Parallelization of the Hoshen-Kopelman Algorithm Using a Finite State Machine

M.W. Berry, J.M. Constantin & B.T. Vander Zanden

Computer Science Department

CS-95-300 August 1995

Submitted to Int. J. of Supercomputer Applications and High Performance Computing, (1995)

Parallelization of the Hoshen-Kopelman Algorithm Using a Finite State Machine

MICHAEL W. BERRY*
Department of Computer Science, University of Tennessee, 107 Ayres Hall, Knoxville TN 37996-1301, `berry@cs.utk.edu`

JEFFREY M. CONSTANTIN
Department of Computer Science, University of Tennessee, 107 Ayres Hall, Knoxville TN 37996-1301, `constant@cs.utk.edu`

BRADLEY T. VANDER ZANDEN**
Department of Computer Science, University of Tennessee, 107 Ayres Hall, Knoxville TN 37996-1301, `bvz@cs.utk.edu`

Abstract. In applications such as landscape ecology, computer modeling is used to assess habitat fragmentation and its ecological implications. Maps (2-D grids) of habitat clusters or patches are analyzed to determine the number, location, and sizes of clusters. Recently, improved sequential and parallel implementations of the Hoshen-Kopelman cluster identification algorithm have been designed. These implementations use a finite state machine to reduce redundant integer comparisons during the cluster identification process. The sequential implementation for *large-scale* maps performs cluster identification by partitioning the map along row boundaries and merging the results of the partitions. The parallel implementation on a 32-processor Thinking Machines CM–5 provides an efficient mechanism for performing cluster identification in parallel. While the sequential implementation achieved promising speed improvements ranging from 1.39 to 2.00 over an existing Hoshen-Kopelman implementation, the parallel implementation achieved a minimum *speedup* of 5.41 over the improved sequential implementation executing on a Sun SPARCstation 10.

Keywords: cluster identification, finite state machine, Hoshen-Kopelman algorithm, landscape ecology, parallel processing

1. Introduction

In landscape ecology, computer modeling can be used to assess habitat fragmentation, the migration patterns of individuals or groups of competing animal species, and the impact of human activities on ecosystems. In such applications, the primary data structure used within spatially explicit landscape models are maps (2-dimensional grids). These maps typically reflect habitat clusters or patches that can be analyzed to determine their numbers, sizes, and geometries. A core function of map analysis is cluster identification—a process by which individual pixels in a map that have been grouped in the same map class are given the same cluster label.

Cluster identification algorithms can be classified by their approach to pixel labeling as holistic or aggregate. Holistic-type algorithms find every pixel belonging to a particular cluster before analyzing the next cluster. These algorithms can be implemented either recursively or by using a working set to maintain a list of candidate pixels. In a working set implementation, each candidate pixel on the list is labeled and in turn, each of the pixel's neighbors is analyzed. If a neighboring pixel is in the same cluster and is currently unlabeled, the pixel is added to the list. This process continues until the list is exhausted signaling all pixels belonging to this cluster have been properly labeled.

In a recursive implementation, the original call to the cluster identification function (CIF) will accept a pointer to the first candidate pixel and will correctly label the pixel and its North-East-West-South (NEWS) neighbors. The CIF function labels the NEWS neighbors by issuing recursive calls to itself, one for each of the NEWS neighbors that are in the current map class and have not

* This author's research was supported by the National Science Foundation under grant numbers NSF-ASC-92-03004 and NSF-CDA-91-15428.
** This author's research was supported by the National Science Foundation under grant number NSF-IRI-9111121.

already been labeled. When the original call to the CIF returns, the entire cluster will have been correctly labeled. The process is repeated for each unique cluster found in the input map.

A recursive implementation requires the entire cluster to be in memory and large amounts of memory to store the execution stack during the recursion which makes this approach impractical for large dense maps. Aggregate-type algorithms traverse a map assigning pixels to temporary clusters and apply merge rules when two temporary clusters overlap. Aggregate-type algorithms are not subject to the high memory requirements of the recursive algorithms and are better suited for cluster identification on large-scale, dense maps. The Hoshen-Kopelman [6] cluster identification algorithm is an aggregate-type algorithm.

