the dissimilarity between them is ≤ 0, i.e., if the corresponding pixels in the image are adjacent and have the same intensity value. All the weights of the edges are identical and are equal to some arbitrary constant, say 1. The single-linkage criterion is used for clustering, since it satisfies the requirements of the problem. Thus the regions in the image are precisely the connected components of the graph, and binary image segmentation is essentially the problem of labeling connected components of an undirected graph.

Figure 4.4 (a)–(c) shows an example of binary image segmentation, where four clusters are identified in the image.

(a)

1	1	0	0
1	0	0	1
1	0	0	1
1	1	1	0

(b)



(c)

1	1	2	2
1	2	2	3
1	2	2	3
1	1	1	4

(a) A binary image; (b) Graph representing clustering problem (4-connectivity of pixels is assumed); (c) Clusters identified in the image, labeled by the integers 1 – 4

Figure 4.4: Clustering in Binary Image Segmentation

4.2.3 Region Growing

Region growing is a general technique for the segmentation of a grey-scale image, where the pixel intensities belong to a range of values (commonly between 0 and 255). Image characteristics are used to group or cluster adjacent pixels together to form regions. Regions are then merged with other regions to "grow" larger regions. A region corresponds to a real-world object or to a meaningful part of one.

The merging of regions is usually governed by a homogeneity criterion that must be satisfied. One homogeneity criterion, known as the *pixel range* homogeneity criterion, requires that the difference between the minimum and maximum intensities of pixels within a region not exceed a given threshold value T.

Region growing can be expressed as a clustering problem, where the objects to be clustered are the pixels in the image and each object is described by the following two variables:

1. The position of the pixel (row and column numbers)
2. The intensity of the pixel (a value between 0 and 255)

The pixels are to be grouped into maximal disjoint clusters which satisfy both of the following two requirements:

> **Connectivity Requirement:** The pixels in a cluster must be connected. That is, whenever two sites O_i and O_j belong to the same cluster, then there must exist a sequence of sites beginning with O_i and ending with O_j such that successive sites in the sequence belong to the cluster and are adjacent in the image.

Homogeneity Requirement: The difference between the minimum and maximum intensities of pixels within a cluster must not exceed the given threshold value T.

The measure of dissimilarity, d, between two objects (or pixels) O_i and O_j, can be defined as follows:

$$d(O_i, O_j) = \begin{cases} 0, \text{ if } O_i \text{ and } O_j \text{ are adjacent in the pixel image} \\ \text{and the difference between their intensities is } \leq T \\ \\ 1, \text{ otherwise.} \end{cases}$$

At first, we may attempt to represent the problem by an undirected, weighted graph (or a one-layer multigraph), where the vertices of the graph represent the pixels and an edge exists between two vertices if the corresponding pixels are adjacent in the image and the difference between their intensities is $\leq T$. We take the weight of an edge to be equal to the difference in intensities of the two pixels joined by that edge. However, we observe that this graph representation poses a difficulty: The single-linkage criterion cannot solely be used for clustering, since it does not guarantee the satisfaction of the homogeneity requirement. Moreover, the complete-linkage criterion cannot solely be used for clustering, since it only yields regions composed of at most two pixels.

The multigraph concept proves to be a convenient tool for formulating the problem and for overcoming this difficulty. A two-layered multigraph can be used to represent the problem, where the vertices of the multigraph represent the pixels in the image. The first layer of edges reflects the adjacency relationship between the pixels, while the second layer represents the differences in intensities between the pixels.

In the first layer we define the measure of dissimilarity, d_1, between two objects O_i and O_j to be:

$$d_1(O_i, O_j) = \begin{cases} 0, \text{ if } O_i \text{ and } O_j \text{ are adjacent in the pixel image} \\ \\ 1, \text{ otherwise,} \end{cases}$$

and we take the threshold value T_1 to be equal to 0. All the weights of the edges are identical and are equal to some arbitrary constant, say 1.

In the second layer we define the measure of dissimilarity, d_2, between two objects O_i and O_j to be

$$d_2(O_i, O_j) = \text{the difference between the intensities of } O_i \text{ and } O_j,$$

and we take the threshold value T_2 to be equal to the given threshold value T. An edge exists between two vertices in the second layer if the difference between the intensities of the two pixels is $\leq$ the given threshold T. The weights of the edges are taken to be equal to this difference.

The single-linkage criterion in the first layer of the multigraph satisfies the connectivity requirement of the problem, since connected components in the first layer are comprised of adjacent pixels. On the other hand, the complete-linkage criterion in the second layer of the multigraph satisfies the homogeneity requirement, since a clique in the second layer guarantees that the difference between the intensities of any two pixels does not exceed the threshold T. Combining these two criteria, we get:

The regions in the image correspond to the maximal, disjoint sub-multigraphs whose vertices belong to a connected subgraph in the *first* layer of the multigraph and to a clique in the *second* layer of the multigraph.

It follows that a region is the *intersection* of a connected subgraph in the first layer of the multigraph and a clique in the second layer of the multigraph.

Figure 4.5 shows an example of clustering in region growing, where four clusters are identified in the image.

4.3 Complexity

Garey and Johnson [46] provide the following definition of the decision problem of clustering. (A decision problem is one whose solution is either "yes" or "no".) This definition assumes a single dissimilarity (distance) measure between pairs of objects and the complete-linkage criterion for clustering.

INSTANCE: Finite set of objects X, a distance $d(x,y) \in Z_0^+$ for each pair $x, y \in X$, and two positive integers K and B.

QUESTION: Is there a partition of X into K disjoint sets $X_1, X_2, \cdots, X_K$, such that, for $1 \leq i \leq K$ and all pairs $x, y \in X_i$, $d(x,y) \leq B$?

According to Garey and Johnson [46], the above problem remains NP-complete even for fixed $K = 3$. It is solvable in polynomial time for $K = 2$. Variants in which we ask that the sum, over all X_i, of $max\{d(x,y) : x, y \in X_i\}$ or of $\sum_{x,y \in X_i} d(x,y)$ be at most B are also NP-complete.

A similar complexity result is obtained from the corresponding multigraph formulation. This is composed of a one-layer multigraph where the vertices represent the objects and an edge joins two objects if the distance between them is $\leq B$. The complete-linkage criterion is used for clustering. The decision problem of clustering in this case is equivalent to the PARTITION INTO CLIQUES problem defined as follows [46]:

INSTANCE: Graph $G = (V, E)$, positive integer $K \leq |V|$

QUESTION: Can the vertices of G be partitioned into $k \leq K$ disjoint sets $V_1, V_2, \cdots, V_k$ such that, for $1 \leq i \leq k$, the subgraph induced by V_i is a complete subgraph?

According to Garey and Johnson [46], the above problem remains NP-complete for all fixed $K \geq 3$. It is solvable in polynomial time for $K \leq 2$.

Thus when the complete-linkage criterion is employed in one or more layers of the multigraph, the corresponding decision problem is NP-complete. It follows that there are no known polynomial-time algorithms for solving the problem, and the time needed is probably an exponential function of the problem size. To reduce the execution time, most computer algorithms employ heuristics to find a "good enough" solution that satisfies all the clustering criteria and constraints, but may not produce the smallest number of clusters [60].

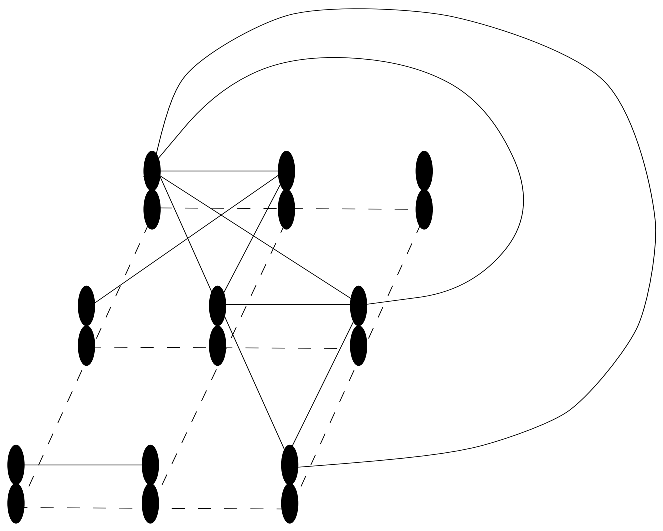
On the other hand, if the single-linkage criterion is employed in every layer of the multigraph, then the problem of clustering is equivalent to finding connected components in a graph and can be solved in polynomial time.

(a)

6	7	9
8	6	5
2	2	5

Threshold = 1

(b)



(c)

1	1	4
2	1	5
3	3	5

(a) A grey-scale image; (b) Graph representing clustering problem; (c) Clusters identified in the image, labeled by the integers 1 – 4

Figure 4.5: Clustering in Region Growing

Chapter 5

A Paradigm for Hierarchical Clustering

A programming paradigm [22], also known as skeleton [30], template [9], archetype [4, 25], or programming model, is a description of a general method for solving a class of problems that have the same control and data-flow structures. A paradigm specifies the overall structure of the computation, while leaving gaps for the definitions of problem-specific procedures and declarations. Familiar paradigms in sequential computation include "divide and conquer", "dynamic programming", and "simulated annealing".

Paradigms are useful for several reasons. First, a paradigm can be customized for the particular problem to be solved, thus reducing the effort required to develop correct and efficient programs. Second, a paradigm allows us to discover and exploit similarities between seemingly different problems. Third, a paradigm can provide the language designer with useful insight about the requirements of certain classes of problems.

5.1 Sequential Hierarchical Clustering Paradigm

The multigraph formulation of hierarchical clustering provides a basis for a programming paradigm that is applicable to a wide variety of hierarchical clustering applications. At the heart of the paradigm is an iterative procedure that clusters or merges vertices of a multigraph together while satisfying all the given criteria and constraints.

The general steps of a sequential, bottom-up hierarchical clustering paradigm can be described as follows:

1. Input the data describing the clustering application.
2. Perform some preprocessing on the data to identify initial clusters. In the simplest case, each data object is considered to be an individual cluster.
3. Construct a multigraph that represents the problem.
4. Iteratively merge vertices of the multigraph together into clusters that satisfy all the given criteria and constraints. The merge continues until no more merges are possible.
5. Output the results.

5.2 Parallel Hierarchical Clustering Paradigm

The general steps of a parallel hierarchical clustering paradigm executing on a distributed memory machine in the host-node programming model can be described as follows:

1. The host processor inputs the data describing the clustering application and distributes it among the node processors.

2. Every node processor performs some preprocessing on its subset of the data to identify initial clusters within the subset. The preprocessing can be performed *in parallel* by all the node processors. In the simplest case, each object is considered to be an individual cluster and is labeled with a unique integer denoting the cluster number.

3. Every node processor constructs a multigraph that represents its subset of the data. The multigraphs can be constructed *in parallel* by all the node processors. The node processors exchange boundary information to form edges between their multigraphs.

4. The node processors cooperate to iteratively merge vertices of the multigraph together into clusters that satisfy all the given criteria and constraints. Often, many cluster pairs can simultaneously merge together without conflicting with each other. The merge continues until no more merges are possible.

5. The host processor collects the results from the node processors and outputs the results.

At a finer level of detail, the steps of the parallel paradigm are as follows:

Step 1: The host processor inputs the data describing the clustering application. Usually, the data consists of the following:

(a) A set of N objects $O_1, \cdots, O_N$.

(b) The values of M variables or attributes describing each object.

(c) A set of M threshold values, $T_1, \cdots, T_M$, associated with each of the M variables.

(d) The clustering criterion to be used for each of the M variables (for example, single-linkage or complete-linkage).

Next, the host processor distributes the data among the node processors such that each processor receives a subset of the objects, the values of the M variables associated with that subset, and the clustering criterion to be used with each variable. Also, each processor receives a copy of the M threshold values.

Step 2: Every node processor performs some preprocessing on its subset of the data to identify initial clusters of objects. The aim of this step is to reduce the overall execution time of the program by using a fast procedure to identify initial clusters. In the simplest case, each object is considered to be an individual cluster and is labeled with a unique integer denoting the cluster number.

Step 3: Every node processor creates an M-layer multigraph representing its subset of the data. The vertices of the multigraph correspond to the objects (or initial clusters) belonging to the processor, and layer l, $1 \leq l \leq M$, of edges describes the dissimilarity between the objects (or initial clusters) relative to the l'th variable. In a given layer l of the multigraph, an edge links two vertices v_i and v_j if the dissimilarity $d_l(O_i, O_j)$ between objects (or initial clusters) O_i and O_j does not exceed the threshold T_l for the l'th variable. The weight of the edge between v_i and v_j in layer l is taken to be equal to

$d_l(O_i, O_j)$.

Next, the node processors exchange boundary information to form the edges between their multigraphs. When an edge joins two vertices in two different processors, only one of the two processors retains a copy of that edge.

Step 4: Every cluster i, $1 \leq i \leq$ number of clusters determined so far, examines all clusters that are adjacent to it in layer 1 of the multigraph and determines which of these clusters it can merge with, while simultaneously satisfying the clustering criteria of layers $1, 2, 3, \cdots, M$. The clusters that satisfy the clustering criterion of every layer are candidates for merging with cluster i. If a neighboring cluster lies in a different processor, then processor communication is required.

Every cluster i selects for merging the candidate cluster determined in the previous step that yields the "best merge" (the "best merge" is application dependent). For example, cluster i may select the candidate cluster that minimizes some function of the M threshold values. The node processors communicate to determine which clusters have selected each other for merging, and which pairs of clusters will *actually merge*.

Step 5: Once two clusters merge, the vertices belonging to the two clusters are labeled by the same cluster number (or, equivalently, the corresponding vertices of the multigraph are coalesced together into one larger super-vertex and the edges in every layer of the multigraph are updated to reflect the new relationships between the vertices).

Step 6: If more merges are possible, then go to step 4. Otherwise, continue on to step 7.

Step 7: The vertices of the multigraph that are labeled by the same cluster number (or, equivalently, the remaining super-vertices of the multigraph) belong to the same cluster. Output the results and terminate.

5.3 High-Level Routines for Hierarchical Clustering

We identify the following high-level routines that implement the various steps of the hierarchical clustering paradigm:

1. **CLUSTER**: This is the main clustering program that calls other required routines.
2. **PREPROCESS**: This routine implements Step 2 of the detailed clustering paradigm. It preprocesses the input data to identify initial clusters.
3. **CONSTRUCT_GRAPH**: This routine implements Step 3 of the detailed clustering paradigm. It constructs the graph that models the application by calling the following two routines:
   (a) **INITIALIZE_VERTEX_ARRAYS**: This routine builds the data structures associated with the vertices of the graph that models the application.
   (b) **INITIALIZE_EDGE_ARRAYS**: This routine builds the data structures associated with the edges of the graph that models the application.
4. **MERGE_VERTICES**: This routine implements Steps 4 – 6 of the detailed clustering paradigm and is the heart of the computation. It iteratively merges

the nodes of the graph into larger and larger clusters that satisfy the given criteria and constraints. In each iteration of **MERGE_VERTICES**, the following routines are called:

   (a) **MATCH_VERTICES**: This routine implements Step 4 of the detailed clustering paradigm. It determines which pairs of vertices can merge together.

   (b) **UPDATE_GRAPH**: This routine implements Step 5 of the detailed clustering paradigm. It updates the graph to reflect new clusters by calling the following two routines:

      i. **UPDATE_VERTICES**: This routine updates the vertices of the graph after a merge iteration.

      ii. **UPDATE_EDGES**: This routine updates the edges of the graph after a merge iteration.

5. **UPDATE_LABELS**: This routine implements Step 7 of the detailed clustering paradigm. It transfers information about the clusters determined in the graph to the input data set so the results of clustering can be displayed.

Chapter 6

Region Growing

In this chapter we study the application of the multigraph-based paradigm for hierarchical clustering to the region growing problem. Region growing was chosen for two main reasons. First, it is a representative hierarchical clustering application (it is the most general of the three applications discussed in Section 4.2). Second, it uses both the single-linkage and complete-linkage criteria for clustering.

As mentioned earlier, region growing is a general technique for the segmentation of a grey-scale image, where the pixel intensities belong to a range of values (commonly between 0 and 255). The aim is to identify "regions" in the image that correspond to real-world objects or to meaningful parts of real-world objects. Image characteristics are used to group adjacent pixels together to form regions; these regions are then merged with other regions to "grow" larger regions. Thus, the term "region growing" is used to describe the process. Figure 6.1 illustrates image segmentation. The top half of the figure shows an image of a woman's face. The bottom half shows the various regions identified within the image.

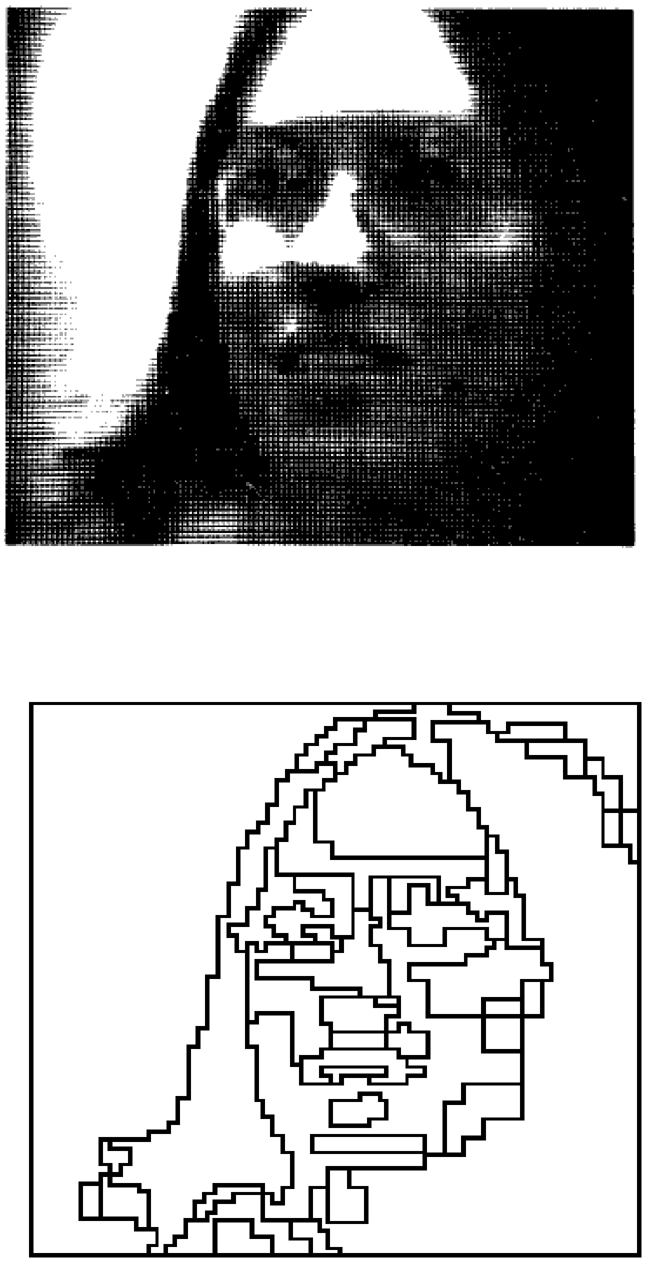


Figure 6.1: An illustration of Image Segmentation

(The figure is reproduced from Horowitz and Pavlidis, 1974)

The merging of regions is usually governed by a homogeneity criterion that must be satisfied. A variety of homogeneity criteria have been proposed [6, 112]. One criterion, known as the *pixel range* homogeneity criterion, requires that the difference between the minimum and maximum pixel intensities within a region not exceed a given threshold value T. It follows that pixels belonging to the same region must have "similar" intensity values. If $f(x,y)$ is the intensity of the pixel with coordinate (x,y) in the image, then the pixel range homogeneity criterion, $H(R)$, for a region R is defined as follows:

$$H(R) = \begin{cases} \mathit{True}, \text{ if for all point pairs } (x_1,y_1) \text{ and } (x_2,y_2) \text{ in } R, \\ \|f(x_1,y_1) - f(x_2,y_2)\| \leq T \\ \\ \mathit{False}, \text{ otherwise.} \end{cases}$$

The input to the region growing problem is composed of the following:

1. An $N \times N$ array of the intensities of the pixels.
2. A threshold value, T, to be used with the pixel range homogeneity criterion.

The output of the region growing problem is a grouping of the pixels into regions such that:

1. Pixels in the same region are connected and satisfy the homogeneity criterion.
2. No two regions can be merged together to form a larger region that satisfies the homogeneity criterion.

Note that the above two requirements guarantee a "good enough" solution to the region growing problem. This solution may be different from the "best" solution which yields the smallest number of regions possible. Also note that more than one grouping of the pixels can satisfy the above two requirements, so the solution is not unique.

The output of the region growing problem is a labeling of the pixels, such that pixels labeled by the same integer value belong to the same region.

6.1 The Split and Merge Approach

There are many algorithms for solving the region growing problem [3, 6, 29, 53, 57, 79, 112]. The effectiveness of a particular algorithm depends on the input image and the requirements of the application at hand. We examine an algorithm for region growing based on the split and merge approach proposed by Horowitz and Pavlidis [60] and later parallelized by Willebeek-LeMair and Reeves [105]. This algorithm is a direct application of the hierarchical clustering paradigm presented in the previous chapter.

The split and merge approach, as expected, consists of two consecutive stages: the split stage and the merge stage. These will be described in detail below.

6.1.1 The Split Stage

The split stage is a preprocessing stage that rapidly identifies initial clusters or regions in the image and thus reduces the number of merge steps in the program. Since this stage involves grouping pixels together into initial clusters, it is actually a preprocessing merge stage.

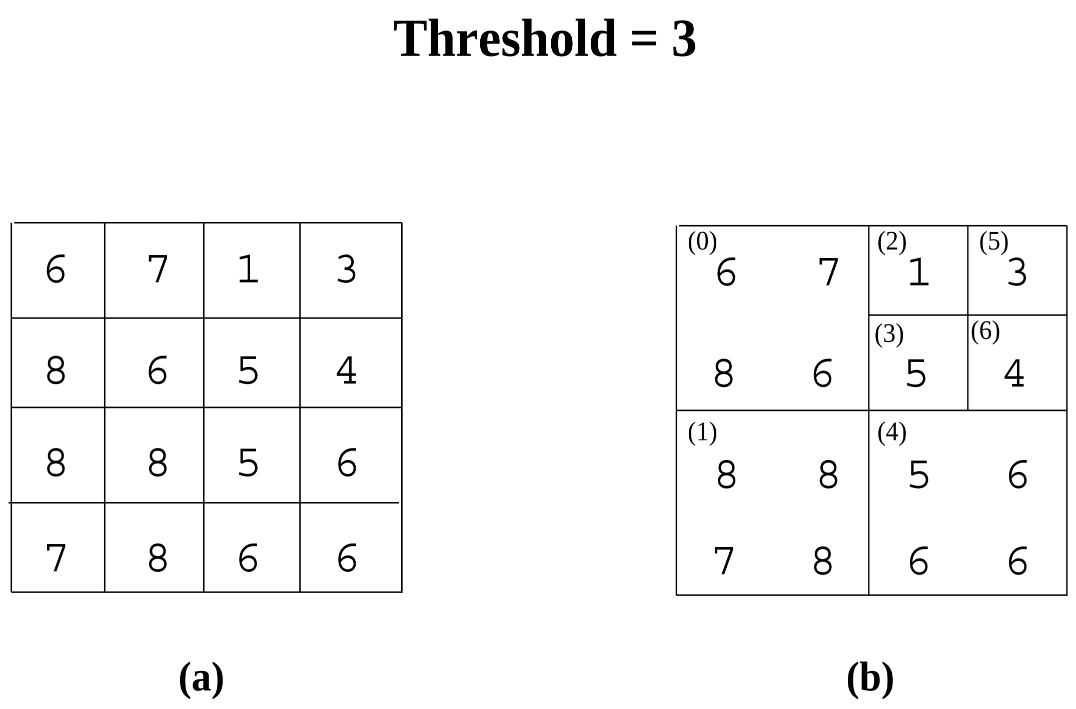
Suppose the image is of size $N \times N$, where N is a power of 2. The image can be partitioned, in a quad-tree fashion, into "square" regions of various dimensions which conform to the homogeneity criterion. At first, each pixel in the image is considered a homogeneous square region of size 1×1. Then every group of four adjacent pixels (corresponding to a quad-tree decomposition) is tested for homogeneity. If the homogeneity criterion is satisfied, the four pixels are combined into one larger square region of size 2×2. Next, every group of four adjacent square regions of size 2×2 is tested for homogeneity. If the homogeneity criterion is satisfied, the four square regions are combined into one larger square region of size 4×4, and so on. The split stage terminates when the whole image is one square region of size $N \times N$, or when no more square regions can be merged. The upper left-hand corner pixel of a square region is designated to be the region representative and is assigned a unique label.

Figure 6.2 illustrates the split stage. Part (a) of the figure shows a 4×4 image, where the threshold value $T = 3$. Part (b) shows the square regions identified after one iteration of the split stage. The small numbers appearing in the upper left-hand corners of the regions denote the region labels.

6.1.2 The Merge Stage

In the merge stage, the square regions determined by the split stage are iteratively merged into larger and larger clusters or regions which conform to the homogeneity criterion. The merge stage can be modeled using a two-layered undirected, weighted multigraph, where the vertices of the multigraph represent the initial clusters or square regions determined by the split stage.

In the first layer of the multigraph, an edge exists between two vertices if the two regions they represent are adjacent in the image. All edges in the first layer are assigned identical weights equal to some arbitrary constant, say 1. The threshold



Square regions: (a) at start of the split stage; (b) after first and final split iteration

Figure 6.2: The Split Stage

value for the first layer is taken to be equal to 0.

In the second layer of the multigraph, an edge exists between two vertices if the difference between the minimum and maximum pixel intensities in the two regions combined is $\leq$ the given threshold T. The weight of the edge is taken to be equal to that difference. The threshold value for the second layer is taken to be equal to the given threshold value T. We use the single-linkage criterion in the first layer of the multigraph and the complete-linkage criterion in the second layer of the multigraph to identify the regions in the image.

Figure 6.3 shows an example of a multigraph modeling the region growing problem. Part (a) of the figure shows the square regions produced by the split stage; part (b) shows the two layers of the corresponding multigraph; and part (c) shows the intersection of the two layers of the multigraph.

In one merge iteration each region selects for merging a region that simultaneously satisfies the single-linkage criterion (connectivity requirement) in layer 1 and the complete-linkage criterion (homogeneity requirement) in layer 2 of the multigraph. If more than one region satisfies these two requirements, then one of these regions is chosen such that the pixel range is minimized. To illustrate, consider Figure 6.4. In that figure, region 4 can merge with region 1, region 3, or region 6, since each of these regions simultaneously satisfies the connectivity and homogeneity requirements. If region 4 merges with region 1, then the difference between the minimum and maximum pixel intensities in the resulting region is equal to 3; if region 4 merges with region 3, then the difference between the minimum and maximum pixel intensities in the resulting region is equal to 1; while if region 4 merges with region 6, then the difference between the minimum and maximum pixel intensities in the resulting region is equal to 2. The smallest of these numbers is 1; therefore, region 4 chooses to merge with region 3. This approach is known as the"best merge" approach and is known to yield good quality regions [105].

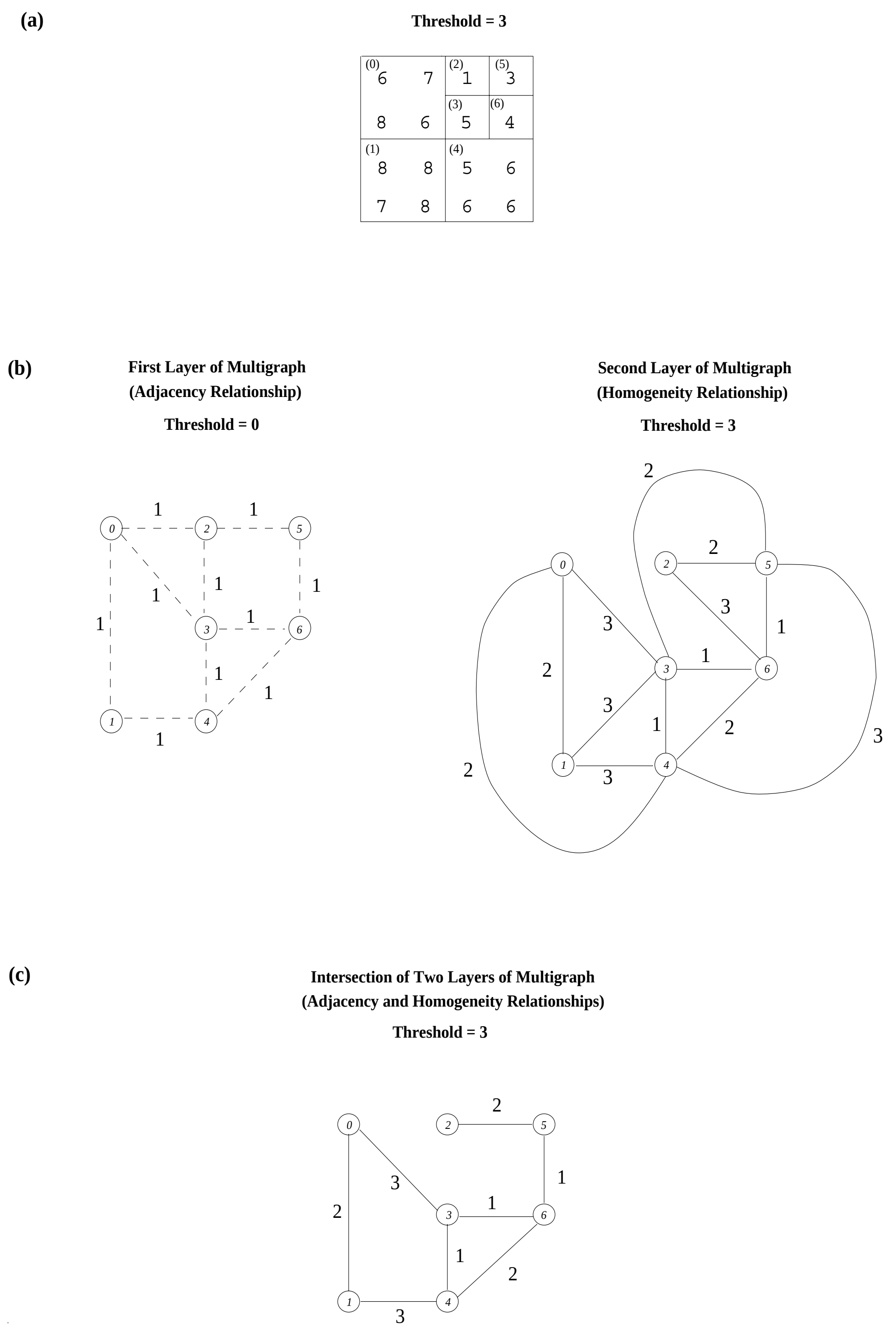

(a)
Threshold = 3
(b)
First Layer of Multigraph
(Adjacency Relationship)
Threshold = 0
Second Layer of Multigraph
(Homogeneity Relationship)
Threshold = 3
(c)
Intersection of Two Layers of Multigraph
(Adjacency and Homogeneity Relationships)
Threshold = 3

Figure 6.3: Regions and Corresponding Multigraph Representation at the Beginning of the Merge Stage

Often in the "best merge" approach the situation arises where two or more neighbors tie for merging with a given region. For example, in Figure 6.3 (c), regions 3 and 5 tie for merging with region 4, since each of these regions minimizes the difference between the minimum and maximum pixel intensities. In such a situation, some criterion must be used to break the tie and choose one of the two regions to merge with region 4. The next section discusses two approaches that can be used.

After each region has selected a partner for merging, two regions *actually merge* if their merge choices are mutual. That is, two regions must select each other in order for them to merge. Once two regions merge, one of the two regions becomes the representative of the two, and the vertices and edges of the multigraph are updated. The merge continues until no more merges are possible.

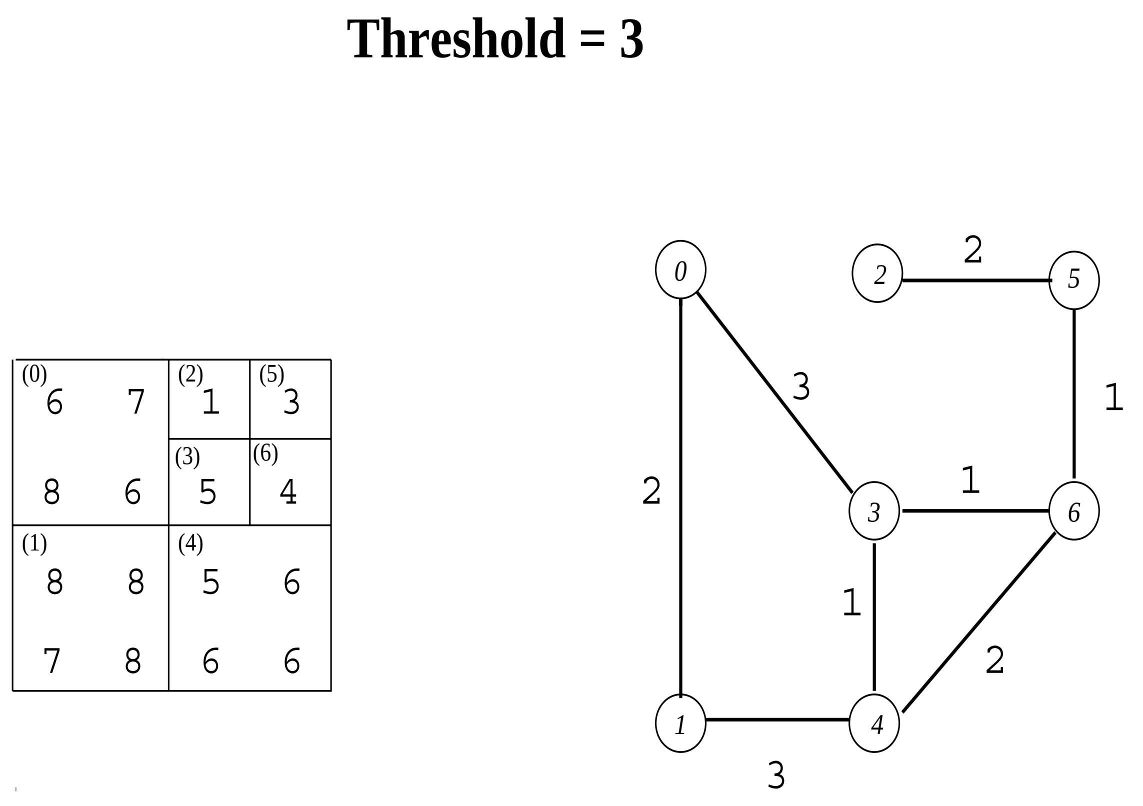


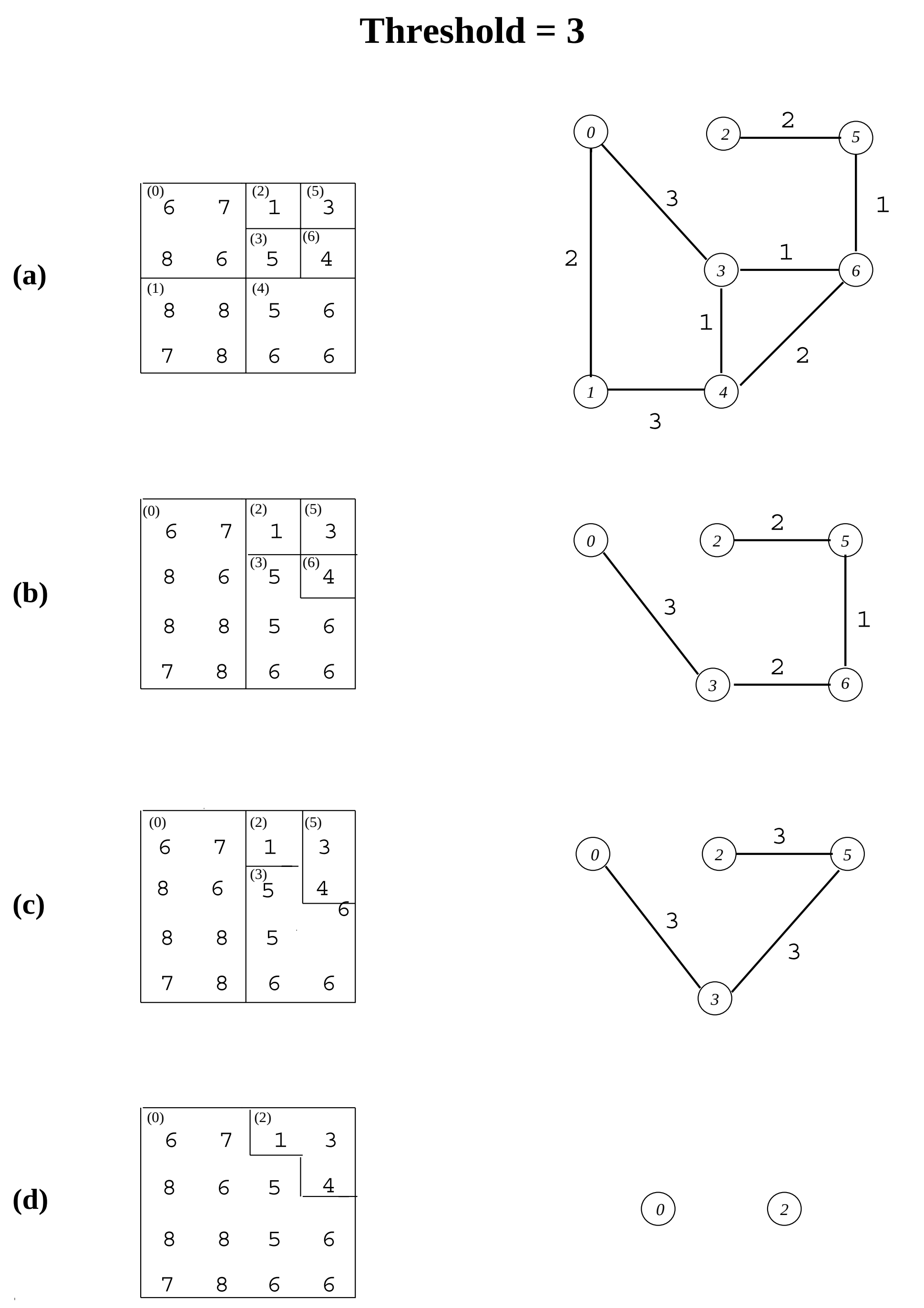
Figure 6.4: Image and Corresponding Graph Representation

6.1.3 Resolving Ties

As mentioned in the previous section, when two or more neighbors tie for merging with a given region, some criterion must be used to break the tie. One approach, known as the *smallest-label approach*, breaks the tie by choosing the neighbor with the smallest label. Another approach, known as the *random approach*, chooses one of the neighbors at random.

Figure 6.5 illustrates the merge stage for the 4×4 image of Figure 6.2 when the smallest-label approach is used for breaking ties. The merge stage takes three iterations to complete, and for each iteration the regions determined in the image and the corresponding graph representation are shown. The graph representing the problem is the intersection of the two layers of the multigraph. The weight of an edge joining two vertices (regions) in the graph is equal to the difference in pixel intensities in the two regions combined. The small numbers in parentheses in the corners of the regions denote the region labels.

In Figure 6.5 regions 3 and 5 tie for merging with region 6, since merging with either of these two regions best satisfies the homogeneity criterion for region 6 (i.e., produces the least increase in pixel range for region 6, as indicated by the weights of the edges). Region 6 chooses to merge with region 3, since ties are broken by choosing the neighbor with the smallest label; but no merge actually takes place, since region 3 chooses to merge with region 4. If, instead, ties were broken at random, then in the first merge iteration regions 5 and 6 could merge at the same time as regions 3 and 4, and the merge stage could take two iterations instead of three, as illustrated by Figure 6.6.

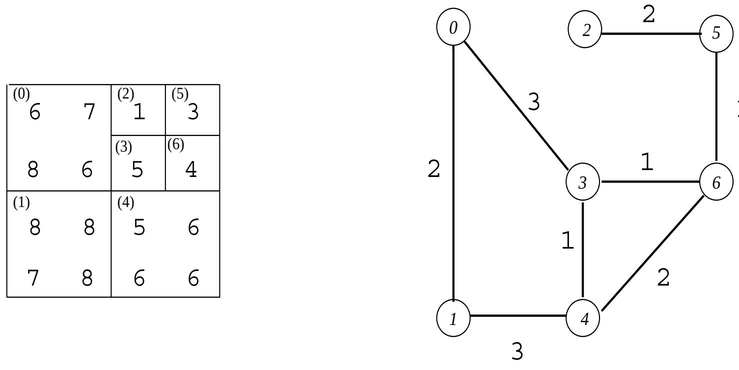


Regions: (a) at start of the merge stage; (b) after first merge iteration;
(c) after second merge iteration; (d) after third and final merge iteration

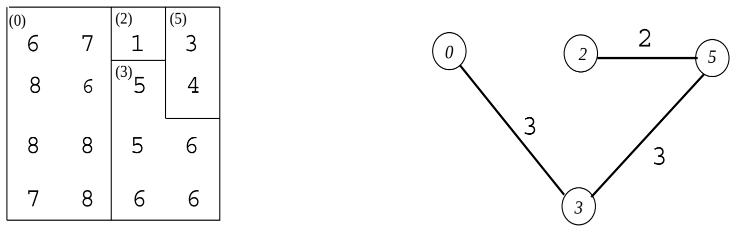
Figure 6.5: The Merge Stage (Smallest-Label Approach in Resolving Ties)

Threshold = 3

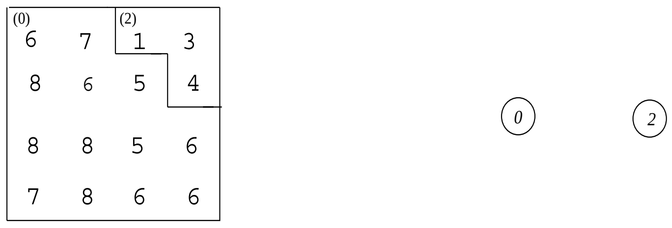
(a)



(b)



(c)



Regions: (a) at start of the merge stage; (b) after first merge iteration; (c) after second and final merge iteration

Figure 6.6: The Merge Stage (Random Approach in Resolving Ties)

To further illustrate the difference between the two approaches, we consider the special case of an image with 7 regions (numbered 0 to 7), where all the pixels have the same intensity value and the threshold value T is some integer ≥ 0. Figure 6.7 shows that it takes the merge stage seven steps to complete when ties are broken by choosing the neighbor with the smallest label. Figure 6.8, on the other hand, shows that it could take the merge stage only three steps when ties are broken by choosing a neighbor at random. Contention for merging is indicated by arrows in the figures. When two regions merge, the new resulting region is labeled by the smaller label of the two regions.

Experimental results with a variety of images have shown that the random approach in resolving ties commonly results in a larger number of merges per merge iteration, and hence is faster when executed on a parallel machine than the approach of selecting the neighbor with smallest (or largest) label. Timings are presented in Section 7.4.

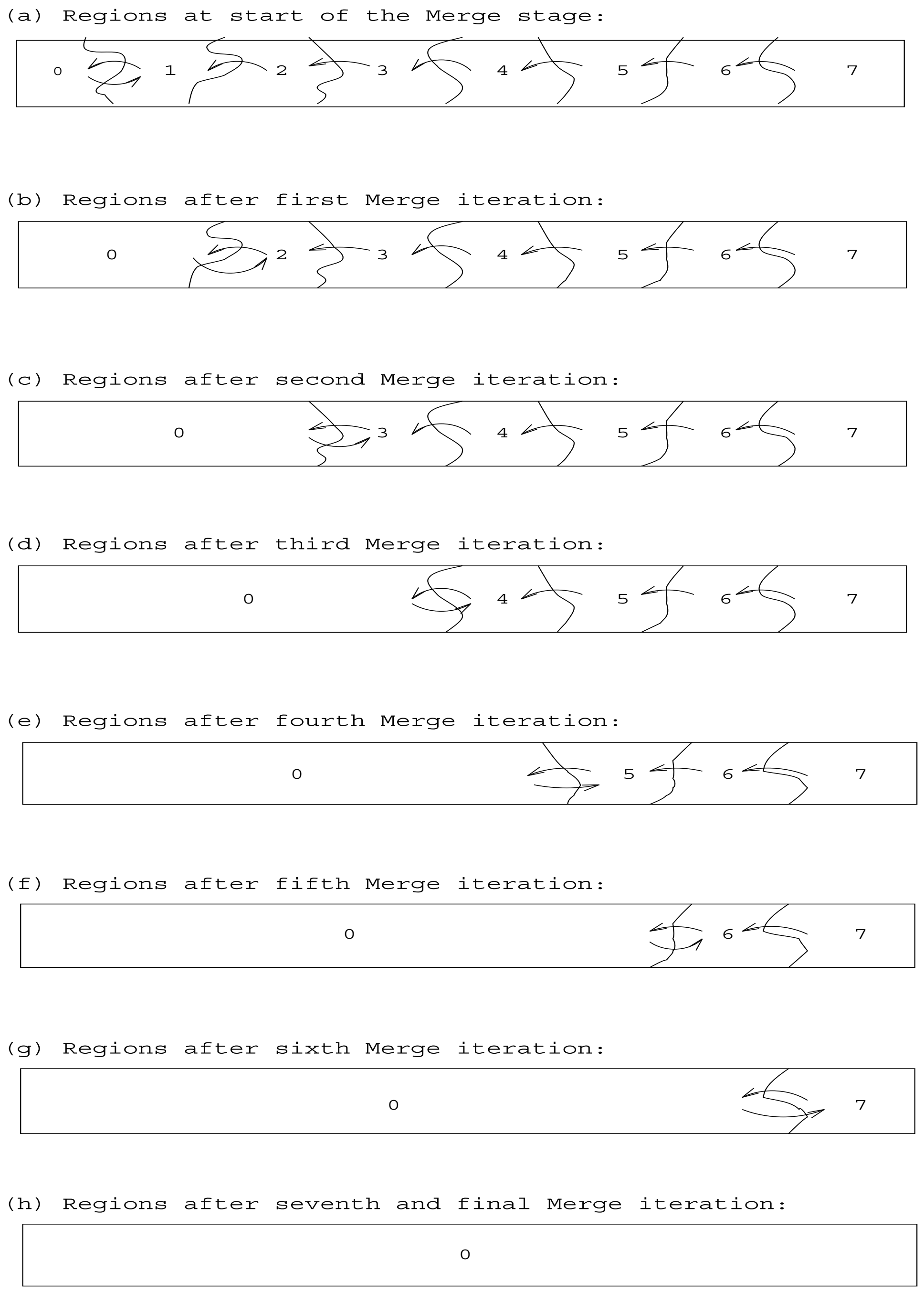

(a) Regions at start of the Merge stage:
0 1 2 3 4 5 6 7
(b) Regions after first Merge iteration:
0 2 3 4 5 6 7
(c) Regions after second Merge iteration:
0 3 4 5 6 7
(d) Regions after third Merge iteration:
0 4 5 6 7
(e) Regions after fourth Merge iteration:
0 5 6 7
(f) Regions after fifth Merge iteration:
0 6 7
(g) Regions after sixth Merge iteration:
0 7
(h) Regions after seventh and final Merge iteration:
0

Figure 6.7: Resolving Ties by Choosing the Neighbor with the Smallest Label

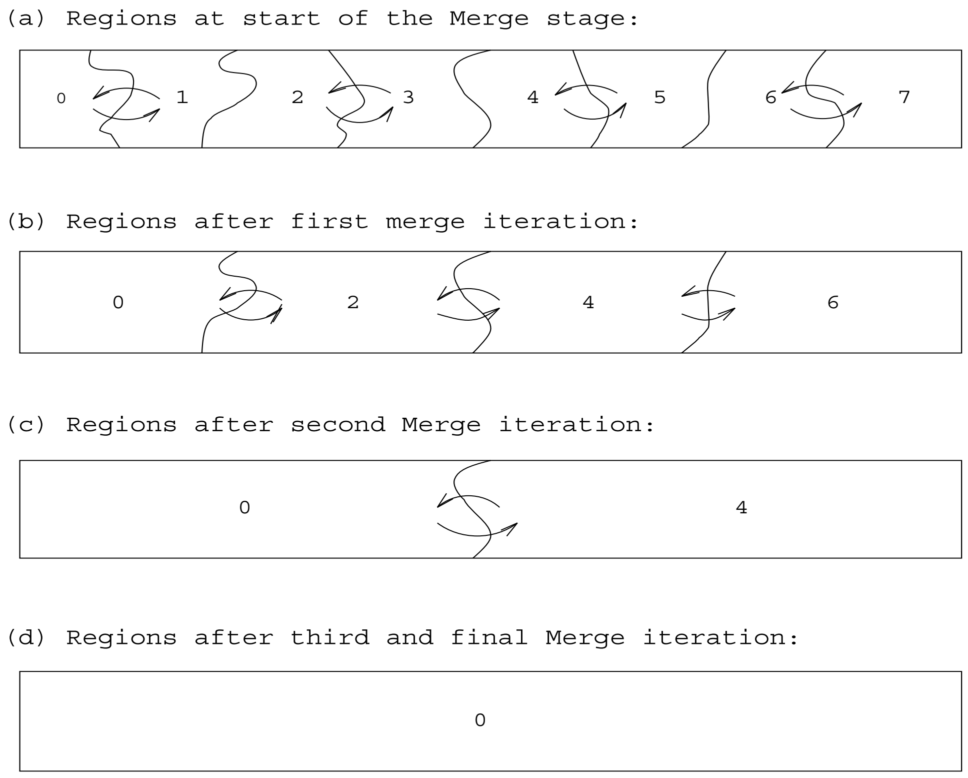

(a) Regions at start of the Merge stage:
0
1
2
3
4
5
6
7
(b) Regions after first merge iteration:
0
2
4
6
(c) Regions after second Merge iteration:
0
4
(d) Regions after third and final Merge iteration:
0

Figure 6.8: Resolving Ties by Choosing a Neighbor at Random

6.1.4 Deadlock

Deadlock refers to the situation when merges are possible, but no pairs of regions select each other for merging and the merge stage iterates indefinitely. This situation arises when region selections for merging form circular chains. Figure 6.9 shows an example of a circular chain, where region 1 selects region 2, region 2 selects region 3, region 3 selects region 4, and region 4 selects region 1 for merging.