1.1. Objectives/Overview

The objectives of this study are to supply a more efficient sequential implementation of the Hoshen-Kopelman [6] (HK) cluster identification algorithm for 2-dimensional binary maps and to provide an efficient parallel implementation for cluster identification on the Thinking Machines CM–5 [7]. Optimizing the Hoshen-Kopelman sequential implementation involves the efficient use of pointers and local variables which can significantly reduce the time spent in *base-plus-offset* array indexing and redundant integer comparisons.

Before Section 2, which discusses the Hoshen-Kopelman one-pass cluster identification algorithm, the input data set, the programming data structures, the cluster *neighborhood rule* and the finite state machine used, the particular file format (ERDAS/Lan) used for test maps is briefly discussed below. Following a discussion of the major components of the finite state machine (FSM) implementation in Section 2, the performance methodology used to evaluate the sequential and parallel implementations and the computing environment are described in Section 3. Section 4 demonstrates the sequential and parallel performance of the FSM implementation, and Section 5 contains a brief summary and a discussion of future work.

1.2. Input File Format: ERDAS/Lan

The ERDAS/Lan file format consists of a 128-byte header containing selected map statistics as shown in Table 1. The data pack type, the number of columns, the number of rows, and the number of map classes are of particular interest to this research.

Following the ERDAS/Lan header are the pixel data values that are packed (pack type) in either 4 bits, 8 bits, or 16 bits and represent the pixel's membership in a *map class*. A map class is a collection of one or more attributes, or pixel characteristics, and is usually assigned a unique integer value. Map class membership is mutually exclusive; each pixel may belong to one and only one map class at any given time. Map classes can be visualized by assigning a unique color or gray-scale to each class and displaying the pixel data. The maximum number of map classes that can be represented in a map is a function of the number of bits in the pack type where a pack type of b bits per pixel can represent 2^b map classes. Figure 1 illustrates a section of an ERDAS/Lan map illustrating 31 map classes that are represented by 31 different gray-scales and the Northern Yellowstone National Park map, shown in Figure 2, illustrates 10 map classes. In this study, a pack type of 8 bits per pixel is used so that the number of map classes is limited to 256.

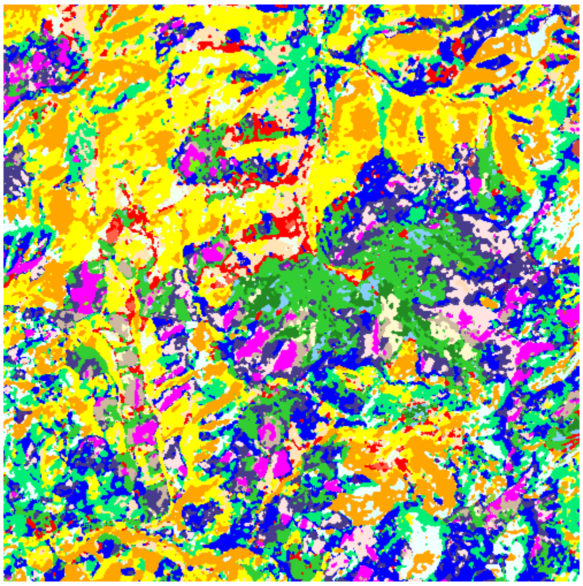

Figure 1. 1/64 of a 7570 × 7940 ERDAS/Lan map illustrating 31 map classes (**FORD** map).

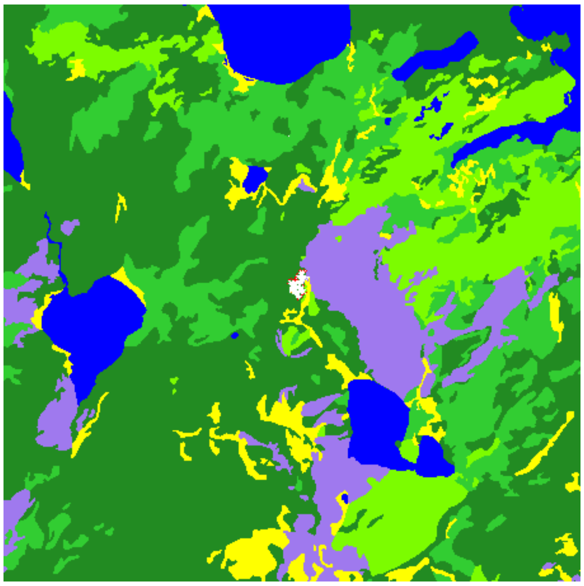

Figure 2. Map of Northern Yellowstone National Park illustrating 10 map classes (**FIRE** map).

Table 1. Contents of ERDAS/Lan 128-byte file header.

Bit Position	Data Type	Data Description
0–5	char	"HEADER" or "HEAD74"
6–7	integer	pack type of data
8–9	integer	number of bands per line
10–15		unused at this time
16–19	integer	width of the map in pixels
20–23	integer	length of the map in pixels
24-27	integer	database X-coordinate of the first pixel
28-31	integer	database Y-coordinate of the first pixel
32–87		unused at this time
88-89	integer	indicator for the type of map units
90–91	integer	the number of map classes
92-105		unused at this time
106–107	integer	the unit of area associated with each pixel
108–111	real	the number of area units represented by each pixel in the units given
112–115	real	the map X-coordinate for the center of the upper left corner pixel in the map
116–119	real	the Y-coordinate for the center of the upper left corner pixel in the map
120–123	real	the X-size of each pixel
124–127	real	the Y-size of each pixel

2. Hoshen-Kopelman Algorithm

This section introduces the Hoshen-Kopelman [6] cluster identification algorithm with particular emphasis given to the input datasets, programming data structures, the cluster neighborhood rule, and the finite-state-machine (FSM) implementation. The discussion of the FSM implementation focuses on the three major implementation components: the temporary label assignment, the search path compression, and the formal finite state machine.

2.1. Pre-Processing the ERDAS/Lan Maps

An ERDAS/Lan map is comprised of integer values ranging from zero to the number of map classes contained in the map. Each pixel contains either a zero or an integer representing the pixel's map class. The HK algorithm uses one data structure to store the map pixel values and as a working set to store temporary pixel label values. This dual purpose will not allow different map classes to be stored in the same structure without altering the cluster identification process. Thus, the finite-state-machine implementation of the Hoshen-Kopelman algorithm requires the input ERDAS/Lan map to be *pre-processed* in a way which alters the original data by filtering out the pixel data values not belonging to the selected map class. Only one map class can be represented in the input map for the finite state machine implementation. For example, to analyze map class 2, all of the map's pixel values equal to 2 would be changed to -1, and the rest of the pixel values in the map would be changed to 0. Figure 3 shows the results of pre-processing map class 2 of a simple ERDAS/Lan map file containing two map classes. The result is an input map containing only -1's and 0's where the -1's distinguish pixels belonging to the chosen map class.

2.2. Data Structures

The Hoshen-Kopelman (HK) algorithm [6] is a one-pass approach to cluster identification that utilizes two working arrays, **`matrix`** and **`csize`**, to store the post-processed input map[1] and

0	**1**	**1**	**0**	**0**	**2**	**2**	**0**
0	**1**	**0**	**0**	**0**	**0**	**2**	**0**
0	**1**	**1**	**0**	**2**	**2**	**2**	**0**
0	**0**	**0**	**0**	**0**	**2**	**0**	**0**
0	**2**	**2**	**0**	**0**	**2**	**0**	**0**
0	**2**	**1**	**0**	**0**	**0**	**0**	**0**

0	**0**	**0**	**0**	**0**	**-1**	**-1**	**0**
0	**0**	**0**	**0**	**0**	**0**	**-1**	**0**
0	**0**	**0**	**0**	**-1**	**-1**	**-1**	**0**
0	**0**	**0**	**0**	**0**	**-1**	**0**	**0**
0	**-1**	**-1**	**0**	**0**	**-1**	**0**	**0**
0	**-1**	**0**	**0**	**0**	**0**	**0**	**0**

Figure 3. Explanation of pre-processing map class 2 of a simple ERDAS/Lan map. Pixels assigned to map class 2 have a pixel value of 2 as seen in the before image (left). Pre-processing the map will change the pixel values, currently 2, to a -1 in the post-processed image (right) substituting zero for the rest of the values.