The smallest (or largest) label approach in resolving ties guarantees that at least two regions will select each other for merging in every merge iteration. That pair can be determined as follows:

Suppose that there are N regions in the image, labeled from 1 to N. Now consider all the possible merges that can take place among these N regions (the merges must satisfy both the connectivity and homogeneity requirements). Each of the merges can be represented as an unordered pair of region labels, and there are at most $\frac{N\times(N-1)}{2}$ such pairs. Find the pair of regions that yields a region with the smallest pixel range. In general, there may be more than one such pair. Call these pairs $p_1, p_2, \cdots, p_m$, $1 \leq m \leq \frac{N\times(N-1)}{2}$. Examine all the region labels in these m pairs, and find the smallest label, s. Then examine all the partners of s, and find the one with the smallest label; call that partner t. The two regions labeled s and t are the two regions that are guaranteed to merge.

The random approach in resolving ties, on the other hand, may result in circular chains and hence deadlock. To avoid this problem, the program needs to check that in every merge iteration at least two regions select each other for merging. If no merges occur for several merge iterations, then the program can use the smallest (or largest) label approach to break the deadlock and then switch back to the random approach.

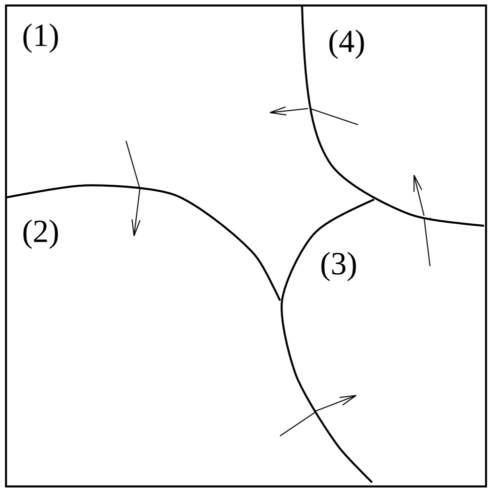


Figure 6.9: A Circular Chain of Region Selections for Merging

(An arrow from region i to region j indicates that region i selects region j for merging)

Chapter 7

Parallel Implementations of Region Growing

In this chapter we examine three parallel implementations of region growing, based on the hierarchical clustering paradigm:

1. A message passing implementation in Fortran 77 plus CMMD [96], executed on a 32-node Connection Machine model CM-5. The implementation consists of two programs, a host program and a node program. The host program runs on the partition manager, while 32 identical copies of the node program run independently on the node processors.

2. A data parallel implementation in the CM Fortran language [94], executed on a 32-node Connection Machine model CM-5.

3. A data parallel implementation in High Performance Fortran, executed on a cluster of DEC Alpha workstations.

The first and second implementations allow us to make a meaningful comparison of the performance of the message passing and data parallel implementations, since they execute on the same platform (a 32-node CM-5). The third implementation allows us to discuss the language and runtime support needed for hierarchical clustering, using the High Performance Fortran Language.

7.1 The Message Passing Implementation

The message passing model of execution requires the programmer to explicitly specify the detailed behavior of individual processors operating asynchronously. The facilities provided by the system software in the data parallel model are exchanged for the ability to program each node individually and to make explicit decisions on communication, synchronization, and load balancing.

The message passing implementation of region growing is written in Fortran 77 and CMMD, and executed on a 32-node Connection Machine model CM-5. The message passing implementation consists of the following steps:

Step 1: The host processor inputs the $N \times N$ array of pixel intensities and the threshold value T. Next, the host processor distributes the pixel intensities among a $P1 \times P2$ node-processor grid (where both $P1$ and $P2$ divide N), such that each processor receives an $\frac{N}{P1} \times \frac{N}{P2}$ sub-image of the original image.

Step 2: Every node processor splits its $\frac{N}{P1} \times \frac{N}{P2}$ sub-image and determines the homogeneous square regions within it. The splitting is performed *in parallel* by all the node processors. If the sub-image within the processor is rectangular in shape, then it is divided into square sections, and the split stage is applied independently to each of these sections in turn.

Step 3: Each node processor constructs the vertices and edges of the graph associated with its own sub-image. The graphs are constructed *in parallel* by all the node processors. Border information is exchanged to set up edges between adjacent processors. For every square region, a corresponding vertex is created, and for each pair of neighboring square regions an edge is created. The weight of the edge is taken to be equal to the difference between the minimum and maximum pixel intensities in the two regions combined. If the weight exceeds the threshold value T, then the edge is marked as inactive; otherwise it is active. Boundary information is exchanged between the node processors so that edges connected to vertices in other processors are created.

Step 4: The node processors cooperate to merge pairs of adjacent regions into larger regions that satisfy the homogeneity criterion. This is achieved by a loop over active edges in each of the node processors. Each region connected to an active edge determines the neighboring region that best satisfies the homogeneity criterion. In the case of a tie, one of the neighboring regions is chosen *at random*. Two regions merge if their merge choices are mutual. Several region pairs can merge at the same time without conflicting with each other.

Step 5: The node processors cooperate to update the vertices and edges of the graph to reflect the new regions in the images. Edges that do not satisfy the homogeneity criterion are deactivated.

Step 6: If there still exist any active edges, then go to step 4. Otherwise, continue on to step 7.

Step 7: The node programs send information about their final regions to the host.

The host outputs the results, and the host and node programs terminate.

Figure 7.1 illustrates how the split stage is applied to square sections of an image. Assuming $P1 \times P2$ node processors are used, where $P2$ is twice $P1$, then the $\frac{N}{P1} \times \frac{N}{P2}$ sub-image in one processor is divided into two square sections, and the split stage is applied independently to each of these sections in turn.

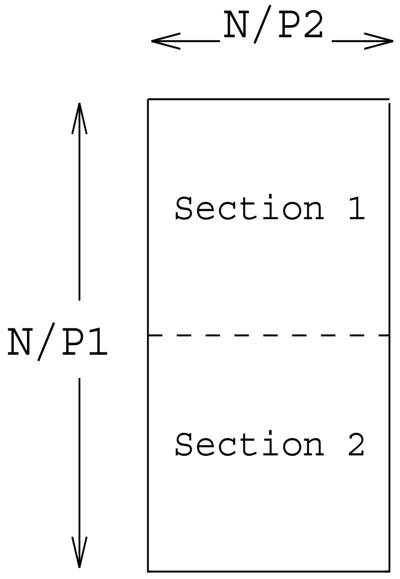


Figure 7.1: A Sub-Image within a Processor Divided into Two Square Sections

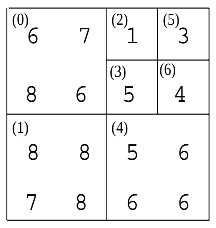
7.1.1 Data Structures

In applying the hierarchical clustering paradigm to region growing, only one- and two-dimensional arrays are used to represent the various data items required. Operations on these arrays are specified sequentially using DO loops, but different processors work on different sections of the arrays simultaneously.

Two-dimensional arrays are used to store the intensities as well as other information pertaining to the pixels such as the pixel column and row numbers and whether a pixel is a region representative or not. One-dimensional arrays are used to store information about the vertices and edges of the graph modeling the problem. Figure 7.2 illustrates the way in which data is stored in the various arrays. Note that edges are represented by two one-dimensional arrays that index the vertex arrays. The first array indexes the first region of the edge, while the second array indexes the second region of the edge.

Threshold = 3

(a)



(b)

Region label:	0	1	2	3	4	5	6
Min. pixel value:	6	7	1	5	5	3	4
Max. pixel value:	8	8	1	5	6	3	4

(c)

Label of first region of Edge:	0	0	0	1	2	2	3	3	4	5
Label of second region of Edge:	1	2	3	4	3	5	4	6	6	6
Min. pixel value in two regions combined:	6	1	5	5	1	1	5	4	4	3
Max. pixel value in two regions combined:	8	8	8	8	5	3	6	5	6	4
Edge active?	Yes	No	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes

(a) Square regions determined by the split stage; (b) Information on vertices of graph; (c) Information on edges of graph

Figure 7.2: Array Data Structures Used in Parallel Implementations

7.1.2 Data Distribution

In the message passing implementation, the two-dimensional array of pixel intensities is distributed equally among the node processors in the (BLOCK, BLOCK) fashion. First the host processor reads the entire array into its memory; then it sends blocks of the image to the node processors.

Given an $N \times N$ array of pixel intensities and $P1 \times P2$ processors (where each of $P1$ and $P2$ divides N), the (BLOCK, BLOCK) scheme maps the array to the processor grid such that each processor receives an $\frac{N}{P1} \times \frac{N}{P2}$ sub-image of the original image. This mapping maintains adjacency between neighboring blocks of the image. The row and column numbers of a node processor in the grid are given by:

$$\begin{aligned} \text{Row} &= (\text{Self address DIV } P2) + 1 \\ \text{Column} &= (\text{Self address MOD } P2) + 1, \end{aligned}$$

where DIV is the integer division operation and MOD is the modulo operation. Figure 7.3 shows an $N \times N$ image mapped to a 4×8 processor grid ($P1 = 4$, $P2 = 8$). The processors are numbered $p_0, p_1, \cdots, p_{31}$.

At the end of the split stage, the node processors set up the vertices and edges of the graph representing the problem. Three different mappings of an edge and its two vertices can arise when distributing a graph data structure among a number of processors. These cases are illustrated in Figure 7.4. In part (a) of the figure, an edge and both its vertices are mapped to the same processor. In part (b), an edge and one of its vertices are mapped to one processor, while the other vertex is mapped to another processor. In part (c), an edge and its two vertices are mapped to three different processors. Obviously, case (a) is the most desirable since it does

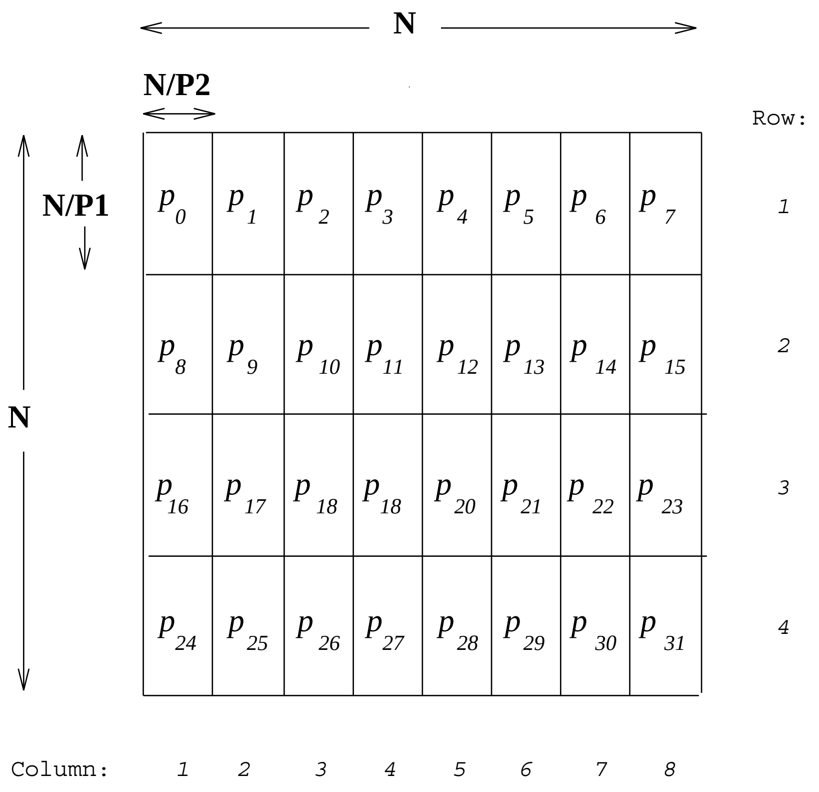

N
N/P2
Row:
N/P1
N
p_0 p_1 p_2 p_3 p_4 p_5 p_6 p_7
p_8 p_9 p_{10} p_{11} p_{12} p_{13} p_{14} p_{15}
p_{16} p_{17} p_{18} p_{18} p_{20} p_{21} p_{22} p_{23}
p_{24} p_{25} p_{26} p_{27} p_{28} p_{29} p_{30} p_{31}
1
2
3
4
Column: 1 2 3 4 5 6 7 8

Figure 7.3: Mapping an $N \times N$ image onto a 4×8 processor grid

not require any communication. Ideally, a partitioning of a graph should consist of mostly case (a) mappings and only a few case (b) mappings and should divide the vertices and edges evenly among the processors.

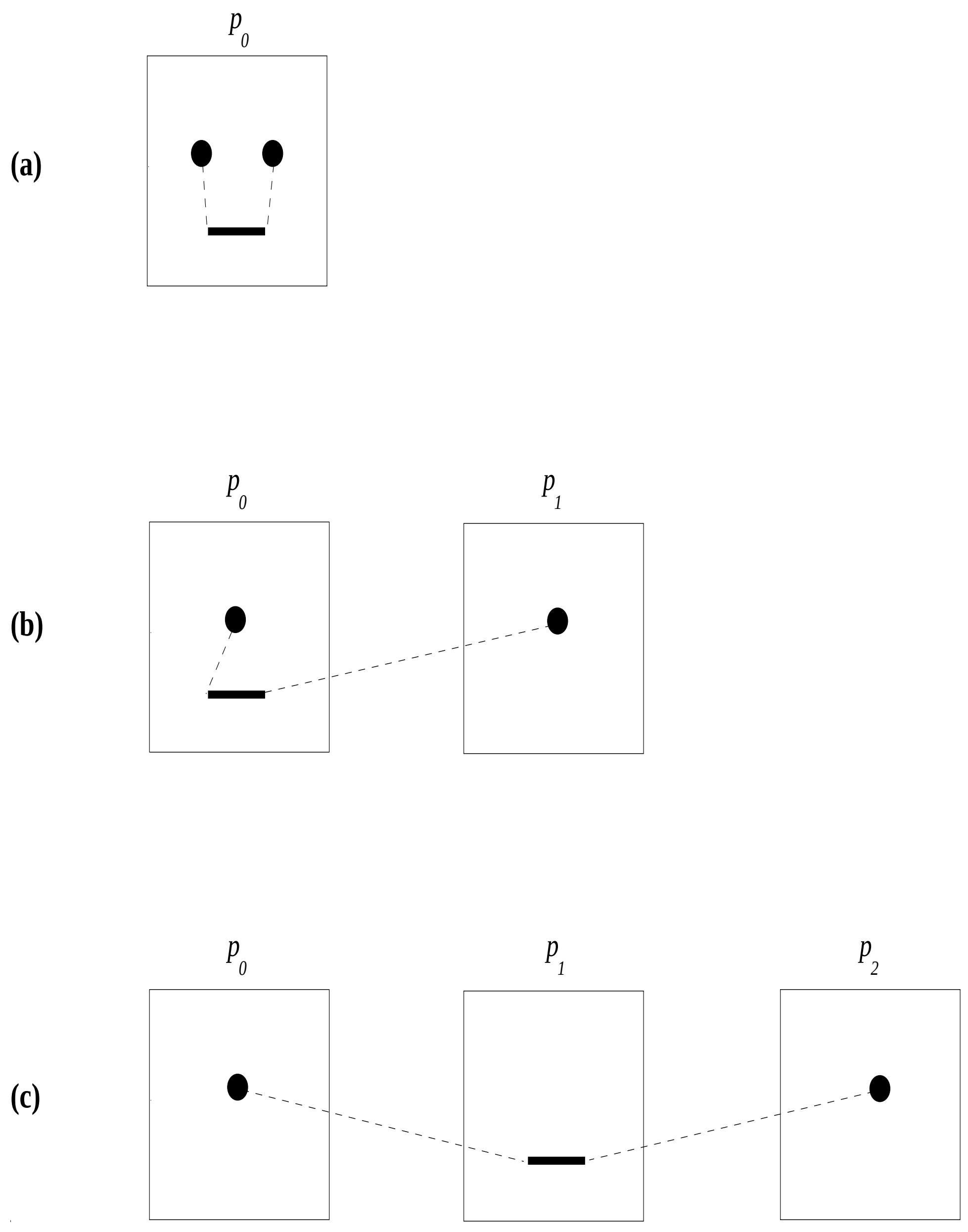


Figure 7.4: Three Different Mappings of an Edge of a Graph and its Two Vertices

In the message passing implementation, every node processor *independently* sets up the edges and vertices of the graph associated with its *own sub-image*. Border information is exchanged to set up edges between adjacent processors. Thus either an edge and both of its vertices are mapped to the same processor (case (a)), or an edge and one of its vertices are mapped to the same processor and the second vertex is mapped to an adjacent processor (case (b)). The situation where an edge and its two vertices are mapped to three different processors (case (c)) does not arise. Thus locality between the vertices and edges of the graph is maintained.

Furthermore, since the number and shape of square regions are not identical in all of the sub-images, it follows that the number of vertices and edges of the graph are not identical in all of the processors. In effect, the vertex and edge arrays are distributed such that different processors are assigned different-sized blocks of the arrays (GENERAL BLOCK distribution [27]). The two logical arrays, VERTEX_ACTIVE and EDGE_ACTIVE, which indicate the number of active elements in the vertex and edge arrays, are also mapped in an GENERAL BLOCK fashion. Figure 7.5 illustrates GENERAL_BLOCK distribution of the graph in Figure 7.2 on two processors.

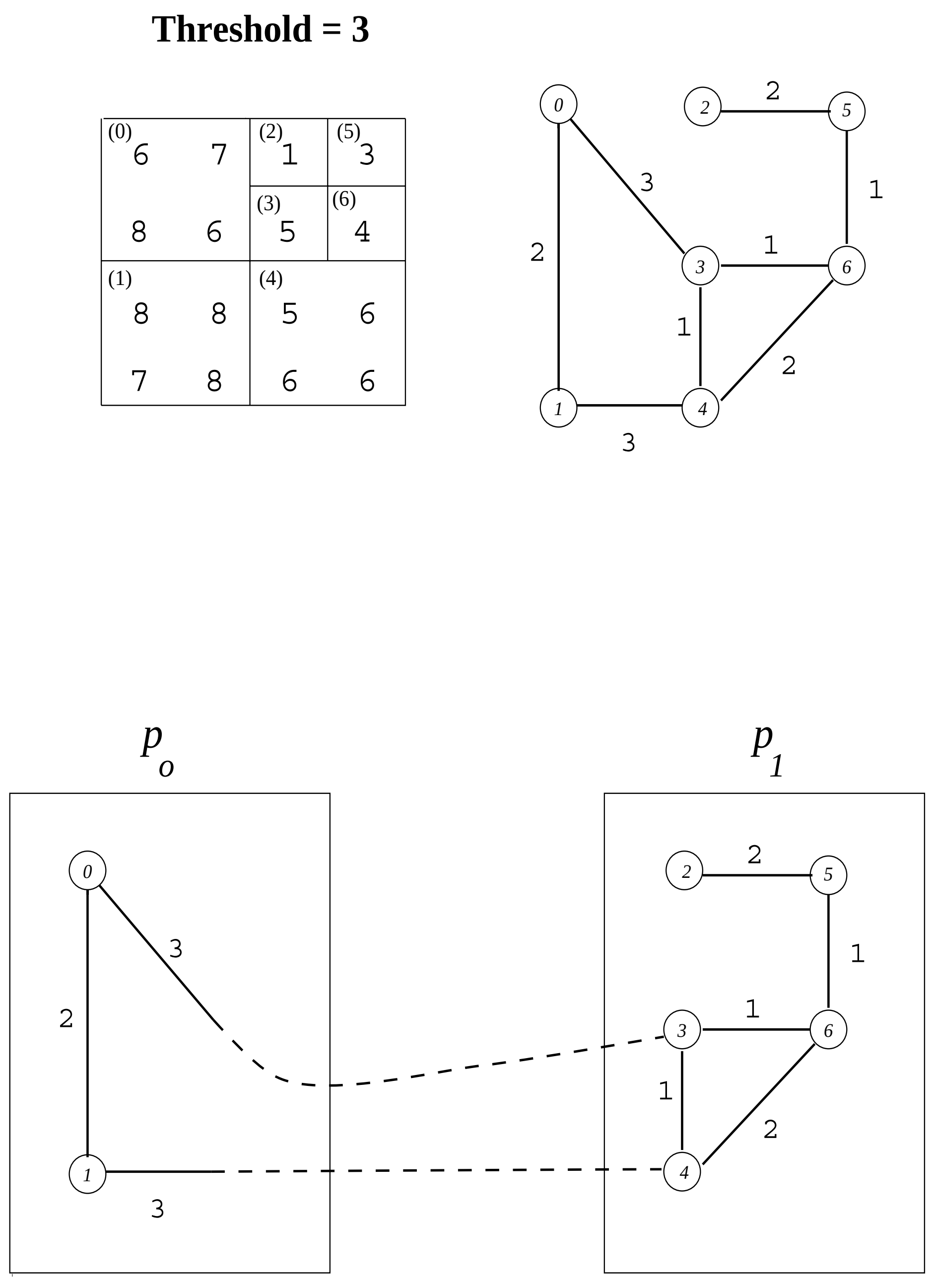


Figure 7.5: Example of GENERAL_BLOCK Distribution

7.1.3 Unstructured Communication

At several points in the message passing implementation, unstructured communication is required, whereby each of the node processors sends zero or more messages to other processors in an irregular fashion. For example, we may have a situation similar to the one shown in Figure 7.6 where each of four processors has a list of destinations to which to send messages. The destinations and sources do not follow a specific structure or pattern. In this case an efficient communication scheme is needed whereby messages are sent and received without causing deadlock.

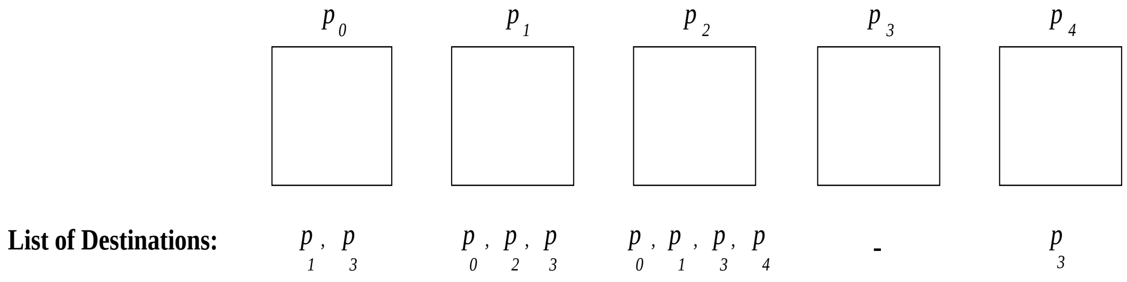


Figure 7.6: An Example of Unstructured Communication

Two different communication schemes were investigated. The first, called *Linear Permutation* (LP) [83], uses *Blocking Send* and *Blocking Receive* (blocking here means that the sending process and receiving process wait until the communication has completed before resuming execution). In this scheme each node obtains a copy of the communication matrix, using a global concatenation operation. Then, in step i, $0 < i < Q$, processor p_k sends a message to processor $p_{(k+i)\ MOD\ Q}$ and receives a message from processor $p_{(k-i)\ MOD\ Q}$, where Q is the total number of node processors. The sender and receiver nodes are blocked until the message is transmitted. The steps of the *Linear Permutation* algorithm are as follows:

```
For all processors $p_k$, $0 \leq k \leq Q-1$, in parallel do
for $i = 1$ to $Q-1$ do
  Processor $p_k$ sends a message to processor $p_{(k+i)\ MOD\ Q}$
  Processor $p_k$ receives a message from processor $p_{(k-i)\ MOD\ Q}$
endfor
```

If processor p_k does not have any message to send to processor $p_{(k+i)\ MOD\ Q}$ for some value of i, $1 \leq i \leq Q-1$, then it does not participate in the send operation for step i. Similarly, if processor p_k does not have any message to receive from processor $p_{(k-i)\ MOD\ Q}$, then it does not participate in the receive operation for step i.

The second communication scheme uses *Non-Blocking Send* and *Blocking Receive* (NS-BR). In this scheme a node that wishes to send a message does not block until its partner node is ready to receive that message, but can pursue other computation. When the receiving node is ready to receive the message, it issues a (blocking) receive and waits for the message to be delivered before it can pursue other work. The steps of the communication scheme are as follows:

1. Using a global reduction operation, each node determines the number of messages it must receive from the other nodes.

2. Every node sends all the messages it wishes to send to other nodes using non-blocking send.

3. Every node loops the required number of times to receive all the messages (if any) that have been sent to it by other nodes. The nodes use blocking receive.

In order to reduce the communication overhead in both schemes, whenever a processor needs to send more than one message to the same destination, all the messages are concatenated together and sent as one large message.

7.2 Data Parallel Implementations

The data parallel implementations (CM Fortran and HPF) of the region growing problem consist of the following steps:

Step 1: The program inputs the $N \times N$ array of pixel intensities and the threshold value T.

Step 2: The two-dimensional pixel image is repeatedly split into homogeneous square regions. Homogeneous regions in different parts of the image can be identified in parallel.

Step 3: For every square region a corresponding vertex is created, and for each pair of neighboring square regions an edge is created. The weight of the edge is taken to be equal to the difference between the minimum and maximum pixel intensities in the two regions combined. If the weight exceeds the threshold value T, then the edge is marked as inactive; otherwise it is active. Boundary information is exchanged between the processors so that edges connected to vertices in other processors are created.

Step 4: Each region connected to an active edge determines its neighboring region that best satisfies the homogeneity criterion. In the case of a tie, one of the neighboring regions is chosen *at random.* Two regions merge if their merge choices are mutual. Several region pairs can merge at the same time without conflicting with each other.

Step 5: The vertices and edges of the graph are updated to reflect the new regions in the image. Edges that do not satisfy the homogeneity criterion are deactivated. Different portions of the graph can be updated in parallel.

Step 6: If there still exist any active edges, then go to Step 4. Otherwise, continue on to step 7.

Step 7: Output the results and terminate.

7.2.1 Data Structures

In the data parallel implementation, as in the message passing implementation, only one- and two-dimensional arrays are used to represent the various data items required. Array data structures are ideal for data parallel languages, as operations can be applied simultaneously to many elements of an array.

7.2.2 Data Distribution

In the data parallel implementations (CM Fortran and HPF), as in the message passing implementation, the two-dimensional array of pixel intensities, PIXEL_VAL, is distributed among the processors using the (BLOCK, BLOCK) distribution scheme (see Figure 7.3). In CM Fortran the (BLOCK, BLOCK) distribution scheme is implemented by default by the compiler. In HPF the programmer must explicitly provide directives to obtain such a distribution:

```
       INTEGER, DIMENSION (N, N) :: PIXEL_VAL

!HPF$  PROCESSORS, DIMENSION (Q1, Q2) :: P2
!HPF$  DISTRIBUTE (BLOCK, BLOCK) ONTO P2 :: PIXEL_VAL
```

The one-dimensional arrays describing the edges and vertices of the graph are distributed among the processors using the BLOCK distribution scheme. In CM Fortran this scheme is implemented by default by the compiler. In HPF the programmer must explicitly provide directives to obtain such a distribution:

```
       INTEGER, ALLOCATABLE :: VERTEX_LABEL(:), VERTEX_MIN(:),
     &    VERTEX_MAX(:), VERTEX_MERGE(:), VERTEX_PARTNER(:),
     &    EDGE_VERTEX_1(:), EDGE_VERTEX_2(:), EDGE_MIN(:),
     &    EDGE_MAX(:), EDGE_ACTIVE(:)

!HPF$  PROCESSORS, DIMENSION (Q1 * Q2) :: P1
!HPF$  DISTRIBUTE (BLOCK) ONTO P1 :: VERTEX_LABEL
!HPF$  ALIGN WITH VERTEX_LABEL :: VERTEX_MIN, VERTEX_MAX,
     &    VERTEX_MERGE, VERTEX_PARTNER

!HPF$  DISTRIBUTE (BLOCK) ONTO P1 :: EDGE_VERTEX_1
!HPF$  ALIGN WITH EDGE_ACTIVE :: EDGE_VERTEX_1, EDGE_VERTEX_2,
     &    EDGE_MIN, EDGE_MAX
```

Given P processors and a one-dimensional array of size M (where M is a multiple of P), the BLOCK distribution scheme divides the array such that each (abstract) processor receives a contiguous block of the array of size $\frac{M}{P}$. Although this distribution divides the computational load more evenly among the processors, case (c) mappings (see Figure 7.4) will generally arise and lead to increased communication. Figure 7.7 illustrates BLOCK distribution of the graph in Figure 7.2 on two processors.

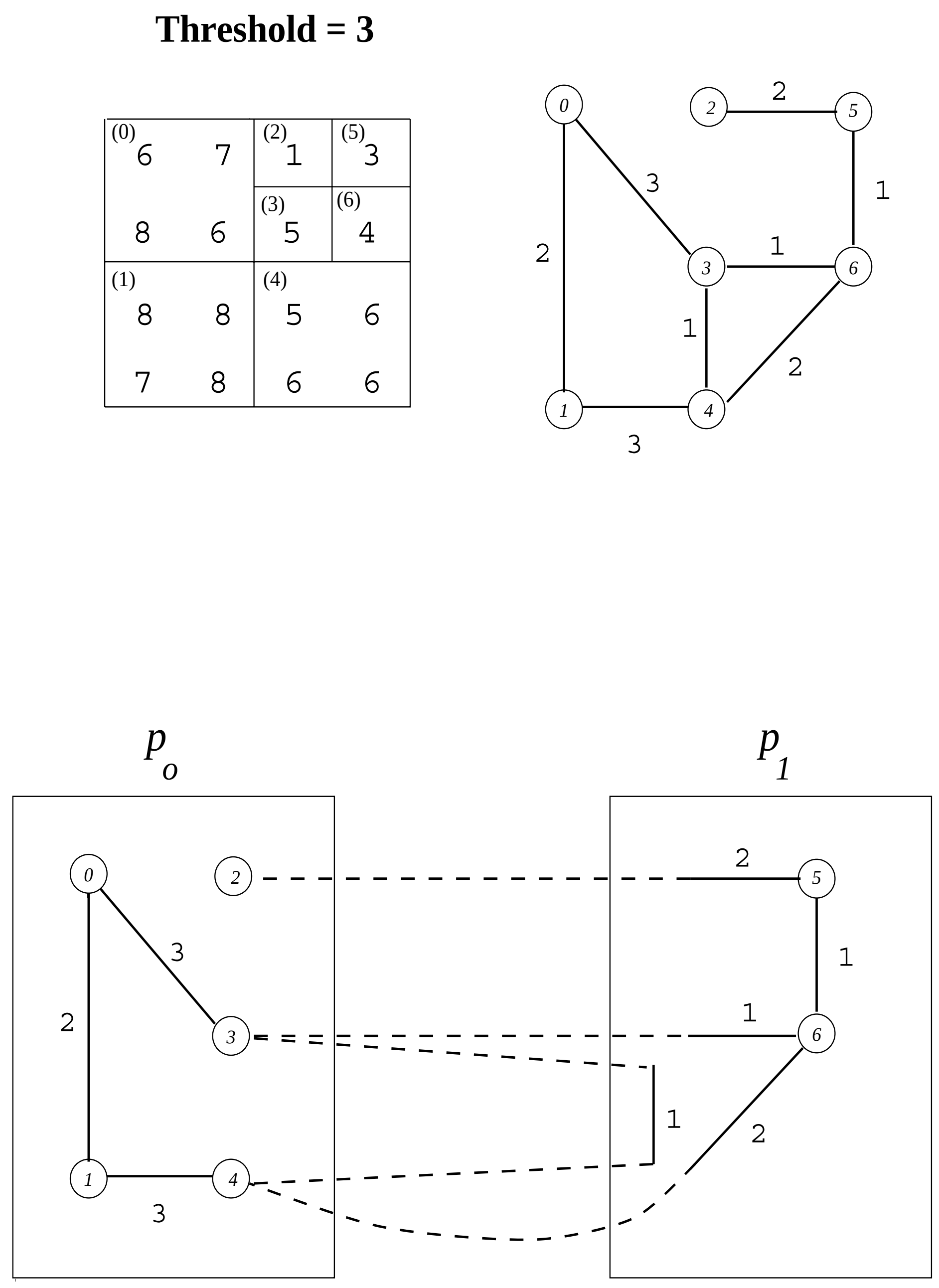


Figure 7.7: Example of BLOCK Distribution

7.2.3 Unstructured Communication

Communication is required whenever data items required for a computation reside on different processors. In the data parallel implementations, communication is implicit and is managed by the compiler and the runtime system.

When indirection arrays are used with FORALL, data accesses are known only at runtime, and the compiler generates calls to a runtime library that handles the required communications. In region growing, indirection arrays are used at several points in the computation, especially since edges are represented using two arrays that index the vertex arrays. An example of a FORALL that uses indirection is given below:

```
      FORALL (I=1:M2, EDGE_ACTIVE(I))
     &    VERTEX_CAND(EDGE_VERTEX_1(I)) = .TRUE.
```

Besides FORALL, CM Fortran and HPF provide library procedures that perform certain communication functions. For example, HPF provides the MINVAL_SCATTER procedure that performs parallel reduction with minimum. The following example illustrates the use of MINVAL_SCATTER when regions select the neighboring regions they wish to merge with:

```
      VERTEX_KEY = MINVAL_SCATTER (KEY_2, VERTEX_KEY, EDGE_VERTEX_1,
     &    MASK = EDGE_ACTIVE)
```

In the above example, elements of the array KEY_2 selected by the mask EDGE_ACTIVE are scattered to positions of the array VERTEX_KEY specified by the index array EDGE_VERTEX_1. Each element of the result is assigned the minimum value of the corresponding element of Base and the elements of EDGE_VERTEX_1 scattered to that position.

HPF versions of the various subroutines used in region growing are listed in Appendix A.

7.3 Complexity

Given an $N \times N$ pixel image, the complexity of region growing depends on the number of processors used and the number of iterations required to find the regions in the image. This number in turn depends on the shape and size of the regions.

Suppose the input image is of size $N \times N$, and suppose that P processors are used.

In both the data parallel and message passing implementations, the split stage is applied only to the sub-image within each processor and does not require any communication. Suppose the image is divided evenly among the P processors, such that each processor contains a sub-image of size $M \times M$, where M is equal to $\sqrt{\frac{N^2}{P}}$. In the best case, when every pixel is a region by itself, only one split iteration is required. In the worst case, when the whole sub-image within a node processor is one homogeneous square region, $log(M)$ split iterations are required.

The number of iterations needed to complete the merge stage of the algorithm is bounded above by the maximum number of sub-regions that must be merged to construct any single region in the image. If a region consists of r sub-regions, then it will require at least $log(r)$ merge iterations. In the worst case, when only one pair of regions is merged in each iteration, it will require $r-1$ merge iterations.

The total time for the merge stage depends on the number of regions in the image at the beginning and at the end of the merge stage. Let R_i and R_f denote these two numbers, respectively. Suppose the number of regions is reduced by a factor of k at every step in the merge stage ($1 \leq k \leq 2$). Then the number of iterations required is $log_k \frac{R_i}{R_f}$. The exact value of k depends on the input image and on how ties are resolved. As the timings in Table I show, the random tie breaking approach generally results in a greater value of k than the smallest (or largest) ID approach.

The number of edges, E, and the number of regions, R_i, at the beginning of the merge stage can be derived by Euler's formula [35]:

$$C + R_i - E = 2,$$

where C is the total number of corners of the square regions. It follows that

$$E = C + R_i - 2.$$

Since each region has at most four corners, we have

$$C \leq 4 \times R_i.$$

From the above two equations, it follows that

$$R_i \leq E \leq 5 \times R_i,$$

which shows that the number of edges is linearly proportional to the number of regions.

In the data parallel and message passing implementations, each merge step of the algorithm requires many-to-many communication. The complexity of the many-to-many communication is difficult to analyze, since it depends on the number of messages sent by every processor, which in turn depends on the image.

7.4 Performance

The message passing implementation (written in Fortran 77 plus CMMD) and the data parallel implementation (written in CM Fortran) were executed on a 32-node CM-5. Figures 7.8 and 7.9 show the various input images used. Performance results are presented in the following two sections.

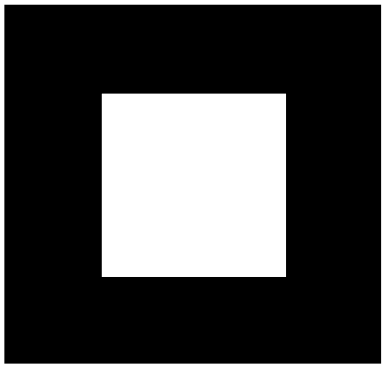


Image 1 :

128 x 128 binary image
composed of two nested regions

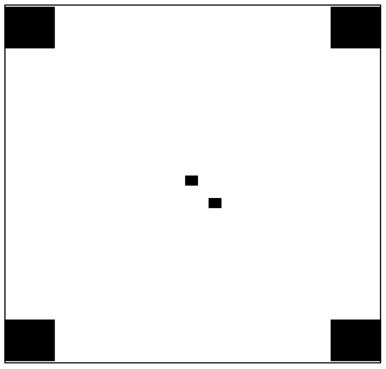


Image 2 :

128 x 128 grey scale image
composed of seven regions

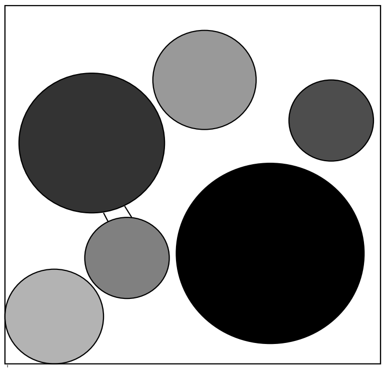


Image 3 :

128 x 128 grey scale image
composed of 11 regions

Figure 7.8: Input Images 1 – 3

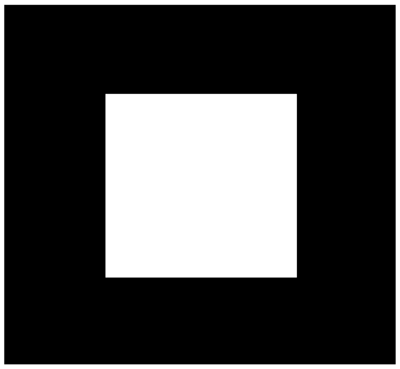


Image 4 :

256 x 256 binary image
composed of two nested regions

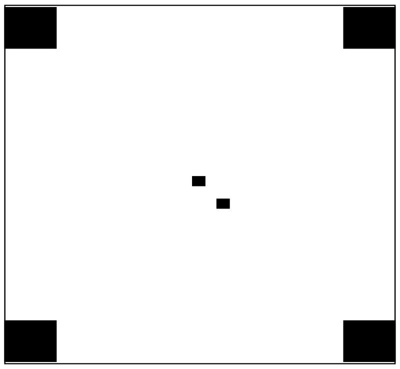


Image 5 :

256 x 256 grey scale image
composed of seven regions

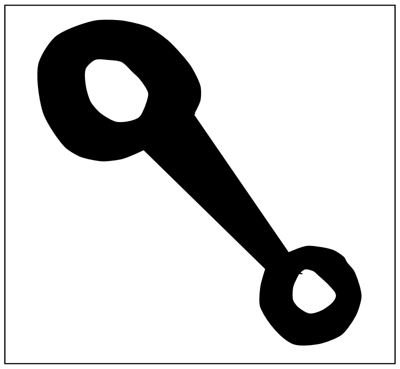


Image 6 :

256 x 256 binary image
composed of 4 regions

Figure 7.9: Input Images 4 – 6

7.4.1 Comparison of Smallest-Label and Random Approaches in Resolving Ties

Table 7.1 compares the smallest-label and random approaches in resolving ties during the merge stage. The table presents the execution time and the number of iterations required by the merge stage of the data parallel implementation (written in CM Fortran) on the CM-5, using each of the two approaches. Invariably, for all of the images the random approach in resolving ties proved to be faster than the approach of selecting the region with the smallest label, as it required a smaller number of iterations. Similar results are obtained for the message passing implementation on the CM-5.

The number of merge iterations required to find the regions in the images is not the same in all cases. This is due to the element of randomness that is introduced in selecting a neighbor for merging. The random numbers generated affect the actual merges that take place and hence the number of merge iterations required to solve the problem.

Table 7.1: Smallest-Label vs. Random Approach in Resolving Ties

CM Fortran Implementation on a 32-Node CM-5

	Merge Stage (Smallest-Label Approach)		Merge Stage (Random Approach)	
	Time (sec)	Iterations	Time (sec)	Iterations
Image 1:	334.948	290	33.013	19
Image 2:	151.670	153	31.615	20
Image 3:	1406.099	809	42.570	27
Image 4:	622.980	549	37.588	25
Image 5:	186.834	226	24.471	16
Image 6:	1754.254	1062	75.582	45

7.4.2 Comparison of the Data Parallel and Message Passing Implementations

Tables 7.2 – 7.7 compare the performance of the message passing and data parallel implementations on the CM-5 when the random approach in resolving ties is used. **LP** refers to the *Linear Permutation* communication scheme, while **NS-BR** refers to the *Non-Blocking Send, Blocking Receive* communication scheme.

The bar chart of Figure 7.10 gives a visual comparison of the times taken by the merge stage in the various implementations on the CM-5.

Table 7.2: Image 1 Performance Data

No. of square regions found at end of split stage = 436
No. of regions found at end of merge stage = 2

	Split Stage		Merge Stage (Random Tie Break)	
	Time (sec)	Iterations	Time (sec)	Iterations
CM Fortan	0.361	4	33.013	19
F77 + CMMD (LP)	0.022	4	6.914	24
F77 + CMMD (NS-BR)	0.021	4	4.025	20

Table 7.3: Image 2 Performance Data

No. of square regions found at end of split stage = 193
No. of regions found at end of merge stage = 7

	Split Stage		Merge Stage (Random Tie Break)	
	Time (sec)	Iterations	Time (sec)	Iterations
CM Fortran	0.360	4	31.615	20
F77 + CMMD (LP)	0.022	4	9.236	35
F77 + CMMD (NS-BR)	0.021	4	6.441	35

Table 7.4: Image 3 Performance Data

No. of square regions found at end of split stage = 1732
No. of regions found at end of merge stage = 11

	Split Stage		Merge Stage (Random Tie Break)	
	Time (sec)	Iterations	Time (sec)	Iterations
CM Fortran	0.361	4	42.570	27
F77 + CMMD (LP)	0.022	4	9.454	33
F77 + CMMD (NS-BR)	0.021	4	5.516	28

Table 7.5: Image 4 Performance Data

No. of square regions found at end of split stage = 823
No. of regions found at end of merge stage = 2

	Split Stage		Merge Stage (Random Tie Break)	
	Time (sec)	Iterations	Time (sec)	Iterations
CM Fortran	2.052	5	37.588	25
F77 + CMMD (LP)	0.097	5	16.512	37
F77 + CMMD (NS-BR)	0.097	5	10.942	29

Table 7.6: Image 5 Performance Data

No. of square regions found at end of split stage = 298
No. of regions found at end of merge stage = 7

	Split Stage		Merge Stage (Random Tie Break)	
	Time (sec)	Iterations	Time (sec)	Iterations
CM Fortran	2.046	5	24.471	16
F77 + CMMD (LP)	0.099	5	14.388	35
F77 + CMMD (NS-BR)	0.098	5	6.640	35

Table 7.7: Image 6 Performance Data

No. of square regions found at end of split stage = 2248
No. of regions found at end of merge stage = 4

	Split Stage		Merge Stage (Random Tie Break)	
	Time (sec)	Iterations	Time (sec)	Iterations
CM Fortran	2.066	5	75.582	45
F77 + CMMD (LP)	0.098	5	12.192	36
F77 + CMMD (NS-BR)	0.098	5	7.236	38

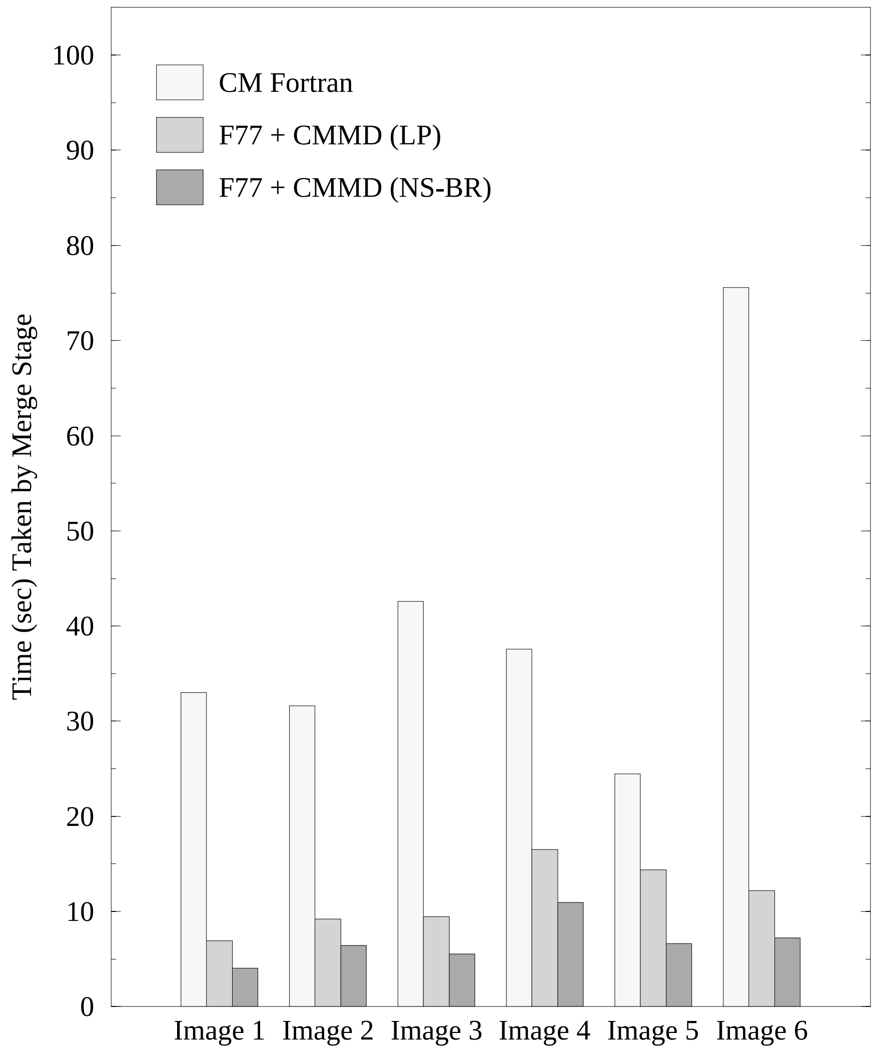


Figure 7.10: Execution Time of the Merge Stage on the CM-5 (32 Nodes)

Chapter 8

Language and Runtime Support for Hierarchical Clustering

In Chapters 6 and 7, we applied the multigraph-based paradigm for hierarchical clustering to the region growing problem and compared two implementations of that problem on the CM-5: (a) a message passing implementation written in Fortran 77 plus CMMD and (b) a data parallel implementation written in CM Fortran.

The timing figures in Tables 7.2 – 7.7 show that in both implementations most of the execution time is spent in the merge stage of the program, where the graph data structure is used. For example, in the case of Image 1, the merge stage in the message passing implementation (using the *Non-Blocking Send, Blocking Receive* (NS-BR) communication scheme) takes about 190 times longer to execute than the split stage. That number is about 90 in the data parallel implementation.

The timing figures also show that the message passing implementation runs significantly faster than the data parallel one. For example, in the case of Image 1, the merge stage in the message passing implementation (using the *Non-Blocking*

Send, Blocking Receive communication scheme) is 8 times faster than the merge stage in the data parallel implementation. In the case of Image 6, the merge stage in the message passing implementation is almost 10 times faster than the merge stage in the data parallel implementation.

However, the data parallel implementation is easier to program than the message passing one. In the message passing implementation the programmer must explicitly specify synchronization, distribute the data, and manage data communication among the processors; while in the data parallel implementation the compiler and the runtime system insert synchronization, lay out the data, and provide communication among the node processors. This is evident in a comparison of the number of lines of source code in the two implementations: 2,525 lines in the message passing implementation, versus 1,128 in the CM Fortran implementation.

The poor performance of the data parallel implementation in comparison to the message passing one prompts us to ask the following questions:

1. What are the main causes of the poor performance of the data parallel implementation?

2. What language and runtime support is needed for the efficient execution of region growing in particular, and hierarchical clustering applications in general, on distributed memory machines?

In this chapter we examine the following issues as they relate to hierarchical clustering applications and propose enhancements to HPF that support the execution of hierarchical clustering applications on distributed memory machines:

1. Data distribution

2. Data redistribution

3. Data communication

4. Processors

5. Library procedures

6. Data structures

7. High-level routines for hierarchical clustering

8.1 Data Distribution

The aim of data distribution is to divide the data among the processors in such a fashion that data communication is minimized and the computational load is as balanced as possible.

In region growing, as in other hierarchical clustering applications, there are two main sets of data that are manipulated by the program. The first is the set of objects and their attributes that are input to the program. The second is the graph data structure built by the program and used to compute the required clustering. For example, in region growing there is the two-dimensional array of pixel intensities that is input to the program, and there is the graph data structure that is composed of one-dimensional vertex and edge arrays that model the problem.

In region growing, as in other hierarchical clustering applications, the input data are used only at the beginning of the program to build the graph modeling the problem and at the end of the program to display the results. During most of the execution time of the program the graph data structure is used. Thus it is not worthwhile to spend much effort on distributing the input data to achieve computational load balance. Rather, it is more worthwhile to focus on distributing the graph data structure.

8.1.1 Distributing the Input Data

In both the message passing and the data parallel implementations of region growing, the two-dimensional array of pixel intensities is distributed using the (BLOCK, BLOCK) distribution scheme. In this scheme the two-dimensional array is divided into contiguous blocks of equal size such that adjacent blocks in the image are stored in adjacent processors in the processor grid. The (BLOCK, BLOCK) scheme maintains locality of data accesses so that no communication is required during the split stage, and only near-neighbor communication is required when setting up the vertices and edges of the graph modeling the problem.

However, while the (BLOCK, BLOCK) distribution scheme maintains data locality, it does not necessarily balance the computational load. A different number of square regions is generally associated with each block. Since the square regions in the image are only known at runtime after the execution of the split stage, and since the time taken by split stage is a small fraction of the overall time of the program, it follows that the (BLOCK, BLOCK) distribution scheme is adequate.

8.1.2 Distributing the Graph Data Structure

A graph data structure is composed of a set of vertices and a set of edges connecting these vertices. The edges may connect the vertices in any arbitrary fashion; the only restriction is that an edge must connect exactly two distinct vertices. Most "interactions" are between an edge and the two vertices that it connects. For example, in region growing, one merge iteration requires a loop over all the active edges so that each vertex of the graph can select the best neighbor to merge with.

When mapping such a graph data structure to a distributed memory machine, the aim is to minimize inter-processor communication and balance the computational load. Communication is minimized by mapping vertices of the graph and edges connecting these vertices to the same processor as much as possible, thus minimizing the number of cross-edges between processors. The computational load is balanced by mapping an approximately equal number of vertices and edges to every processor.

Finding a data distribution that minimizes inter-processor communication and balances the computational load is equivalent to solving the graph partitioning problem. Garey and Johnson [46] provide the following definition of the decision problem of graph partitioning:

> **INSTANCE:** Graph $G = (V, E)$, weight $w(v) \in Z^+$ for each $v \in V$, and weight $d(e) \in Z^+$ for each $e \in E$, positive integers K and J.
>
> **QUESTION:** Is there a partition of V into disjoint sets $V_1, V_2, \cdots, V_k$ such that $\sum_{v \in V_i} w(v) \leq K$ for $1 \leq i \leq m$ and such that if $E' \subseteq E$ is the set of edges that have their endpoints in two different sets V_i, then $\sum_{e \in E'} d(e) \leq J$?

The graph partitioning problem is NP-complete. Instead of finding an exact solution to the problem, many approaches attempt to give a good but not necessarily optimal solution. These approaches include methods based on the Kernighan and Lin [66] graph partitioning algorithm, simulated annealing [67], recursive bisection methods [88, 106], genetic algorithms [59, 72], and linear programming [76].

In hierarchical clustering, the shape of the graph data structure modeling an application is not known in advance. Therefore, decisions about distributing the vertex and edge arrays have to be made at runtime. The arrays can be initially declared to be distributed by some simple scheme such as BLOCK and then redistributed by some more suitable scheme as soon as the data is available.

In the message passing implementation of region growing, the vertex and edge arrays are distributed by BLOCK, but the number of elements stored in each block is different from one processor to another. This, in effect, is equivalent to a GENERAL_BLOCK distribution where possibly different-sized blocks of the arrays are assigned to different processors. This scheme is used by the message passing implementation because it yields only case (a) and case (b) mappings (see Figure 7.4) of vertices and edges of the graph. While this scheme may not distribute the computational load evenly among the processors, it reduces the overhead of communication.

In the data parallel implementation of region growing, on the other hand, the vertex and edge arrays are distributed by BLOCK among the processors. Equal-sized blocks of the arrays are assigned to every processor without paying attention to the underlying graph structure. Although the BLOCK distribution scheme divides the computational load more evenly among the processors, case (c) mappings (see Figure 7.4) will generally arise and lead to increased communication.

The current version HPF supports only the BLOCK, CYCLIC, and BLOCK CYCLIC distribution schemes. As the message passing implementation of region growing demonstrates, hierarchical clustering applications would benefit from extending HPF to support GENERAL_BLOCK distributions. Figure 8.1 shows example specifications of GENERAL_BLOCK distributions in HPF. In the figure, the one-dimensional arrays VERT, EDGE1, and EDGE2 are distributed in a GENERAL_BLOCK fashion onto an array of Q processors. The SIZE array S1 is used to specify the size of each block of the array VERT in every processor. Since there are Q processors, S1 is of size Q, and the sum of the elements of S1 must be equal to N, the size of the array VERT. Similarly, array S2 is used to specify the size of each block of the arrays EDGE1 and EDGE2 in every processor. Arrays S1 and S2 themselves may be distributed in BLOCK fashion or may be replicated in each processor if Q is small. The values of the elements of the SIZE arrays, S1 and S2, can either be supplied by the programmer or computed by a partitioner.

```
      INTEGER, DIMENSION(N) :: VERT
      INTEGER, DIMENSION(M) :: EDGE1, EDGE2
      INTEGER, DIMENSION(Q) :: S1, S2

!HPF$ PROCESSORS P(Q)
!HPF$ DISTRIBUTE (BLOCK IRREGULAR), SIZE (S1) :: VERT
!HPF$ DISTRIBUTE (BLOCK IRREGULAR), SIZE (S2) :: EDGE1, EDGE2
!HPF$ ALIGN EDGE2 (I) WITH EDGE1 (I)

      ...
```

Figure 8.1: Specifying a GENERAL_BLOCK Distribution

Besides allowing the GENERAL_BLOCK distribution scheme, there are several approaches to extending a data parallel language to support irregular problems. We discuss a few of these approaches below and their applicability to hierarchical clustering applications:

1. **Mapping Arrays:** A mapping array (or translation table) maps the elements of a data array to processors in the abstract processor array. Mapping arrays can be used to specify arbitrary distributions that do not require replication of array elements. Fortran D [40] and the CHAOS system [80] provide mechanisms for using mapping arrays. Chapman et al. [27] propose the use of mapping arrays in HPF. The values of the mapping array can be supplied by the programmer or computed by a partitioner that the program invokes. Figure 8.2 illustrates the use of a mapping array. In the figure, the mapping array, MAP_VERT, is used to map the i'th element of VERT to processor P(MAP_VERT(i)). Similarly, the mapping array, MAP_EDGE is used to map the i'th element of array EDGE1 (or EDGE2) to processor P(MAP_EDGE(i)).

2. **User-Defined Distribution Functions (UDDFs):** A UDDF [27] also maps elements of a data array to an abstract processor array, but it is more general than using a mapping array because it allows the replication of data elements across processors. Syntactically, a UDDF is similar to a Fortran function, but its activation results in the computation and execution of a distribution. The keywords TARGET_ARRAY and PROCESSOR_ARRAY are used to specify the array to be distributed and the abstract processor array used in the distribution, respectively. A UDDF may include declarations of local data structures and Fortran executable statements. It must

contain at least one distribution statement that maps the elements of the target array to the processors. Figure 8.3 illustrates the use of a user-defined distribution function called EXAMPLE.

3. **Reorder Arrays:** The CHAOS system [81] proposes the use of *reorder arrays* to embed an irregular mapping in a regular mapping (such as BLOCK), thus eliminating the need for mapping arrays. A reorder array is used to reorder the elements of a vertex array and renumber the elements of edge arrays. Figure 8.4 illustrates this approach. Initially, arrays VERT, EDGE1, EDGE2, and REORDER are distributed by BLOCK. Next, a graph partitioner is called to obtain the reorder array, REORDER, which is used to reorder the elements of the vertex array, VERT, such that VERT(I) is moved to the position REORDER(I). Then array REORDER is used to update the values of the edge arrays, EDGE1 and EDGE2, such that EDGE1(I) is modified to REORDER(EDGE1(I)) and EDGE2(I) is modified to REORDER(EDGE2(I)).

4. **Value-Based Mappings:** In a value-based mapping [55, 56], an array X is distributed according to the values of its elements, rather than the indices of its elements. Two elements of X that are close together in value are assigned to the same processor. Figure 8.5 shows an example of using a value-based mapping. Initially, arrays VERT, EDGE1, EDGE2 are distributed by BLOCK. After the values of array VERT are input, the array is distributed according to the values stored in the array. That is, the values of VERT determine the mapping of the elements of the array onto the processors. Next, the edge arrays EDGE1 and EDGE2 are aligned by value with VERT. That is, element EDGE1(I) is mapped to the same proces-

sor to which VERT(EDGE1(I)) is mapped. Similarly, element EDGE2(I) is mapped to the same processor to which VERT(EDGE2(I)) is mapped. Since edges tend to connect vertices that are close to each other, it is expected that EDGE1(I) and EDGE2(I) are mapped to the same processor. In case of a conflict, heuristics can be used to find the best mapping of the edge.

```
       INTEGER, DIMENSION(N) :: VERT, MAP_VERT
       INTEGER, DIMENSION(M) :: EDGE1, EDGE2, MAP_EDGE

!HPF$ PROCESSORS P(Q)
C
!HPF$ DISTRIBUTE (BLOCK) :: VERT
!HPF$ ALIGN MAP_VERT (I) WITH VERT (I)
C
!HPF$ DISTRIBUTE (BLOCK) :: EDGE1
!HPF$ ALIGN EDGE2 (I) WITH EDGE1 (I)
!HPF$ ALIGN MAP_EDGE (I) WITH EDGE1 (I)

C      Compute a new distribution for A and save it in the
C      mapping array MAP.  The i'th element of A is mapped to
C      the processor whose number is stored in MAP(i), i.e.
C      to processor P(MAP(i)).

       CALL PARTITIONER (VERT, EDGE1, EDG2, MAP_VERT, MAP_EDGE)
       ...


C      Redistribute VERT as specified by the mapping array MAP_VERT.

!HPF$  REDISTRIBUTE VERT (MAP (MAP_VERT))
       ...


C      Redistribute EDGE1 and EDGE2 as specified by the mapping array
C      MAP_EDGE.

!HPF$  REDISTRIBUTE EDGE1 (MAP (MAP_EDGE))
!HPF$  REDISTRIBUTE EDGE2 (MAP (MAP_EDGE))
       ...
```

Figure 8.2: An Example of a Mapping Array

```
!HPF$ DFUNCTION EXAMPLE
!HPF$ TARGET_ARRAY A(:)
!HPF$ PROCESSOR_ARRAY P(:)

      DO I = 1, SIZE(A)
         DISTRIBUTE A(I) TO P( ..some function of I.. )
      END DO
!HPF$ END DFUNCTION EXAMPLE
```

Figure 8.3: An Example of a User-Defined Distribution Function (UDDF)

```
       INTEGER, DIMENSION(N) :: VERT
       INTEGER, DIMENSION(M) :: EDGE1, EDGE2
       INTEGER, DIMENSION(:) :: REORDER

!HPF$ PROCESSORS P(Q)
!HPF$ DISTRIBUTE (BLOCK) :: VERT
!HPF$ DISTRIBUTE (BLOCK) :: EDGE1
!HPF$ ALIGN EDGE2 (I) WITH EDGE1 (I)

C      Call partitoner to obtain array REORDER.

       CALL PARTITIONER (..., VERT, EDGE1, EDGE2, REORDER ...)

       ...

C      Use array REORDER to reorder the vertices.
C      In reordering, the value of VERT(I) is moved
C      to the position REORDER(I) in the array.

       CALL REORDER_VERTICES (REORDER, VERT)


C      Use array REORDER to renumber the edge arrays.  After
C      renumbering is complete, EDGE1(I) is modified to
C      REORDER(EDGE1(I)) and EDGE2(I) is modified to
C      REORDER(EDGE2(I)).

       CALL RENUMBER_EDGES (REORDER, EDGE1)
       CALL RENUMBER_EDGES (REORDER, EDGE2)

       ...
```

Figure 8.4: An Example of a Reorder Array

```
       INTEGER, DIMENSION(N) :: VERT
       INTEGER, DIMENSION(M) :: EDGE1, EDGE2

!HPF$ PROCESSORS P(Q)
!HPF$ DISTRIBUTE (BLOCK) :: VERT
!HPF$ DISTRIBUTE (BLOCK) :: EDGE1
!HPF$ ALIGN EDGE2 (I) WITH EDGE1 (I)


C      Input the data describing the vertices and store the values
       in array VERT.

       CALL INPUT (VERT)


C      Distribute VERT according to values stores in the array.

!HPF$ DISTRIBUTE_BY_VALUE VERT


C      Distribute EDGE1 and EDGE2 according to the values stored
C      in VERT.

!HPF$ REALIGN_BY_VALUE EDGE1 WITH VERT
!HPF$ REALIGN_BY_VALUE EDGE2 WITH VERT

       ...
```

Figure 8.5: An Example of a Value-Based Mapping

The mapping-array and UDDF approaches suffer from the overhead of accessing a translation table or computing a mapping function every time an element of a distributed array is to be accessed. This cost may be somewhat reduced by re-using communication schedules, but often this is not possible for hierarchical clustering applications where the clusters change rapidly.

The reorder-array and value-based mapping approaches, on the other hand, eliminate the need to access a translation table every time an element of a distributed is to be accessed, and are, therefore, more suitable for hierarchical clustering applications. The reorder-array approach would be identical to the value-based mapping approach when the partitioner called to compute the REORDER array returns a value-based reordering of VERT.

8.2 Data Redistribution

The graph modeling a hierarchical clustering application constantly evolves at run-time. With each merge iteration, pairs of vertices merge into one larger vertex and cause the graph to contract. As the merge stage progresses, the initial distribution of the vertex and edge arrays will no longer be adequate for minimizing communication or for balancing the computational load. Dynamic redistribution of the vertex and edge arrays among the processors will be needed.

Obviously, the redistribution of the arrays carries with it a communication and computation overhead. The best algorithm to use and the ideal frequency of redistribution depend on the input data.

When the changes in the graph in each merge iteration are small and isolated, it is advantageous to use local methods such as incremental graph partitioning [76] to redistribute the data. Local methods use only information obtained from a small number of neighboring processors to balance the load. For example, in the case of a two-dimensional array of processors, every processor periodically compares its load to that of its four neighbors and transfers computation if its load exceeds a certain threshold.

On the other hand, when the graph changes rapidly because many pairs of vertices merge together in each merge iteration, it may be advantageous to use global methods that take longer to execute but produce better mappings. Examples of global methods include recursive graph bisection and recursive spectral bisection [39].

In the case of region growing, the graph changed rapidly in all of the test cases. For example, in the case of Image 1, the graph initially has 439 vertices that diminish to 2 vertices after about 20 iterations. This implies that, on the average, about 20 pairs of vertices merge in one merge iteration. In the case of Image 3, this number is even greater. The graph initially has 1,732 vertices that diminish to 11 vertices after about 28 iterations and, on the average, about 60 pairs of vertices merge per merge iteration.

HPF provides the REDISTRIBUTE and REALIGN directives for distributing data at runtime. These directives can effectively be used for distributing the arrays when combined with the facility to specify irregular distributions.

8.3 Unstructured Communication

In hierarchical clustering applications most "interactions" are between an edge and the two vertices that it connects. However, the edge and the two vertices may lie in relatively distant processors. Clustering applications therefore possess both local as well as global features.

We identify the following different types of unstructured communication in the hierarchical clustering paradigm:

1. Unstructured communication using FORALL with left-hand-side indirection (values of the indirection array must be unique).

2. Unstructured communication using FORALL with right-hand-side indirection.

3. A combination of 1 and 2 above.

4. Parallel reduction using MINVAL_SCATTER:

   MINVAL_SCATTER (ARRAY, BASE, INDX, MASK)

   The elements of ARRAY selected by MASK are scattered to positions indicated by an index array INDX. Each element of the result is assigned the minimum value of the corresponding element of BASE and the elements of ARRAY scattered to that position.

In the case of region growing, one merge iteration requires a loop over all the active edges of the graph so that each vertex of the graph can select the best neighbor to merge with. In this loop, edge arrays are used as indirection arrays to access elements of the vertex arrays. Since the data stored in the edge arrays is only known at runtime, it follows that an efficient runtime system is needed to carry out the required communications.

The following optimizations have led to reduced communication costs in the message passing implementation of region growing:

1. *Message aggregation.* Sending a small number of large messages is less costly than sending a large number of small messages.

2. *The use of an efficient communication scheme.* Of the two communication schemes investigated on the CM-5, *Linear Permutation* (LP) and *Non-Blocking Send, Blocking Receive* (NS-BR), the latter is the faster one. In the NS-BR scheme a node can pursue other computation until messages are ready to be received, while in the LP scheme all the nodes must loop the same number of times until all the required sends and receives are completed. The effect of using an efficient communication scheme is clearly illustrated in the timings for Image 5. The merge stage of the message passing implementation took 14.388 seconds to execute when using LP, and only 6.64 seconds when using NS-BR.

3. *Maintaining locality of data accesses.* Communication on distributed memory machines is generally expensive relative to computation. Therefore, it is important to reduce communication by placing data objects that "interact" in the same processor as much as possible. As discussed in Section 8.1, the graph in the message passing implementation was distributed so that data locality was maintained.

Finally, we note that since the graph modeling a hierarchical clustering application changes rapidly, re-using communication schedules is often not possible.

8.4 Library Procedures

A number of operations are commonly performed by hierarchical clustering applications. We divide these operations into the following five categories:

1. Fortran 90 elemental intrinsic functions

2. Fortran 90 transformational intrinsic functions

3. Fortran 90 intrinsic subroutines

4. HPF library procedures

5. Other procedures

The last category, “Other procedures”, refers to operations not provided by Fortran 90 or HPF but that are nonetheless commonly used in hierarchical clustering.

In the following sections we describe the operations in each of the five categories.

8.4.1 Fortran 90 Elemental Intrinsic Functions

The following Fortran 90 elemental intrinsic functions are used in region growing and are useful for hierarchical clustering applications in general:

1. MIN: Returns the minimum of two or more arguments

2. MAX: Returns the maximum of two or more arguments

3. MOD: Returns the remainder of a division

4. LOG10: Returns the base-10 logarithm of a number of type real (used to compute the number of decimal digits needed to represent a value)

5. NINT: Rounds a real value to the nearest integer

6. REAL: Converts an integer value to type real

8.4.2 Fortran 90 Transformational Intrinsic Functions

The following Fortran 90 transformational intrinsic functions are used in region growing and are useful for hierarchical clustering applications in general:

1. ANY: Determines whether any element a logical array has the value TRUE. In region growing this function is used to determine whether any active edges exist in the graph.

2. PACK: Packs the elements of an array into consecutive locations in a one-dimensional array, under the control of a mask. In region growing, this function is used to transfer data from the two-dimensional pixel arrays to the one-dimensional vertex arrays. Also, this function is used to pack the elements of an array immediately before sorting it.

3. UNPACK: Unpacks the elements of an array of rank one into an array, under the control of a mask. In region growing this function is used at the end of the program to transfer the cluster labels from a one-dimensional vertex array to a two-dimensional pixel array.

4. COUNT: Counts the number of TRUE elements in a logical array along a specified dimension. In region growing this function is used to count the number of active edges in the graph.

5. MAXVAL: Returns the maximum value in an array along a specified dimension, under the control of a mask. In region growing this function is used to find the largest size of the square regions.

8.4.3 Fortran 90 Intrinsic Subroutines

The following Fortran 90 intrinsic subroutines are used in region growing and other hierarchical clustering applications when ties are broken at random:

1. RANDOM_NUMBER: Returns an array of random numbers
2. RANDOM_SEED: Restarts the pseudorandom number generator used by RANDOM_NUMBER

8.4.4 HPF Library Procedures

The following HPF library procedures are used in region growing and are useful for hierarchical clustering applications in general:

1. COPY_SCATTER: Parallel reduction. If several values are scattered to the same array position, then an arbitrary one of these values is chosen and assigned to that position. This function is used to transfer data from the two-dimensional pixel arrays to the one-dimensional vertex arrays.
2. COUNT_PREFIX: Computes a segmented COUNT scan along a specified dimension of a logical array. In region growing this function is used to assign consecutive labels to the square regions found in the pixel array.
3. GRADE_UP, GRADE_DOWN: Returns a permutation of the indices of an array, sorted by descending (ascending) array element values. This permutation can be used to sort the elements of an array. In region growing sort is used to identify duplicate values of an edge.
4. MINVAL_SCATTER: Parallel reduction. If several values are scattered to the same array position, then the minimum of these values is assigned to

that position. In region growing this function is used during the merge stage to determine which neighbor a region selects for merging.

8.4.5 Other Procedures

Other commonly used operations in hierarchical clustering that are not available in Fortran 90 or HPF are: removing duplicate values, spreading-by-section, and sorting.

Removing Duplicate Values

As the vertices of the graph modeling a clustering application merge, the edges of the graph must be updated to reflect the new relationships between the clusters. In the process of updating the edges the programmer must check that no duplicate edges occur. In the region growing example this is achieved as follows: First, the values of the two vertices that comprise an edge are concatenated together into one larger value so that an edge is represented by one composite value instead of two. These composite values are then packed into contiguous array locations and sorted. After the sort, pairs of consecutive elements are compared to check whether their values are the same. Duplicate values are deactivated and removed from the original arrays of edges. This procedure for removing duplicates is carried out at the end of every merge iteration and can be time consuming. The availability of a library procedure that allows us to identify duplicate values in an array would make the task of programming easier and more efficient.

The following is a possible interface to a library procedure called

REMOVE_DUPLICATE which returns a result of type logical.

```
REMOVE_DUPLICATE (ARRAY, MASK)

Arguments:

  ARRAY     Must be of type integer, real, or character.

  MASK      Must be of type logical and must have the same
            shape as ARRAY.


Result:

  The result is of type logical and is of the same shape as
  ARRAY.  Every value of ARRAY selected by MASK is selected
  exactly once by the result array.


Examples:

  A    =  (/ 1 1 1 1 1 /)
  MASK =  (/ T T T T T /)
  REMOVE_DUPLICATE (A, MASK)  is  (/ T F F F F /)

  A    =  (/ 1 3 4 1 4 5 3 4 2 /)
  MASK =  (/ T F T T T F T T T /)
  REMOVE_DUPLICATE (A, MASK)  is  (/ T F T F F F T F T/)
```

Spreading-by-Section

Another operation used in hierarchical clustering applications where the input data is represented in the form of an array is the "spread-by-section" operation. In region growing, for example, this operation is used at the end of the merge stage to label the pixels in the input image with the labels of the clusters to which they belong. First, every representative of a square region in the image (the top-left corner pixel) is assigned the label of the cluster to which it belongs, then that value is spread to all other pixels in the square region.

Figure 8.6 illustrates the spread-by-section operation. In part (a) of the figure, the shaded elements of the array are the elements whose values are to be spread. Associated with each of these elements is the length of the spread along each dimension, indicated by the arrows in the figure. The result of spread-by-section is shown in part (b) of the figure; the shaded areas correspond to the elements whose values have been updated after the spread.

HPF provides the COPY_PREFIX library procedure that computes a segmented copy scan along a dimension of the array. This procedure can be used to achieve the "spread-by-section" result described above. However, COPY_PREFIX requires the use of logical segment arrays which must be constructed by the programmer. It also spreads values along one dimension of the array only, and therefore it must be called more than once to spread values along more than one dimension.

When the sizes of the sections are available, it would be advantageous to have a fast library routine that can be called once to achieve "spread-by-section". The following is a possible interface to a library procedure called
SPREAD_BY_SECTION that achieves the required operation:

SPREAD_BY_SECTION (ARRAY, MASK, LENGTH1, ..., LENGTHn)

Arguments:

ARRAY	May be of any type.
MASK	Must be of type logical and must have the same shape as ARRAY.
LENGTH1, ..., LENGTHn	n arrays of type integer of the same shape as ARRAY. The number of LENGTH arguments must be equal to the rank of ARRAY.

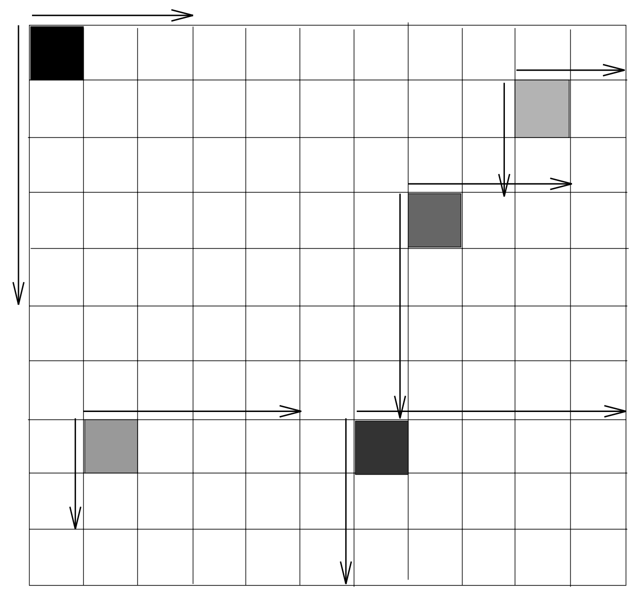
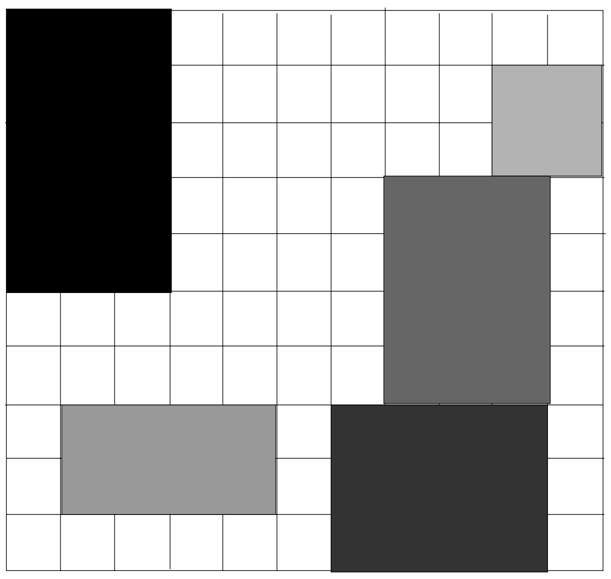
Result:

The result is of the same type and shape as ARRAY. Every value of ARRAY selected by MASK is spread along dimension i to the number of elements specified by the corresponding element of.

Examples:

```
A    =  (/ 1 1 1 1 1 /)
MASK =  (/ T T T T T /)
REMOVE_DUPLICATE (A, MASK)  is  (/ T F F F F /)

A    =  (/ 1 3 4 1 4 5 3 4 2 /)
MASK =  (/ T F T T T F T T T /)
REMOVE_DUPLICATE (A, MASK)  is  (/ T F T F F F T F T/)
```

(a) Array elements before spread;   (b) Array elements after spread

Figure 8.6: An illustration of Spread-by-Section

Sort

HPF provides the GRADE_UP and GRADE_DOWN library procedures that return a permutation of the indices of an array, sorted by ascending and descending array element values, respectively. While a permutation of the indices of an array is useful for sorting, it would be more convenient to have a library procedure that actually sorts the elements of the array.

The following is a possible interface to a library procedure SORT_UP that sorts, in ascending order, elements of an array selected by mask. A similar interface can be defined for SORT_DOWN.

```
SORT_UP (ARRAY, MASK)

Arguments:

  ARRAY      May be of any type.

  MASK       Must be of type logical and must have the same
             shape as ARRAY.


Result:

  The result is of the same type and shape as ARRAY.  The values
  stored in the result array in positions selected by MASK are the
  values of ARRAY selected by MASK and sorted in ascending order.

Examples:

  A    =  (/ 8 7 6 3 2 1 /)
  MASK =  (/ T T T T T T /)
  SORT_UP (A, MASK)  is  (/ 1 2 3 4 5 6 7 8/)

  A    =  (/ 8 7 6 3 2 1 /)
  MASK =  (/ T F T F T F /)
  SORT_UP (A, MASK)  is  (/ 2 7 6 3 8 1 /)
```

8.5 Processors

Hierarchical clustering applications exhibit a dynamic behavior that starts with a high degree of parallelism that very rapidly diminishes to a much lower degree of parallelism. As the graph modeling the problem becomes smaller in size, the initial number of processors assigned to the problem may no longer be required. To reduce the overhead of communication and release unwanted resources, it may be preferable to use a smaller number of physical processors to execute the later iterations of the merge stage. This would require re-distributing the data in the program onto a smaller number of physical processors.

In the case of CM Fortran on the CM-5, the program is automatically assigned all the physical processors in the partition. In the case of HPF on the DEC Alpha cluster, the programmer specifies the number of physical processors using a compilation flag, and this number cannot be changed while the program is executing. It would be useful to provide in HPF the facility to change the number of physical processors assigned to a program while the program is executing.

The following example illustrates the use of a new HPF directive, PHYSICAL_PROCESSORS, that allows the programmer to specify and modify the number of physical processors assigned to a program, from within the program itself. As usual, the PROCESSORS directive specifies the number of abstract processors.

```
       INTEGER, DIMENSION(1024), DYNAMIC :: A

!HPF$  PHYSICAL_PROCESSORS 32              !  32 physical processors
!HPF$  PROCESSORS P(128)                   ! 128 abstract processors
!HPF$  DISTRIBUTE A (BLOCK) ONTO P

...

!HPF$  PHYSICAL_PROCESSORS 4               !  4 physical processors
!HPF$  PROCESSORS P(16)                    ! 16 abstract processors
!HPF$  REDISTRIBUTE A (BLOCK) ONTO P

...
```

Besides the ability to re-distribute data onto a smaller number of physical processors, it would also be useful to allow different views of the same set of abstract processors. In the current version of HPF, a one-dimensional array must be distributed over a one-dimensional processor arrangement, and a d-dimensional array must be distributed over a d-dimensional processor arrangement ($d \geq 2$). However, it would be useful to relax this requirement and allow a one-dimensional array to be distributed over a d-dimensional processor arrangement ($d \geq 1$). In region growing, for example, it would be more "natural" to view the edge and vertex arrays as distributed over the same two-dimensional processor arrangement as the pixel image.

8.6 Data Structures

In the message passing and data parallel implementations of region growing, only one-dimensional arrays are used to represent the graph structure. However, it would be more natural to represent a graph data structure using a linked-list, as follows:

```
Neighbors: Node 1 --> Node 2 --> Node 4 --> Node 5 --> o

           Node 2 --> Node 1 --> o

           Node 3 --> Node 4 --> o

           Node 4 --> Node 1 --> Node 3 --> o

           Node 5 --> Node 1 --> o
```

It would also be more natural to represent a hierarchy of clusterings using a tree.

Although Fortran 90 and HPF allow the definition of *derived types* (data structures), the current version of HPF does not allow the distribution of individual components of a derived type; it allows only the distribution of arrays of derived types. Extending HPF with the facility to distribute derived types would allow programmers to use more sophisticated data structures and write programs at a more abstract level.

8.7 Library Routines for Hierarchical Clustering

In Section 5.3 we identified a number of high-level routines that implement the various steps of the hierarchical clustering paradigm:

1. CLUSTER
2. PREPROCESS
3. CONSTRUCT_GRAPH

    (a) INITIALIZE_VERTEX_ARRAYS

    (b) INITIALIZE_EDGE_ARRAYS

4. MERGE_VERTICES

    (a) MATCH_VERTICES

    (b) UPDATE_GRAPH

        i. UPDATE_VERTICES

        ii. UPDATE_EDGES

5. UPDATE_LABELS

To facilitate the task of programming, it would be useful to provide skeletons that can be customized for particular clustering applications. Moreover, it would be useful to provide a library of procedures for implementing particular clustering applications, including procedures for various graph operations.

Chapter 9

Summary and Conclusions

This dissertation examined particular types of loosely synchronous irregular problems known as hierarchical clustering applications, and identified the language and runtime support needed for their efficient execution on distributed memory machines.

The steps in identifying the language and runtime support were as follows:

Starting with a definition of hierarchical clustering, a graph-theoretic formulation of hierarchical clustering using the multigraph concept was presented. This formulation was used to describe a programming paradigm for hierarchical clustering. The paradigm was applied to a representative clustering application, region growing, and was implemented on a distributed memory machine in both the data parallel and message passing models. The following three implementations of region growing were examined:

1. A message passing implementation written in Fortran 77 plus CMMD, executed on a 32-node CM-5

2. A data parallel implementation written in CM Fortran language, executed

on a 32-node CM-5

3. A data parallel implementation written in High Performance Fortran, executed on a cluster of DEC Alpha workstations

A comparison of the message passing and data parallel implementations on the CM-5 reveals that the former is significantly faster than the latter, although the computations carried out in both implementations were identical. The main differences between the two implementations are related to the following:

1. The distribution of the graph modeling the problem among the processing nodes.

2. The schemes or algorithms used to carry out the required inter-processor communication at runtime.

A close examination of the parallel implementations led us to identify the language and runtime support needed for the execution of clustering applications on distributed memory machines. The findings can be summarized as follows:

1. **Data distribution**: Efficient distribution schemes are needed to distribute the graph modeling a hierarchical clustering application. Suitable distribution schemes include: GENERAL_BLOCK, use of reorder arrays, and value-based mappings.

2. **Data redistribution**: Since the graph modeling a hierarchical clustering application constantly contracts at runtime, dynamic redistribution of data may be necessary. Local load balancing algorithms would be suitable for graphs that evolve slowly, while global algorithms would suitable for graphs that evolve rapidly.

3. **Data communication**: Clustering applications have irregular data access patterns that are known only at runtime, and thus efforts should be focused on developing efficient routines for unstructured communication. The use of message aggregation and non-blocking send have been shown to improve performance.

4. **Library Procedures**: Hierarchical clustering applications use a variety of Fortran 90 and HPF library procedures, but they employ some commonly used operations that are not provided by the Fortran 90 or HPF libraries. We proposed that the following library procedures be made available: REMOVE_DUPLICATE, SPREAD_BY_SECTION, and SORT.

5. **Processors**: As the graph modeling a hierarchical clustering application becomes smaller in size, the initial number of processors assigned to the problem may no longer be required. We proposed that a directive for specifying the number of physical processors be included in HPF.

6. **Data Structures:** Extending HPF with the facility to distribute derived types would allow programmers to use more sophisticated data structures and write programs at a more abstract level.

7. **High-Level Routines**: We identified a number of high-level routines that implement the various steps of the hierarchical clustering paradigm. To facilitate the task of programming, it would be useful to provide skeletons that can be customized for particular clustering applications. Moreover, it would be useful to provide a library of procedures for implementing particular clustering applications, including procedures for various graph operations.

The above findings emphasize the need for the following:

- High-level programming parallel languages that hide the details of the architecture from the programmer, thereby facilitating the task of parallel programming.

- Advanced compiler technology that can detect parallelism and produce an efficient object code.

- Runtime support libraries that can be used by compilers for distributed memory machines.

- High-level parallel programming paradigms the programmer can use as building blocks in solving certain types of problems.

- A library of (graph) procedures for implementing particular clustering applications.

We hope that this work will motivate the High Performance Fortran Forum, as well as designers and developers of other parallel languages, to consider the support for irregular problems identified in this study.

Bibliography

[1] M. S. Aldenderfer and R. K. Blashfield , "Computer Programs for Performing Hierarchical Cluster Analysis," *Applied Psychological Measurement*, Vol. 2, No. 3, pp. 403-411, Summer 1978.

[2] M. R. Anderberg, *Cluster Analysis For Applications*, Academic Press, New York, 1973.

[3] H. Alnuweiri and V. Prasanna, "Parallel Architectures and Algorithms for Image Component Labeling," *IEEE Trans. Patt. Anal. Machine Intell.*, Vol. 14, pp. 1014-1034, 1992.

[4] Archetypes "Electronic Textbook."
URL: http://www.etext.caltech.edu

[5] D. A. Bader, J. Ja'Ja', D. Harwood, and L. S. Davis, "Parallel Algorithms for Image Enhancement and Segmentation by Region Growing with an Experimental Study," Technical Report, Institute for Advanced Computer Studies, University of Maryland, May 1995.

[6] D. Ballard and C. Brown, *Computer Vision*, Prentice Hall, Englewood Cliffs, NJ, 1982.

[7] K. D. Bailey, "Cluster Analysis," in *Sociological Methodology 1975*, edited by D. R. Heise, Jossey-Bass Inc. Publishers, San Francisco, pp. 59-128, 1974.

[8] C. F. Baille and P. Coddington, "Cluster Identification Algorithms for Spin Models – Sequential and Parallel," *Concurrency: Practice and Experience*, Vol. 3, No. 2m, April 1991, pp. 129-144.

[9] R. Barrett et al, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, 1994.
URL: http://www.netlib.org/templates/templates.ps

[10] K. Binder and D.W. Heermann, *Monte Carlo Simulation in Statistical Physics*, Springer-Verlag, Berlin, 1988.

[11] R. K. Blashfield and M. S. Aldenderfer, "Computer Programs for Performing Iterative Partitioning Cluster Analysis," *Applied Psychological Measurement*, Vol. 2, No 4, pp. 533-541, Fall 1978.

[12] R. K. Blashfield and M. S. Aldenderfer, "The Literature on Cluster Analysis," *Multivariate Behavioral Research*, Vol. 13, pp. 271-295, 1978.

[13] R. K. Blashfield, "The Growth of Cluster Analysis: Toyron, Ward, and Johnson," *Multivariate Behavioral Research*, Vol. 15, pp. 439-458, 1980.

[14] R. K. Blashfield, M. S. Aldenderfer, and L. C. Morey, "Cluster Analysis Software," in *Handbook of Statistics, Volume 2: Classification, Pattern Recognition, and Reduction of Dimensionality*, edited by P. R. Krishnaiah and L. N. Kanal, North Holland, New York, pp. 245-265, 1982.

[15] F. Bodin, P. Beckman, D. Gannon, S. Narayana, S. X. Yang, "Distributed pC++: Basic Ideas for an Object Parallel Language," *Scientific Programming*, Vol. 2, No. 3, 1993.

[16] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, B. Mohr, "Implementing a Parallel C++ Runtime System for Scalable Parallel Systems," *Proc. Supercomputing '93*, November 1993.

[17] J. A. Bondy and U. S. R. Murty, *Graph Theory With Applications*, North Holland, New York, 1976.

[18] Z. Bozkus, *Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers*, Ph.D. Dissertation, Syracuse University, Syracuse, New York, 1995.

[19] P. Brinch Hansen, "The Joyce Language Report," *Software - Practice and Experience*, Vol. 19, pp. 553-578, June 1989.

[20] P. Brinch Hansen, "Model Programs for Computational Science: A Programming Methodology for Multicompters," *Concurrency: Practice and Experience*, Vol. 5, No. 5, pp. 407-423, 1993.

[21] P. Brinch Hansen, "The Programming Language SuperPascal," *Software - Practice and Experience*, Vol. 24, pp. 467-483, May 1994.

[22] P. Brinch Hansen *Studies in Computational Science: Parallel Programming Paradigms*, Prentice Hall, Englewood Cliffs, NJ, 1995.

[23] K. R. Castleman, *Digital Image Processing*, Prentice Hall, Englewood Cliffs, New Jersey, 1979.

[24] K. M. Chandy and C. Kesselman, "CC++: A Declarative Concurrent Object Oriented Notation," in *Research in Object Oriented Programming*, MIT Press, 1993.

[25] K. M. Chandy, "Concurrent Program Archetypes," Department of Computer Science, California Institute of Technology, Pasadena, California.
URL: http://www.etext.caltech.edu/Papers2/ArchetypeOverview/ArchPaper.html

[26] B. Chapman, P. Mehrotra, and H. Zima, "Programming in Vienna Fortran," *Scientific Programming.* Vol. 1. No. 1, pp. 31-50, 1992.

[27] B. Chapman, P. Mehrotra, and H. Zima, "Extending HPF for Advanced Data Parallel Applications," Technical Report TR94-7, Institute for Software Technology and Parallel Systems, University of Vienna, Austria, 1994.

[28] P. D. Coddington, and C. F. Baille, "Cluster Algorithms for Spin Models on MIMD Computers," in *The Fifth Distributed Memory Computing Conference, Volume I*, edited by D. W. Walker W. F. Stout, IEEE Computer Society Press, Los Alamitos, California, 1990.

[29] N. Copty, S. Ranka, G. Fox, and R. Shankar, "Data Parallel Algorithms for Solving the Region Growing Problem on the Connection Machine," *Journal of Parallel and Distributed Computing, Special Issue on Data Parallel Languages and Programming*, Vol. 21, No. 1, pp. 160-168, April 1994.

[30] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*, The MIT Press, Cambridge, Massachusetts, 1989.

[31] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill, New York, 1990.

[32] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*, John Wiley & Sons, New York, 1973.

[33] B. S. Duran and P. L. Odell, "Cluster Analysis: A Survey," Lecture Notes in Economics and Mathematical Systems, No. 100, Springer-Verlag, New York, 1974.

[34] J. W. Essam, *Reports on Progress in Physics*, Vol. 43, p. 830, 1980.

[35] S. Even, *Graph Algorithms*, Computer Science Press, Potomac, MD, 1979.

[36] J. L. Fleiss and J. Zubin, "On the Methods and Theory of Clustering," *Multivariate Behavioral Research*, Vol. 4, No. 2, pp. 235-250, April 1969.

[37] Applied Parallel Research (APR) home page.
URL: http://www.infomall.org/apri

[38] I. Foster and K. M. Chandy, "Fortran M: A Language for Modular Parallel Programming," *Journal of Parallel and Distributed Computing* , 1994.

[39] I. Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995.

[40] G. C. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, M. Wu, "The Fortran D Language Specification," Technical Report CRPC-TR90079, Center for Research on Parallel Computation, Rice University, Houston, Texas, December 1990 (Revised April 1991).

[41] G. Fox, "The Architecture of Problems and Portable Parallel Software Systems," Technical Report SCCS-134, Northeast Parallel Architectures Center, Syracuse University, 1991.
URL: http://www.npac.syr.edu/techreports/html/0100/abs-0134.html

[42] G. Fox et al, "Software Support for Irregular and Loosely Synchronous Problems," Technical Report, Northeast Parallel Architectures Center, Syracuse University, May 1992.

[43] G. Fox and P. D. Coddington, "An Overview of High performance Computing for the Physical Sciences," Technical Report SCCS-488, Northeast Parallel Architectures Center, Syracuse University, 1993.
URL: http://www.npac.syr.edu/techreports/html/0450/abs-0488.html

[44] G. C. Fox, R. D. Williams, and P. C. Messina, *Parallel Computing Works!*, Morgan Kauffmann Publishers, San Francisco, CA, 1994.

[45] K. S. Fu and J. K. Mui, "A Survey on Image Segmentation," *Pattern Recognition*, Vol. 13, pp. 3-16, 1981.

[46] M. R. Garey and D. S. Johnson, *A Guide to Computer Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, 1979.

[47] J. S. Greenfield, *Distributed Programming Paradigms With Cryptography Applications*, Lecture Notes in Computer Science, Vol. 870, Springer-Verlag, NY, 1994.

[48] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A Users Guide and Tutorial for Network Parallel Computing*, MIT Press, 1994.

[49] E. Godehardt, *Graphs as Structural Models: The Application of Graphs and Multigraphs in Cluster Analysis*, Second Edition, (Volume 4 of Advances in System Analysis), Published by Vieweg, Germany, 1990.

[50] A. D. Gordon, *Classification*, Chapman and Hall, New York, 1981.

[51] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI*, MIT Press, 1994.

[52] J. A. Hartigan, *Clustering Algorithms*, John Wiley & Sons, New York, 1975.

[53] S. Hambrusch, X. He, and R. Miller, "Parallel Algorithms for Gray-Scale Digitized Picture Component Labeling on a Mesh-Connected Computer," *J. Parallel Distrib. Comput.*, to appear.

[54] A. El-Hamdouchi and P. Willett, "Comparison of Hierarchic Agglomerative Clustering Methods for Document Retrieval," *The Computer Journal*, Vol. 32, No. 3, pp. 220-227, 1989.

[55] R. V. Hanxleden, K. Kennedy, and J. Saltz, "Value-Based Distributions and Alignments in Fortran D," Technical Report CRPC-TR93365-S, Center for Research on Parallel Computation, Rice University, Houston, Texas, December 1993.

[56] R. V. Hanxleden, *Compiler Support for Machine-Independent Parallelization of Irregular Problems*, Ph.D. Dissertation, Rice University, Houston, Texas, December 1994.

[57] R. M. Haralick and L. G. Shapiro, "Image Segmentation Techniques," *Computer Vision, Graphics, and Image Processing*, Vol. 29, pp. 100-132, 1985.

[58] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Compiling Fortran D for MIMD Distributed Memory Machines," *Communications of the ACM*, Vol. 35, No. 8, pp. 66-80, August 1992.

[59] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis With Applications to Biology, Control, and Artificial Intelligence* , University of Michigan Press, Ann Arbor, Michigan, 1975.

[60] S. L. Horowitz and T. Pavlidis, "Picture Segmentation By a Directed Split-and-Merge Procedure," *Proc. 2nd International Joint Conf. on Pattern Recognition*, pp. 424-433, 1974.

[61] "High Performance Computing and Communications: Foundation for America's Information Future" (FY 1996 Blue Book), prepared by High Performance Computing, Communications, and Information Technology (HPC-CIT) Subcommittee of the National Science and Technology Council's Committee on Information and Communications (CIC). This report, which supplements the President's Fiscal Year 1996 Budget, describes the interagency High Performance Computing and Communications Program.
URL: http://www.hpcc.gov

[62] High Performance Fortran Forum, *High Performance Fortran Language Specification*, Version 1.1, Rice University, Houston Texas, November 1994.

[63] High Performance Fortran Forum, *HPF-2 Scope of Activities and Motivating Applications*, Version 0.8, Rice University, Houston Texas, November 1994.

[64] L. J. Hubert, "Some Applications of Graph Theory To Clustering," *Psychometrica*, Vol. 39, No. 3, pp. 283-309, September 1974.

[65] A. K. Jain and R. C. Dubes, *Algorithms for Clustering Data*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[66] B. Kernighan and S. Lin, “An Efficient Heuristic Procedure for Partitioning Graphs,” *Bell Systems Technical Journal*, Vol. 49, No. 2, pp. 291-308, 1970.

[67] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, “Optimization by Simulated Annealing,” *Science*, 220, pp. 671-680, 1983.

[68] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel, *The High Performance Fortran Handbook*, The MIT Press, Cambridge, Massachusetts, 1994.

[69] S. Koonin, *Computational Physics*, Benjamin-Cummings, Menlo Park, California, 1986.

[70] W. L. G. Koontz, P. M. Narendra, and K. Fukunaga, “A Graph-Theoretic Approach to Nonparametric Cluster Analysis,” *IEEE Transactions on Computers*, Vol. C-25, No. 9, pp. 936-944, September 1976.

[71] C. Leiserson, “The Network Architecture of the Connection Machine CM-5,” *Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, San Diego, CA, 1992.

[72] N. Mansour, *Physical Optimization Algorithms for mapping Data to Distributed-Memory Multiprocessors*, Ph.D. Dissertation, Syracuse University, Syracuse, New York, 1992.

[73] L. L. McQuitty, *Pattern-Analytic Clustering: Theory, Method, Research and Configural Findings*, University Press of America, New York, 1987.

[74] N. Metropolis, N. Rosenbluth, A. W. Rosenbluth, M. N. Teller, E. Teller, “Equation of State Calculations By Fast Computing Machines,” *Journal of Chemical Physics*, Vol. 21, pp. 1087-1091, 1953.

[75] O. Mouritsen, *Computer Studies of Phase Transitions and Critical Phenomena*, Springer-Verlag, Berlin, 1984.

[76] C.-W. Ou and S. Ranka, "Parallel Incremental Graph Partitioning," Technical Report, Northeast Parallel Architectures Center, Syracuse University.

[77] Parallel Compiler Runtime Consortium (PCRC).
URL: http://aldebaran.npac.syr.edu:1955/index.html

[78] Parasoft Corporation, *Express Fortran Reference Guide, Version 3.0*, 1990.

[79] T. Pavlidis, "Image Analysis," *Annual Review of Computer Science*, Vol. 3, pp. 121-146, 1988.

[80] R. Ponnusamy, J. Saltz, A. Choudhary, Y.-S. Hwang, G. Fox, "Runtime Support and Compilation Methods for User-Specified Irregular Distributions," Technical Report UMIACS-TR-93-135, Institute for Advanced Computer Studies, University of Maryland.

[81] R. Ponnusamy, Y.-S. Hwang, J. Saltz, A. Choudhary, G. Fox, "Supporting Irregular Distributions in Fortran 90D/HPF Compilers," *IEEE Transactions on Parallel and Distributed Technology*, Spring 1995.

[82] S. Ranka and S. Sahni, *Hypercube Algorithms.* Springer-Verlag, New York, 1990.

[83] S. Ranka, J. Wang, and G. Fox, "Static and Runtime Algorithms for All-To-Many Personalized Communication on Permutation Networks," *Proc. International Conf. on Parallel and Distributed Systems*, 1992.

[84] H. C. Romesburg, *Cluster Analysis for Researchers*, Lifetime Learning Publications, Belmont, California, 1984.

[85] K. Rose, *Deterministic Annealing, Clustering, and Optimization*, Ph.D. Dissertation, California Institute of Technology, Pasadena, California, 1991.

[86] K. Rose, E. Gurewitz, and G. C. Fox, "Constrained Clustering as an Optimization Method," Technical Report SCCS-21, Northeast Parallel Architectures Center, Syracuse University.

[87] R. Sedgewick, *Algorithms*, Second Edition, Addison-Wesley Publishing Co., Reading, Massachusetts, 1988.

[88] H. Simon. "Partitioning of Unstructured Mesh Problems for Parallel Processing," *Proc. Conf. Parallel Methods on Large Scale Structural Analysis and Physics Applications*, Pergamon Press, 1991.

[89] P. H. A. Sneath and R. R. Sokal, *Numerical Taxonomy*, W. H. Freeman, San Francisco, 1973.

[90] D. Stauffer, *Introduction to Percolation Theory*, Taylor and Francis, London, 1985.

[91] R. H. Swendsen and J. Wang, "Nonuniversal Critical Dynamics in Monte Carlo Simulations," *Physical Review Letters*, 58(2):86-88, 1987.

[92] H. N. V. Temperley, *Graph Theory and Applications*, Halsted Press (A division of John Wiley & Sons), New York, 1981.

[93] Thinking Machines Corporation, Cambridge, MA, *The Connection Machine CM-5 Technical Summary*, 1991.

[94] Thinking Machines Corporation, Cambridge, MA, *CM Fortran Reference Manual*, Version 1.0, 1991.

[95] Thinking Machines Corporation, Cambridge, MA, *Porting to CM-5 *Lisp*, Version 7.1, 1991.

[96] Thinking Machines Corporation, Cambridge, MA, *CMMD Reference Manual*, Version 3.0, 1993.

[97] Thinking Machines Corporation, Cambridge, MA, *C* Programming Guide*, 1993.

[98] J. C. Tilton, "Image segmentation by iterative parallel region growing with applications to data compression and image analysis," *Proc. 2nd Symposium on the Frontiers of Massively Parallel Computation*, 1988.

[99] I. Tomescu, *Problems in Combinatorics and Graph Theory*, Trans. by R. A. Melter, John Wiley & Sons, 1985.

[100] W. T. Tutte, *Connectivity in Graphs*, (Mathematical Expositions No. 15), University of Toronto Press, 1966.

[101] W. T. Tutte, *Graph Theory*, (Encyclopedia of Mathematics and its Applications, Vol. 21), Addison-Wesley, Menlo Park, California, 1984.

[102] R. Urquhart, "Graph Theoretical Clustering Based on Limited Neighborhood Sets," *Pattern Recognition*, Vo. 15, No. 3, pp. 173-187, 1982.

[103] J. Van Ryzin, Editor, *Classification and Clustering*, Proceedings of an advanced seminar conducted by the Mathematics Research Center, The Uni-

versity of Wisconsin at Madison, May 3-5, 1976, Academic Press, New York, 1977.

[104] J.-C. Wang, T.-H. Lin, S. Ranka, "NICE: Non-uniform Irregular Communication Exchange on Distributed Memory Systems," Technical Report SCCS-546, Northeast Parallel Architectures Center, Syracuse University, 1993. URL: http://www.npac.syr.edu/techreports/html/0500/abs-0546.html

[105] M. Willebeek-LeMair and A. Reeves, "Solving non-uniform problems on SIMD computers: Case study on region growing," *J. Parallel Distrib. Comput.*, Vol. 8, pp. 135-149, 1990.

[106] R. Williams, "Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations, *Concurrency: Practice and Experience*, Vol. 3, No. 5, pp. 457-481, 1991.

[107] R. J. Wilson, *Introduction to Graph Theory*, Second Edition, Academic Press, New York, 1979.

[108] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S-W Liao, C-W Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers," *ACM SIGPLAN Notices*, Vol. 29, No. 12, pp. 31-37, December 1994.

[109] Z. Wu and R. Leahy, "An Optimal Graph Theoretic Approach to Data Clustering: Theory and Its Application to Image Segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 15, No. 11, pp. 1101-1113, November 1993.

[110] T. Y. Young and K. Fu, Editors, *Handbook of Pattern Recognition and Image Processing*, Academic Press, Orlando, Florida, 1986.

[111] C. T. Zahn, "Graph-Theoretical Methods for Detecting and Describing Gestalt Clusters," *IEEE Transactions on Computers*, Vol. C-20, No. 1, January 1971.

[112] S. W. Zucker, "Region growing: Childhood and adolescence," *Computer Graphics and Image Processing*, Vol. 5, pp. 382-399, 1976.

Appendix A

Clustering Subroutines in HPF

A.1 Main Program

```
C  ---------------------------------------------------------------------
C  This subroutine clusters a two-dimensional array of pixel intensities
C  into homogeneous regions.
C
C  The input arguments are:
C
C       N              : Dimension of the array of pixel intensities
C       THRESHOLD      : The threshold value
C       PIXEL_VAL      : Two-dimensional array of pixel intensities
C
C  The output argument is:
C
C       CLUSTER_LABEL  : Two-dimensional array of cluster labels.
C                        The array has the same shape as PIXEL_VAL.
C  ---------------------------------------------------------------------

      SUBROUTINE CLUSTER (N, THRESHOLD, PIXEL_VAL, CLUSTER_LABEL)

      INTEGER N, M1, M2, THRESHOLD

      INTEGER, DIMENSION (N, N) :: PIXEL_VAL, CLUSTER_LABEL,
     &   CLUSTER_SIZE, CLUSTER_MIN, CLUSTER_MAX

      LOGICAL, DIMENSION (N, N) :: CLUSTER_REP
      INTEGER, ALLOCATABLE :: VERTEX_LABEL(:), VERTEX_MIN(:),
     &   VERTEX_MAX(:), VERTEX_MERGE(:), VERTEX_PARTNER(:),
     &   EDGE_VERTEX_1(:), EDGE_VERTEX_2(:), EDGE_MIN(:),
     &   EDGE_MAX(:), EDGE_ACTIVE(:)

!HPF$ DISTRIBUTE (BLOCK, BLOCK) :: PIXEL_VAL
!HPF$ ALIGN WITH PIXEL_VAL :: CLUSTER_LABEL, CLUSTER_REP,
     &   CLUSTER_SIZE, CLUSTER_MIN, CLUSTER_MAX

!HPF$ DISTRIBUTE (BLOCK) :: VERTEX_LABEL
!HPF$ ALIGN WITH VERTEX_LABEL :: VERTEX_MIN, VERTEX_MAX, VERTEX_MERGE,
     &   VERTEX_PARTNER

!HPF$ DISTRIBUTE (BLOCK) :: EDGE_ACTIVE
!HPF$ ALIGN WITH EDGE_ACTIVE :: EDGE_VERTEX_1, EDGE_VERTEX_2, EDGE_MIN,
     &   EDGE_MAX


C     Split stage.
      CALL SPLIT (N, THRESHOLD, PIXEL_VAL, CLUSTER_REP, CLUSTER_LABEL,
     &   CLUSTER_SIZE, CLUSTER_MIN, CLUSTER_MAX)


C     Allocate vertex and edge arrays.
      M1 = COUNT(CLUSTER_REP)
      M2 = M1 * 5

      ALLOCATE (VERTEX_LABEL(M1), VERTEX_MIN(M1), VERTEX_MAX(M1),
     &   VERTEX_MERGE(M1), VERTEX_PARTNER(M1))

      ALLOCATE (EDGE_VERTEX_1(M2), EDGE_VERTEX_2(M1), EDGE_MIN(M2),
     &   EDGE_MAX(M2), EDGE_ACTIVE(M2))


C     Form graph modeling problem.
      CALL INITIALIZE_VERTEX_ARRAYS (CLUSTER_LABEL, CLUSTER_MIN,
     &   CLUSTER_MAX, CLUSTER_REP, VERTEX_LABEL, VERTEX_MIN,
     &   VERTEX_MAX)

      CALL INITIALIZE_EDGE_ARRAYS (N, THRESHOLD,
     &   CLUSTER_LABEL, CLUSTER_SIZE, CLUSTER_REP,
     &   VERTEX_MIN, VERTEX_MAX, EDGE_VERTEX_1,
     &   EDGE_VERTEX_2, EDGE_MIN, EDGE_MAX, EDGE_ACTIVE)
```

```
C      Merge Stage.
       CALL MERGE_VERTICES (N, THRESHOLD, VERTEX_LABEL, VERTEX_MIN,
     &    VERTEX_MAX, EDGE_VERTEX_1, EDGE_VERTEX_2, EDGE_MIN,
     &    EDGE_MAX, EDGE_ACTIVE, VERTEX_MERGE, VERTEX_PARTNER)


C      Update two-dimensional array CLUSTER_LABEL.
       CALL UPDATE_LABELS (CLUSTER_REP, CLUSTER_LABEL, VERTEX_LABEL)


C      Deallocate vertex and edge arrays.
       DEALLOCATE (VERTEX_LABEL, VERTEX_MIN, VERTEX_MAX,
     &    VERTEX_MERGE, VERTEX_PARTNER)

       DEALLOCATE (EDGE_VERTEX_1, EDGE_VERTEX_2, EDGE_MIN,
     &    EDGE_MAX, EDGE_ACTIVE)


       RETURN
       END SUBROUTINE CLUSTER
```

A.2 Split

```
C --------------------------------------------------------------------
C This subroutine splits a two-dimensional array of pixel intensities
C into initial clusters (square regions) that satisfy the homogeneity
C requirement.
C
C The input arguments are:
C
C      N              : Dimension of the array of pixel intensities
C      THRESHOLD      : The threshold value
C      PIXEL_VAL      : Two-dimensional array of pixel intensities
C
C The output arguments are:
C
C      CLUSTER_REP    : Indicates whether a pixel is a square region
C                       representative
C      CLUSTER_LABEL  : Two-dimensional array of cluster labels
C                       assigned to square regions.  The array has
C                       the same shape as PIXEL_VAL
C      CLUSTER_SIZE   : Sizes of square regions (number of pixels per
C                       row or column of region)
C      CLUSTER_MIN    : Minimum pixel values in square regions
C      CLUSTER_MAX    : Maximum pixel values in square regions
C --------------------------------------------------------------------

      SUBROUTINE SPLIT (N, THRESHOLD, PIXEL_VAL, CLUSTER_REP,
     &   CLUSTER_LABEL, CLUSTER_SIZE, CLUSTER_MIN, CLUSTER_MAX)

      INTEGER N, THRESHOLD, I, J, K, K1, K2, MAX_SIZE
      INTEGER, DIMENSION (:, :) :: PIXEL_VAL, CLUSTER_LABEL,
     &   CLUSTER_SIZE, CLUSTER_MIN, CLUSTER_MAX
      LOGICAL, DIMENSION (:, :) :: CLUSTER_REP
      INTEGER, DIMENSION (N, N) :: NEW_MIN, NEW_MAX
      LOGICAL, DIMENSION (N, N) :: FLAG_1, FLAG_2

!HPF$ INHERIT :: PIXEL_VAL, CLUSTER_REP, CLUSTER_LABEL, CLUSTER_SIZE,
     &   CLUSTER_MIN, CLUSTER_MAX

!HPF$ ALIGN WITH PIXEL_VAL :: NEW_MIN, NEW_MAX, FLAG_1, FLAG_2


C     Initialize.
      CLUSTER_REP   = .TRUE.
```

```
       CLUSTER_LABEL = 1
       CLUSTER_SIZE  = 1
       CLUSTER_MIN   = PIXEL_VAL
       CLUSTER_MAX   = PIXEL_VAL
       K             = 1


C      Loop to perform splitting.
       DO WHILE ((K < N) .AND. (COUNT (FLAG_2) > 0))
          FLAG_1    = .FALSE.
          FLAG_2    = .FALSE.
          NEW_MIN   = 0
          NEW_MAX   = 0

          FORALL (I=1:N-K, J=1:N-K,
     &      (MOD (I, K * 2) == 1) .AND.
     &      (MOD (J, K * 2) == 1))
     &      FLAG_1(I, J) =
     &      CLUSTER_REP(I, J)              .AND.
     &      CLUSTER_REP(I, J+K)            .AND.
     &      CLUSTER_REP(I+K, J)            .AND.
     &      CLUSTER_REP(I+K, J+K)          .AND.
     &      (CLUSTER_SIZE(I, J)     == K)  .AND.
     &      (CLUSTER_SIZE(I, J+K)   == K)  .AND.
     &      (CLUSTER_SIZE(I+K, J)   == K)  .AND.
     &      (CLUSTER_SIZE(I+K, J+K) == K)

          FORALL (I=1:N-K, J=1:N-K, FLAG_1(I,J))
             NEW_MIN(I,J) = MIN (CLUSTER_MIN(I, J), CLUSTER_MIN(I, J+K),
     &          CLUSTER_MIN(I+K, J), CLUSTER_MIN(I+K, J+K))
             NEW_MAX(I,J) = MAX (CLUSTER_MAX(I, J), CLUSTER_MAX(I, J+K),
     &          CLUSTER_MAX(I+K, J), CLUSTER_MAX(I+K, J+K))
          END FORALL

          FORALL (I=1:N-K, J=1:N-K)
     &       FLAG_2(I, J) =
     &       FLAG_1(I, J) .AND.
     &       ((NEW_MAX(I, J) - NEW_MIN(I,J) <= THRESHOLD))

          FORALL (I=1:N-K, J=1:N-K, FLAG_2(I, J))
             CLUSTER_SIZE(I, J)    = K * 2
             CLUSTER_MIN(I, J)     = NEW_MIN(I, J)
             CLUSTER_MAX(I, J)     = NEW_MAX(I, J)
             CLUSTER_REP(I, J+K)   = .FALSE.
             CLUSTER_REP(I+K, J)   = .FALSE.
```

```
            CLUSTER_REP(I+K, J+K) = .FALSE.
         END FORALL

         K = K * 2
      END DO


C     Assign a unique label to each representative of a square region.
      CLUSTER_LABEL = COUNT_PREFIX (CLUSTER_REP)


C     Assign label of representative of a square region to all pixels
C     within that region.
      MAX_SIZE = MAXVAL (CLUSTER_SIZE) - 1
      DO K1 = 0, MAX_SIZE
         DO K2 = 0, MAX_SIZE
            FORALL (I=1:N-K1, J=1:N-K2,
     &         CLUSTER_REP(I,J) .AND.
     &         (K1 < CLUSTER_SIZE(I,J)) .AND.
     &         (K2 < CLUSTER_SIZE(I,J)))
     &         CLUSTER_LABEL(I + K1, J + K2) =  CLUSTER_LABEL(I,J)
         END DO
      END DO


      RETURN
      END SUBROUTINE SPLIT
```

A.3 Initialize Vertex Arrays

```
C  ---------------------------------------------------------------------
C  This subroutine initializes the one-dimensional arrays that describe
C  the vertices of the graph modeling the problem.
C
C  The input arguments are:
C
C       CLUSTER_LABEL  : Two-dimensional array of cluster labels
C                        assigned to square regions.
C       CLUSTER_MIN    : Minimum pixel values in square regions
C       CLUSTER_MAX    : Maximum pixel values in square regions
C       CLUSTER_REP    : Indicates whether a pixel is a square region
C                        representative
C
C  The ouput arguments are:
C
C       VERTEX_LABEL   : Labels assigned to vertices indicating to
C                        which clusters they belong
C       VERTEX_MIN     : Minimum pixel values in clusters represented
C                        by vertices.
C       VERTEX_MAX     : Maximum pixel values in clusters represented
C                        by vertices.
C  ---------------------------------------------------------------------

       SUBROUTINE INITIALIZE_VERTEX_ARRAYS (CLUSTER_LABEL,
     &    CLUSTER_MIN, CLUSTER_MAX, CLUSTER_REP, VERTEX_LABEL,
     &    VERTEX_MIN, VERTEX_MAX)

       INTEGER I
       INTEGER, DIMENSION (:, :) :: CLUSTER_LABEL, CLUSTER_MIN,
     &    CLUSTER_MAX
       LOGICAL, DIMENSION (:, :) :: CLUSTER_REP
       INTEGER, DIMENSION (:) :: VERTEX_LABEL, VERTEX_MIN, VERTEX_MAX
       INTEGER, DIMENSION (SIZE(VERTEX_LABEL)) :: ZERO

!HPF$  INHERIT :: CLUSTER_LABEL, CLUSTER_MIN, CLUSTER_MAX,
     &    CLUSTER_REP, VERTEX_LABEL, VERTEX_MIN, VERTEX_MAX

!HPF$  ALIGN ZERO WITH VERTEX_LABEL


C      Initialize.
       ZERO = 0
```

```
C       Assign values to vertex arrays.
        FORALL (I=1:SIZE(VERTEX_LABEL)) VERTEX_LABEL(I) = I

        VERTEX_MIN = COPY_SCATTER (CLUSTER_MIN, ZERO, CLUSTER_LABEL,
     &     MASK = CLUSTER_REP)

        VERTEX_MAX = COPY_SCATTER (CLUSTER_MAX, ZERO, CLUSTER_LABEL,
     &     MASK = CLUSTER_REP)


        RETURN
        END SUBROUTINE INITIALIZE_VERTEX_ARRAYS
```

A.4 Initialize Edge Arrays

```
C  ---------------------------------------------------------------------
C  This subroutine initializes the one-dimensional arrays that describe
C  the edges of the graph modeling the problem.
C
C  The input arguments are:
C
C       N             : Dimension of the array of pixel intensities
C       THRESHOLD     : The threshold value
C       CLUSTER_LABEL : Two-dimensional array of cluster labels
C                       assigned to square regions.
C       CLUSTER_SIZE  : Sizes of square regions (number of pixels per
C                       row or column of region)
C       CLUSTER_REP   : Indicates whether a pixel is a square region
C                       representative
C       VERTEX_MIN    : Minimum pixel values in clusters represented
C                       by vertices.
C       VERTEX_MAX    : Maximum pixel values in clusters represented
C                       by vertices.
C
C  The ouput arguments are:
C
C       EDGE_VERTEX_1 : Index of first vertex of an edge
C       EDGE_VERTEX_2 : Index of second vertex of an edge
C       EDGE_MIN      : Minimum pixel value in first and second
C                       vertices of edge combined
C       EDGE_MAX      : Maximum pixel value in first and second
C                       vertices of edge combined
C       EDGE_ACTIVE   : Indicates whether an edge is active
C  ---------------------------------------------------------------------

      SUBROUTINE INITIALIZE_EDGE_ARRAYS (N, THRESHOLD,
     &   CLUSTER_LABEL, CLUSTER_SIZE, CLUSTER_REP,
     &   VERTEX_MIN, VERTEX_MAX, EDGE_VERTEX_1,
     &   EDGE_VERTEX_2, EDGE_MIN, EDGE_MAX, EDGE_ACTIVE)

      INTEGER N, THRESHOLD, M, N_NUM, S_NUM, E_NUM, W_NUM,
     &   L_BOUND, U_BOUND, I, J

      INTEGER, DIMENSION (:, :) :: CLUSTER_LABEL, CLUSTER_SIZE

      LOGICAL, DIMENSION (:, :) :: CLUSTER_REP
```

```
       INTEGER, DIMENSION (:) :: VERTEX_MIN, VERTEX_MAX,
     &    EDGE_VERTEX_1, EDGE_VERTEX_2, EDGE_MIN, EDGE_MAX

       LOGICAL, DIMENSION (:) :: EDGE_ACTIVE

       INTEGER, DIMENSION (N, N) :: N_NEIGHBOR, S_NEIGHBOR, E_NEIGHBOR,
     &    W_NEIGHBOR, TEMP

       LOGICAL, DIMENSION (N, N) :: N_FLAG, S_FLAG, E_FLAG, W_FLAG

!HPF$  INHERIT :: CLUSTER_LABEL, CLUSTER_SIZE, CLUSTER_REP,
     &    VERTEX_MIN, VERTEX_MAX, EDGE_VERTEX_1, EDGE_VERTEX_2,
     &    EDGE_MIN, EDGE_MAX, EDGE_ACTIVE

!HPF$  ALIGN WITH CLUSTER_LABEL :: N_NEIGHBOR, S_NEIGHBOR,
     &    E_NEIGHBOR, W_NEIGHBOR, TEMP, N_FLAG, S_FLAG, E_FLAG,
     &    W_FLAG


C      Initialize.
       EDGE_VERTEX_1 = 0
       EDGE_VERTEX_2 = 0
       EDGE_MIN      = 0
       EDGE_MAX      = 0
       EDGE_ACTIVE   = .FALSE.
       N_NEIGHBOR    = 0
       S_NEIGHBOR    = 0
       E_NEIGHBOR    = 0
       W_NEIGHBOR    = 0
       FORALL (I=1:N, J=1:N)  N_FLAG(I,J) =
     &    CLUSTER_REP(I,J) .AND. (I .NE. 1)
       FORALL (I=1:N, J=1:N)  S_FLAG(I,J) =
     &    CLUSTER_REP(I,J) .AND. ((I + CLUSTER_SIZE(I,J)) .LE. N)
       FORALL (I=1:N, J=1:N)  E_FLAG(I,J) =
     &    CLUSTER_REP(I,J) .AND. ((J + CLUSTER_SIZE(I,J)) .LE. N)
       FORALL (I=1:N, J=1:N)  W_FLAG(I,J) =
     &    CLUSTER_REP(I,J) .AND. (J .NE. 1)
       N_NUM = COUNT (N_Flag)
       S_NUM = COUNT (S_Flag)
       E_NUM = COUNT (E_Flag)
       W_NUM = COUNT (W_Flag)


C      Find North, South, East, and West neighbors of every homogeneous
C      square region.
```

```
      forall (I=1:N, J=1:N, N_FLAG(I,J))
     &   N_NEIGHBOR(I,J) =  CLUSTER_LABEL(I-1, J)
      forall (I=1:N, J=1:N, S_FLAG(I,J))
     &   S_NEIGHBOR(I,J) =  CLUSTER_LABEL(I + CLUSTER_SIZE(I,J), J)
      forall (I=1:N, J=1:N, E_FLAG(I,J))
     &   E_NEIGHBOR(I,J) =  CLUSTER_LABEL(I, J + CLUSTER_SIZE(I,J))
      forall (I=1:N, J=1:N, W_FLAG(I,J))
     &   W_NEIGHBOR(I,J) =  CLUSTER_LABEL(I, J-1)


C     Store neighbor data in EDGE_VERTEX_1 and EDGE_VERTEX_2.
      L_BOUND = 1
      U_BOUND = N_NUM
      TEMP = MIN(CLUSTER_LABEL, N_NEIGHBOR)
      EDGE_VERTEX_1(L_BOUND:U_BOUND) = PACK (TEMP, N_FLAG)
      TEMP = MAX(CLUSTER_LABEL, N_NEIGHBOR)
      EDGE_VERTEX_2(L_BOUND:U_BOUND) = PACK (TEMP, N_FLAG)

      L_BOUND = U_BOUND + 1
      U_BOUND = L_BOUND + S_NUM - 1
      TEMP = MIN(CLUSTER_LABEL, S_NEIGHBOR)
      EDGE_VERTEX_1(L_BOUND:U_BOUND) = PACK (TEMP, S_FLAG)
      TEMP = MAX(CLUSTER_LABEL, S_NEIGHBOR)
      EDGE_VERTEX_2(L_BOUND:U_BOUND) = PACK (TEMP, S_FLAG)

      L_BOUND = U_BOUND + 1
      U_BOUND = L_BOUND + E_NUM - 1
      TEMP = MIN(CLUSTER_LABEL, E_NEIGHBOR)
      EDGE_VERTEX_1(L_BOUND:U_BOUND) = PACK (TEMP, E_FLAG)
      TEMP = MAX(CLUSTER_LABEL, E_NEIGHBOR)
      EDGE_VERTEX_2(L_BOUND:U_BOUND) = PACK (TEMP, E_FLAG)

      L_BOUND = U_BOUND + 1
      U_BOUND = L_BOUND + W_NUM - 1
      TEMP = MIN(CLUSTER_LABEL, W_NEIGHBOR)
      EDGE_VERTEX_1(L_BOUND:U_BOUND) = PACK (TEMP, W_FLAG)
      TEMP = MAX(CLUSTER_LABEL, W_NEIGHBOR)
      EDGE_VERTEX_2(L_BOUND:U_BOUND) = PACK (TEMP, W_FLAG)


C     Update EDGE_ACTIVE.
      FORALL (I=1:U_BOUND) EDGE_ACTIVE(I) = .TRUE.


C     Identify duplicate edges and set them inactive.
```

```
      CALL DEACTIVATE_DUPLICATE_EDGES (N, EDGE_VERTEX_1, EDGE_VERTEX_2,
     &   EDGE_ACTIVE)

C     Update EDGE_MIN, EDGE_MAX, and EDGE_ACTIVE.
      CALL UPDATE_EDGES_MIN_MAX (THRESHOLD, VERTEX_MIN, VERTEX_MAX,
     &   EDGE_VERTEX_1, EDGE_VERTEX_2, EDGE_MIN, EDGE_MAX, EDGE_ACTIVE)

      RETURN
      END SUBROUTINE INITIALIZE_EDGE_ARRAYS
```

A.5 Merge

```
C  ------------------------------------------------------------------
C  This subroutine implements the merge stage of hierarchical
C  clustering.
C
C  The input arguments are:
C
C       N               : Dimension of the array of pixel intensities
C       THRESHOLD       : The threshold value
C
C  The input/output arguments are:
C
C       VERTEX_LABEL    : Labels assigned to vertices indicating to
C                         which clusters they belong
C       VERTEX_MIN      : Minimum pixel values in clusters represented
C                         by vertices.
C       VERTEX_MAX      : Maximum pixel values in clusters represented
C                         by vertices.
C       EDGE_VERTEX_1   : Index of first vertex of an edge
C       EDGE_VERTEX_2   : Index of second vertex of an edge
C       EDGE_MIN        : Minimum pixel value in first and second
C                         vertices of edge combined
C       EDGE_MAX        : Maximum pixel value in first and second
C                         vertices of edge combined
C       EDGE_ACTIVE     : Indicates whether an edge is active
C  ------------------------------------------------------------------

       SUBROUTINE MERGE_VERTICES (N, THRESHOLD, VERTEX_LABEL,
     &    VERTEX_MIN, VERTEX_MAX, EDGE_VERTEX_1, EDGE_VERTEX_2,
     &    EDGE_MIN, EDGE_MAX, EDGE_ACTIVE)

       INTEGER N, THRESHOLD, M1, M2, NUM_CLUSTERS, NEW_NUM_CLUSTERS,
     &    TRIAL
       INTEGER, DIMENSION (:) :: VERTEX_LABEL, VERTEX_MIN, VERTEX_MAX,
     &    EDGE_VERTEX_1, EDGE_VERTEX_2, EDGE_MIN, EDGE_MAX
       LOGICAL, DIMENSION (:) :: EDGE_ACTIVE
       LOGICAL, DIMENSION (SIZE (VERTEX_LABEL)) :: VERTEX_MERGE
       INTEGER, DIMENSION (SIZE (VERTEX_LABEL)) :: VERTEX_PARTNER

!HPF$  INHERIT :: VERTEX_LABEL, VERTEX_MIN, VERTEX_MAX,
     &    EDGE_VERTEX_1, EDGE_VERTEX_2, EDGE_MIN, EDGE_MAX, EDGE_ACTIVE

!HPF$  ALIGN WITH VERTEX_LABEL :: VERTEX_MERGE, VERTEX_PARTNER
```

```
C       Initialize.
        M1    = SIZE (VERTEX_LABEL)
        M2    = SIZE (EDGE_ACTIVE)
        TRIAL = 0

C       Loop to perform merge stage.
        DO WHILE (COUNT (EDGE_ACTIVE) .NE. 0)

C          Match vertices.
           IF (TRIAL == 5) THEN
              CALL MATCH_VERTICES (.FALSE., N, EDGE_VERTEX_1,
     &           EDGE_VERTEX_2, EDGE_MIN, EDGE_MAX, EDGE_ACTIVE,
     &           VERTEX_MERGE, VERTEX_PARTNER)
              TRIAL = 0
           ELSE
              CALL MATCH_VERTICES (.TRUE., N, EDGE_VERTEX_1,
     &           EDGE_VERTEX_2, EDGE_MIN, EDGE_MAX, EDGE_ACTIVE,
     &           VERTEX_MERGE, VERTEX_PARTNER)
           END IF

C          Update vertices.
           CALL UPDATE_VERTICES (VERTEX_MIN, VERTEX_MAX, VERTEX_LABEL,
     &        VERTEX_MERGE, VERTEX_PARTNER)

C          Update edges.
           CALL UPDATE_EDGES (N, THRESHOLD, VERTEX_MIN, VERTEX_MAX,
     &        VERTEX_MERGE, VERTEX_PARTNER, EDGE_VERTEX_1,
     &        EDGE_VERTEX_2, EDGE_MIN, EDGE_MAX, EDGE_ACTIVE)

C          Find number of clusters determined so far and compare to
C          previous number of clusters.
           NEW_NUM_CLUSTERS = NUMBER_OF_CLUSTERS (VERTEX_CLUSTER)
           IF (NEW_NUM_CLUSTERS == NUM_CLUSTERS) THEN
              TRIAL = TRIAL + 1
           ELSE
              TRIAL = 0
           END IF
           NUM_CLUSTERS = NEW_NUM_CLUSTERS
        END DO

        RETURN
        END SUBROUTINE MERGE_VERTICES
```

A.6 Match

```
C  -----------------------------------------------------------------
C  This subroutine determines which vertices of the graph will
C  actually merge.
C
C  The input arguments are:
C
C       RANDOM         : Indicates how ties are to be resolved:
C                        RANDOM = .TRUE. indicates that ties are to be
C                        resolved by selecting a neighbor at random;
C                        RANDOM = .FALSE. indicates that ties are to be
C                        resolved by selecting the neighbor with the
C                        smallest label
C       N              : Dimension of the array of pixel intensities
C       EDGE_VERTEX_1  : Index of first vertex of an edge
C       EDGE_VERTEX_2  : Index of second vertex of an edge
C       EDGE_MIN       : Minimum pixel value in first and second
C                        vertices of edge combined
C       EDGE_MAX       : Maximum pixel value in first and second
C                        vertices of edge combined
C       EDGE_ACTIVE    : Indicates whether an edge is active
C
C  The output arguments are:
C
C       VERTEX_MERGE   : Indicates whether a vertex will actually
C                        merge with another
C       VERTEX_PARTNER : Merge partner of a vertex
C  -----------------------------------------------------------------

      SUBROUTINE MATCH_VERTICES (RANDOM, N, EDGE_VERTEX_1,
    &    EDGE_VERTEX_2, EDGE_MIN, EDGE_MAX, EDGE_ACTIVE, VERTEX_MERGE,
    &    VERTEX_PARTNER)

      INTEGER N, M1, M2, SHIFT_1, SHIFT_2, I
      LOGICAL RANDOM

      INTEGER, DIMENSION (:) :: EDGE_VERTEX_1, EDGE_VERTEX_2, EDGE_MIN,
    &    EDGE_MAX, VERTEX_PARTNER
      LOGICAL, DIMENSION (:) :: EDGE_ACTIVE, VERTEX_MERGE
      INTEGER, DIMENSION (SIZE(EDGE_ACTIVE)):: KEY_1, KEY_2, RANGE,
    &    RANDOM_NUM
      REAL, DIMENSION (SIZE(EDGE_ACTIVE)) :: REAL_RANDOM_NUM
      INTEGER, DIMENSION (SIZE (VERTEX_MERGE)) :: VERTEX_KEY
      LOGICAL, DIMENSION (SIZE (VERTEX_MERGE)) :: VERTEX_CAND

!HPF$ INHERIT :: EDGE_VERTEX_1, EDGE_VERTEX_2, EDGE_MIN, EDGE_MAX,
     &   EDGE_ACTIVE, VERTEX_MERGE, VERTEX_PARTNER

!HPF$ ALIGN WITH VERTEX_MERGE :: VERTEX_CAND, VERTEX_KEY

!HPF$ ALIGN WITH EDGE_ACTIVE :: KEY_1, KEY_2, RANGE, REAL_RANDOM_NUM,
     &   RANDOM_NUM


C     Initialize.
      M1             = SIZE (VERTEX_MERGE)
      M2             = SIZE (EDGE_ACTIVE)
      VERTEX_MERGE   = .FALSE.
      VERTEX_CAND    = .FALSE.
      VERTEX_PARTNER = 0
      VERTEX_KEY     = 0
      RANGE          = 0
      SHIFT_1        = 10 ** (NINT (LOG10 (REAL (N * N))) + 1)
      SHIFT_2        = SHIFT_1 * 100
      FORALL (I=1:M2, EDGE_ACTIVE(I))
     &   RANGE(I) = EDGE_MAX(I) - EDGE_MIN(I)
      CALL RANDOM_NUMBER (REAL_RANDOM_NUM)
      RANDOM_NUM = INT (REAL_RANDOM_NUM * 100)

C     Loop over active edges and pack information about vertices
C     into KEY_1 and KEY_2.
      IF (RANDOM) THEN
         FORALL (I=1:M2, EDGE_ACTIVE(I))
     &      KEY_1(I) = (RANGE(I) * SHIFT_2) +
     &                 (RANDOM_NUM(I) * SHIFT_1) +
     &                 EDGE_VERTEX_1(I)
         FORALL (I=1:M2, EDGE_ACTIVE(I))
     &      KEY_2(I) = (RANGE(I) * SHIFT_2) +
     &                 (RANDOM_NUM(I) * SHIFT_1) +
     &                 EDGE_VERTEX_2(I)
      ELSE
         FORALL (I=1:M2, EDGE_ACTIVE(I))
     &      KEY_1(I) = (RANGE(I) * SHIFT_1) + EDGE_VERTEX_1(I)
         FORALL (I=1:M2, EDGE_ACTIVE(I))
     &      KEY_2(I) = (RANGE(I) * SHIFT_1) +  EDGE_VERTEX_2(I)
      END IF

C     Initialize VERTEX_KEY to some big number that is larger than any
```

```
C      element of KEY_1 or KEY_2.
       VERTEX_KEY = 1 ! Some big number

C      Scatter elements of KEY_2 selected by EDGE_ACTIVE to positions of
C      VERTEX_KEY indicated by EDGE_VERTEX_1.  Each element of the result
C      will be assigned the minimum value of the existing element
C      of VERTEX_KEY and elements of KEY_2 scattered to that position.
       VERTEX_KEY = MINVAL_SCATTER (KEY_2, VERTEX_KEY, EDGE_VERTEX_1,
     &    MASK = EDGE_ACTIVE)

C      Scatter elements of KEY_1 selected by edge_active to positions of
C      VERTEX_key indicated by edge_VERTEX_2.  Each element of the result
C      will be assigned the minimum value of the existing element
C      of VERTEX_KEY and elements of KEY_1 scattered to that position.
       VERTEX_KEY = MINVAL_SCATTER (KEY_1, VERTEX_KEY, EDGE_VERTEX_2,
     &    MASK = EDGE_ACTIVE)

C      Update VERTEX_CAND.
       FORALL (I=1:M2, EDGE_ACTIVE(I))
     &    VERTEX_CAND(EDGE_VERTEX_1(I)) = .TRUE.
       FORALL (I=1:M2, EDGE_ACTIVE(I))
     &    VERTEX_CAND(EDGE_VERTEX_2(I)) = .TRUE.

C      Extract from VERTEX_KEY labels of vertices that are selected for
C      merging.  Store labels in VERTEX_PARTNER.
       FORALL (I=1:M1, VERTEX_CAND(I))
     &    VERTEX_PARTNER(I) = MOD (VERTEX_KEY(I), SHIFT_1)

C      Update VERTEX_MERGE to indicate which vertices will actually merge.
C      These are pairs of vertices that select each other for merging.
       FORALL (I=1:M1, VERTEX_CAND(I))
     &    VERTEX_MERGE(I) = (VERTEX_PARTNER(VERTEX_PARTNER(I)) == I)

       RETURN
       END SUBROUTINE MATCH_VERTICES
```

A.7 Update Vertices

```
C  ----------------------------------------------------------
C  This subroutine updates the vertices of the graph after a
C  merge iteration.
C
C  The input arguments are:
C
C       N              : Dimension of the array of pixel intensities
C       VERTEX_LABEL   : Labels assigned to vertices indicating to
C                        which clusters they belong
C       VERTEX_MERGE   : Indicates whether a vertex will actually
C                        merge with another
C                        by vertices
C       VERTEX_PARTNER : Merge partner of a vertex
C
C  The input/output arguments are:
C
C       VERTEX_MIN     : Minimum pixel values in clusters represented
C                        by vertices
C       VERTEX_MAX     : Maximum pixel values in clusters represented
C                        by vertices
C  ----------------------------------------------------------

      SUBROUTINE UPDATE_VERTICES (VERTEX_MIN, VERTEX_MAX,
     &   VERTEX_LABEL, VERTEX_MERGE, VERTEX_PARTNER)

      INTEGER M, I
      INTEGER, DIMENSION (:) :: VERTEX_MIN, VERTEX_MAX, VERTEX_LABEL,
     &   VERTEX_PARTNER
      LOGICAL, DIMENSION (:) :: VERTEX_MERGE
      LOGICAL, DIMENSION (SIZE (VERTEX_MERGE)) ::SMALLER

!HPF$  INHERIT :: VERTEX_MIN, VERTEX_MAX, VERTEX_LABEL, VERTEX_MERGE,
     &   VERTEX_PARTNER

!HPF$ ALIGN WITH VERTEX_LABEL :: SMALLER


C     Initialize.
      M = SIZE (VERTEX_MERGE)
      FORALL (I=1:M)
     &   SMALLER(I) = VERTEX_MERGE(I) .AND. (I < VERTEX_PARTNER(I))
```

```
C      Update cluster label of vertex with larger index so that
C      it belongs to vertex with smaller index.
       FORALL (I=1:M, SMALLER(I))
     &    VERTEX_LABEL(VERTEX_PARTNER(I)) = I

C      Update VERTEX_MIN and VERTEX_MAX for vertices with smaller index.
       FORALL (I=1:M, SMALLER(I)) VERTEX_MIN(I) =
     &    MIN (VERTEX_MIN(I), VERTEX_MIN(VERTEX_PARTNER(I)))
       FORALL (I=1:M, SMALLER(I)) VERTEX_MAX(I) =
     &    MAX (VERTEX_MAX(I), VERTEX_MAX(VERTEX_PARTNER(I)))

       RETURN
       END SUBROUTINE UPDATE_VERTICES
```

A.8 Update Edges

```
C  ----------------------------------------------------------
C  This subroutine updates edges of the graph after a merge
C  iteration.
C
C  The input arguments are:
C
C       N               : Dimension of the array of pixel intensities
C       THRESHOLD       : The threshold value
C       VERTEX_MIN      : Minimum pixel values in clusters represented

C       VERTEX_MAX      : Maximum pixel values in clusters represented
C                         by vertices
C       VERTEX_MERGE    : Indicates whether a vertex will actually
C                         merge with another
C       VERTEX_PARTNER  : Merge partner of a vertex
C
C  The input/output arguments are:
C
C       EDGE_VERTEX_1   : Index of first vertex of an edge
C       EDGE_VERTEX_2   : Index of second vertex of an edge
C       EDGE_MIN        : Minimum pixel value in first and second
C                         vertices of edge combined
C       EDGE_MAX        : Maximum pixel value in first and second
C                         vertices of edge combined
C       EDGE_ACTIVE     : Indicates whether an edge is active
C  ----------------------------------------------------------

       SUBROUTINE UPDATE_EDGES (N, THRESHOLD, VERTEX_MIN, VERTEX_MAX,
     &    VERTEX_MERGE, VERTEX_PARTNER, EDGE_VERTEX_1, EDGE_VERTEX_2,
     &    EDGE_MIN, EDGE_MAX, EDGE_ACTIVE)

       INTEGER N, THRESHOLD, M, I
       INTEGER, DIMENSION (:) :: VERTEX_MIN, VERTEX_MAX, VERTEX_PARTNER,
     &    EDGE_VERTEX_1, EDGE_VERTEX_2, EDGE_MIN, EDGE_MAX
       LOGICAL, DIMENSION (:) :: VERTEX_MERGE, EDGE_ACTIVE
       INTEGER, DIMENSION (SIZE(EDGE_ACTIVE))  ::  PARTNER_VERTEX_1,
     &    PARTNER_VERTEX_2, TEMP
       LOGICAL, DIMENSION (SIZE(EDGE_ACTIVE)) :: MERGE_1, MERGE_2

!HPF$  INHERIT VERTEX_MIN, VERTEX_MAX, VERTEX_MERGE, VERTEX_PARTNER,
     &    EDGE_VERTEX_1, EDGE_VERTEX_2, EDGE_MIN, EDGE_MAX, EDGE_ACTIVE
```

```
!HPF$  ALIGN WITH EDGE_ACTIVE :: MERGE_1, MERGE_2, PARTNER_VERTEX_1,
     &    PARTNER_VERTEX_2, TEMP


C      Initialize.
       M                 = SIZE (EDGE_ACTIVE)
       MERGE_1           = .FALSE.
       MERGE_2           = .FALSE.
       PARTNER_VERTEX_1  = 0
       PARTNER_VERTEX_2  = 0
       TEMP              = 0
       FORALL(I=1:M, EDGE_ACTIVE(I))
     &    MERGE_1(I) = VERTEX_MERGE(EDGE_VERTEX_1(I))
       FORALL(I=1:M, EDGE_ACTIVE(I))
     &    MERGE_2(I) = VERTEX_MERGE(EDGE_VERTEX_2(I))
       FORALL(I=1:M, MERGE_1(I))
     &    PARTNER_VERTEX_1(I) = VERTEX_PARTNER(EDGE_VERTEX_1(I))
       FORALL(I=1:M, MERGE_2(I))
     &    PARTNER_VERTEX_2(I) = VERTEX_PARTNER(EDGE_VERTEX_2(I))


C      Update vertices of edges after merge.
C      Case 1:  Two vertices joined by an edge are merged together.
       WHERE (MERGE_1 .AND. MERGE_2 .AND.
     &    (PARTNER_VERTEX_1 == EDGE_VERTEX_2))
     &    EDGE_ACTIVE = .FALSE.

C      Case 2:  First vertex of edge is merged with some vertex
C      other than second vertex of that edge.
       WHERE (MERGE_1 .AND.
     &    (.NOT. (MERGE_2 .AND. (PARTNER_VERTEX_1 == EDGE_VERTEX_2))))
     &    EDGE_VERTEX_1 = MIN (EDGE_VERTEX_1, PARTNER_VERTEX_1)

C      Case 3:  Second vertex of edge is merged with some vertex
C      other than first vertex of that edge.
       WHERE (MERGE_2 .AND.
     &    (.NOT. (MERGE_1 .AND. (PARTNER_VERTEX_1 == EDGE_VERTEX_2))))
     &    EDGE_VERTEX_2 = MIN (EDGE_VERTEX_2, PARTNER_VERTEX_2)


C      Arrange data in EDGE_VERTEX_1 and EDGE_VERTEX_2 such that
C      EDGE_VERTEX_1(I) <= EDGE_VERTEX_2(I).
       FORALL(I=1:M, EDGE_ACTIVE(I))
     &    TEMP(I) = MIN (EDGE_VERTEX_1(I), EDGE_VERTEX_2(I))
       FORALL(I=1:M, EDGE_ACTIVE(I))
     &    EDGE_VERTEX_2(I) = MAX (EDGE_VERTEX_1(I), EDGE_VERTEX_2(I))
      FORALL(I=1:M, EDGE_ACTIVE(I))
     &    EDGE_VERTEX_1(I) = TEMP(I)

C       Identify duplicate edges and set them inactive.
      CALL DEACTIVATE_DUPLICATE_EDGES (N, EDGE_VERTEX_1, EDGE_VERTEX_2,
     &    EDGE_ACTIVE)


C       Update EDGE_MIN and EDGE_MAX.
      CALL UPDATE_EDGES_MIN_MAX (THRESHOLD, VERTEX_MIN, VERTEX_MAX,
     &    EDGE_VERTEX_1, EDGE_VERTEX_2, EDGE_MIN, EDGE_MAX, EDGE_ACTIVE)

      RETURN
      END SUBROUTINE UPDATE_EDGES
```

A.9 Update Minimum and Maximum Values Associated with Edges

```
C  -------------------------------------------------------------------
C  This subroutine updates the minimum and maximum pixel values for
C  the edges.  Edges between regions that exceed threshold are set
C  inactive.
C
C  The input arguments are:
C
C       THRESHOLD      : The threshold value
C       VERTEX_MIN     : Minimum pixel values in clusters represented
C                        by vertices.
C       VERTEX_MAX     : Maximum pixel values in clusters represented
C                        by vertices.
C
C       The input/output arguments are:
C
C       EDGE_VERTEX_1  : Index of first vertex of an edge
C       EDGE_VERTEX_2  : Index of second vertex of an edge
C       EDGE_MIN       : Minimum pixel value in first and second
C                        vertices of edge combined
C       EDGE_MAX       : Maximum pixel value in first and second
C                        vertices of edge combined
C       EDGE_ACTIVE    : Indicates whether an edge is active
C  -------------------------------------------------------------------

      SUBROUTINE UPDATE_EDGES_MIN_MAX (THRESHOLD, VERTEX_MIN,
     &   VERTEX_MAX, EDGE_VERTEX_1, EDGE_VERTEX_2, EDGE_MIN,
     &   EDGE_MAX, EDGE_ACTIVE)

      INTEGER THRESHOLD, M, I
      INTEGER, DIMENSION (:) :: VERTEX_MIN, VERTEX_MAX,
     &   EDGE_VERTEX_1, EDGE_VERTEX_2, EDGE_MIN, EDGE_MAX
      LOGICAL, DIMENSION (:) :: EDGE_ACTIVE

!HPF$ INHERIT :: VERTEX_MIN, VERTEX_MAX, EDGE_VERTEX_1,
     &   EDGE_VERTEX_2, EDGE_MIN, EDGE_MAX, EDGE_ACTIVE


C     Initialize:
      M = SIZE (EDGE_ACTIVE)
```

```
C      Update EDGE_MIN for each edge to be the minimum pixel value
C      of first and second vertices of edge combined.
       FORALL (I=1:M, EDGE_ACTIVE(I))
     &    EDGE_MIN(I) = MIN (VERTEX_MIN(EDGE_VERTEX_1(I)),
     &                       VERTEX_MIN(EDGE_VERTEX_2(I)))

C      Update EDGE_MAX for each edge to be the maximum pixel value
C      of first and second vertices of edge combined.
       FORALL (I=1:M, EDGE_ACTIVE(I))
     &    EDGE_MAX(I) = MAX (VERTEX_MAX(EDGE_VERTEX_1(I)),
     &                       VERTEX_MAX(EDGE_VERTEX_2(I)))

C      Set edges that exceed threshold inactive.
       FORALL (I=1:M,
     &    EDGE_ACTIVE(I) .AND.
     &      (EDGE_MAX(I) - EDGE_MIN(I)) .GT. THRESHOLD)
     &    EDGE_ACTIVE(I) = .FALSE.

       RETURN
       END SUBROUTINE UPDATE_EDGES_MIN_MAX
```

A.10 Deactivate Duplicate Edges

```
C  --------------------------------------------------------------------
C  This subroutine identifies duplicate edges and sets them inactive.
C
C  The input arguments are:
C
C       N              : Dimension of the array of pixel intensities
C       EDGE_VERTEX_1  : Index of first vertex of an edge
C       EDGE_VERTEX_2  : Index of second vertex of an edge
C
C  The input/output argument is:
C
C       EDGE_ACTIVE    : Indicates whether an edge is active
C  --------------------------------------------------------------------

       SUBROUTINE DEACTIVATE_DUPLICATE_EDGES (N, EDGE_VERTEX_1,
     &    EDGE_VERTEX_2, EDGE_ACTIVE)

       INTEGER N, M, NUM, SHIFT, I

       INTEGER, DIMENSION (:) :: EDGE_VERTEX_1, EDGE_VERTEX_2

       LOGICAL, DIMENSION (:) :: EDGE_ACTIVE

       INTEGER, DIMENSION (SIZE (EDGE_ACTIVE)) :: KEY, PACKED_KEY,
     &    SORTED_KEY, INDEX, PACKED_INDEX, SORTED_INDEX,
     &    BIG_NUM, PERMUTATION

       LOGICAL, DIMENSION (SIZE (EDGE_ACTIVE)) :: DUPLICATE

!HPF$  INHERIT :: EDGE_VERTEX_1, EDGE_VERTEX_2, EDGE_ACTIVE

!HPF$  ALIGN WITH EDGE_ACTIVE :: KEY, PACKED_KEY, SORTED_KEY,
     &    INDEX, PACKED_INDEX, SORTED_INDEX, BIG_NUM, PERMUTATION


C      Initialize.
       M            = SIZE (EDGE_ACTIVE)
       NUM          = COUNT (EDGE_ACTIVE)
       KEY          = 0
       PACKED_KEY   = 0
       SORTED_KEY   = 0
       FORALL (I=1:M) INDEX(I) = I
```

```
        PACKED_INDEX = 0
        SORTED_INDEX = 0
        PERMUTATION  = 0
        DUPLICATE    = .FALSE.
        SHIFT        = 10 ** (NINT (LOG10 (REAL (N * N))) + 1)

C       BIG_NUM is any big number larger than any value assigned to KEY
        BIG_NUM      = (N * SHIFT) + N + 1


C       Pack EDGE_VERTEX_1(I) and EDGE_VERTEX_2(I) into one number (KEY)
C       to be used as key in sorting.
        WHERE (EDGE_ACTIVE)
     &     KEY = (EDGE_VERTEX_1 * SHIFT) +  EDGE_VERTEX_2

C       Pack elements to be sorted and their indices.
        PACKED_KEY   = PACK (KEY, EDGE_ACTIVE, BIG_NUM)
        PACKED_INDEX = PACK (INDEX, EDGE_ACTIVE, BIG_NUM)
        PERMUTATION  = GRADE_UP (PACKED_KEY, DIM=1)
        FORALL (I=1:NUM) SORTED_KEY(PERMUTATION(I))   = PACKED_KEY(I)
        FORALL (I=1:NUM) SORTED_INDEX(PERMUTATION(I)) = PACKED_INDEX(I)

C       Identify duplicate edges and set them inactive.
        FORALL (I=1:NUM-1, (SORTED_KEY(I) == SORTED_KEY(I+1)))
     &     DUPLICATE(I) = .TRUE.
        FORALL (I=1:NUM, DUPLICATE(I))
     &     EDGE_ACTIVE(SORTED_INDEX(I)) = .FALSE.

        RETURN
        END SUBROUTINE DEACTIVATE_DUPLICATE_EDGES
```

A.11 Assign Cluster Labels

```
C      ---------------------------------------------------------------
C      This subroutine updates the cluster labels in the two-dimensional
C      image by the labels assigned to the vertices of the graph.
C
C      The input arguments are:
C
C       CLUSTER_REP    : Indicates whether a pixel is a square region
C                        representative
C       VERTEX_LABEL   : Labels assigned to vertices indicating to
C                        which clusters they belong
C
C      The input/output argument is:
C
C       CLUSTER_LABEL  : Two-dimensional array of cluster labels
C                        assigned to square regions.
C      ---------------------------------------------------------------

       Subroutine UPDATE_LABELS (CLUSTER_REP, CLUSTER_LABEL,
     &    VERTEX_LABEL)

       INTEGER I, M
       INTEGER, DIMENSION (:, :) :: CLUSTER_LABEL
       LOGICAL, DIMENSION (:, :) :: CLUSTER_REP
       INTEGER, DIMENSION (:) :: VERTEX_LABEL
       INTEGER, DIMENSION (SIZE(VERTEX_LABEL)) :: OLD_VERTEX_LABEL,
     &    LABEL
       LOGICAL, DIMENSION (SIZE(VERTEX_LABEL)) :: FLAG

!HPF$  INHERIT :: CLUSTER_LABEL, CLUSTER_REP, VERTEX_LABEL

!HPF$  ALIGN WITH CLUSTER_REP  :: FLAG
!HPF$  ALIGN WITH VERTEX_LABEL :: OLD_VERTEX_LABEL, LABEL


C      Initialize.
       M = SIZE (VERTEX_LABEL)
       OLD_VERTEX_LABEL = 0
       FLAG = .TRUE.


C      Determine cluster label to which each graph vertex (square region)
C      belongs.
```

```
      DO WHILE (COUNT(FLAG) .GT. 0)
         FORALL (I=1: M)
     &        LABEL(I) = VERTEX_LABEL(VERTEX_LABEL(I))
         OLD_VERTEX_LABEL = VERTEX_LABEL
         VERTEX_LABEL = LABEL
         FORALL (I=1:M)
     &        FLAG(I) = OLD_VERTEX_LABEL(I) == VERTEX_LABEL(I)
      END DO


C     Transfer cluster labels to two-dimensional image.
      CLUSTER_LABEL = UNPACK (VERTEX_LABEL, CLUSTER_REP, -1)

      RETURN
      END ! UPDATE_LABELS
```

VITA

NAME OF AUTHOR: Nawal Copty

PLACE OF BIRTH: Baghdad, Iraq

DATE OF BIRTH: February 24, 1958

UNDERGRADUATE AND GRADUATE SCHOOLS ATTENDED:

American University of Beirut, Beirut, Lebanon

University of Aston in Birmingham, Birmingham, U.K.

Syracuse University, Syracuse, New York, U.S.A.

DEGREES AWARDED:

Bachelor of Science in Mathematics, 1980, American University of Beirut

Master of Science in Computer Science with Applications, 1982, University of Aston in Birmingham

Master of Science in Computer Science, 1990, Syracuse University