interim cluster statistics. The algorithm traverses the map from left to right, and then from top to bottom, assigning a temporary cluster label to each non-zero pixel and recording any pixel label or cluster membership changes in the two working arrays. Temporary label assignment is a function of the enforced neighborhood rule and the algorithm implementation, which are explained in detail in Sections 2.3 and 2.4, respectively. **`Matrix`** is a two dimensional integer array of size $(n+1)\times(m+1)$, where n is the width (columns) and m is the height (rows) of the map in pixels. The dimensions n and m are increased by 1 to accommodate boundary columns and rows, containing unique boundary integer values, that are necessary for this implementation. Six integer pointers (**`firstRow`**, **`secondRow`**, **`lastRow`**, **`current`**, **`west`**, and **`north`**) are maintained to traverse the array and make changes when necessary. Pointers are used to index the **`matrix`** array and to minimize the overhead of base-plus-offset array indexing. Integer storage (32 bits/pixel) is required because the **`matrix`** array serves a dual purpose. First, **`matrix`** holds the post-processed map data, and second, it serves as a working array for the pixels' temporary cluster label. If it is impossible to store the entire **`matrix`** array in memory, a *divide-and-conquer* approach can be implemented whereby cluster identification is performed by partitioning the map along row boundaries and merging the results of each partition. Each partition is of size $k \times m$ where k is the number of rows in each partition. If n is not evenly divisible by k, the last partition would have less than k rows.

An integer array (**`csize`**) is used to store either the running total of cluster size or an index value for another entry in the **`csize`** array. If the stored value is positive, it is the running total of the pixel membership in that cluster. If it is negative, the absolute value of the stored number is an index to the true cluster label. Negative indexes can follow a non-circular, recursive path for a finite number of steps. Figure 4 illustrates the **`csize`** working array displaying the positive or negative entries for each index in the array. Temporary cluster 1 shows a membership of 12 pixels whereas temporary cluster 2 is the beginning of a 5-step pointer path ($2 \rightarrow\ 4 \rightarrow\ 7 \rightarrow\ 6 \rightarrow\ 3$) pointing to temporary cluster 3 containing 22 pixels. The existence of the path signifies that the temporary cluster at the head of the path had been previously merged into the temporary cluster at the tail of the path. For example, as the result of four independent merge operations, temporary cluster 2

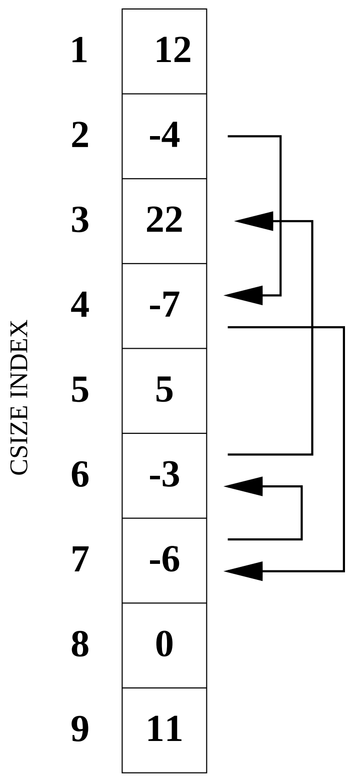

Figure 4. The **csize** working array illustrating either a positive or negative value for each index in the array. Positive values are a count of the pixels in the cluster labeled with the array index value and negative values are pointers to other indexes in the array. Negative indexes can follow a non-circular, recursive path for a finite number of steps.

had previously been merged into temporary cluster 3. Compression of this path is significant to the overall efficiency of the FSM implementation and is discussed in more detail in Section 2.5.2.

2.3. Neighborhood Rule

A pixel's membership in a cluster depends on the neighborhood rule used. This research employs the North-East-West-South (NEWS) neighborhood rule in which two pixels are included in the same cluster if both the pixels are in the same class and the pixels are neighbors according to the neighborhood rule. NEWS neighbors share a pixel boundary on the north, east, west, or south. Consider the map as a 2-dimensional matrix of size $n \times m$ and the current pixel is **map(i,j)** where $0 \leq i \leq n$ and $0 \leq j \leq m$. The north, east, west, and south neighbors would have the coordinates **map(i,j-1)**, **map(i+1,j)**, **map(i-1,j)**, and **map(i,j+1)**, respectively (see Figure 5).

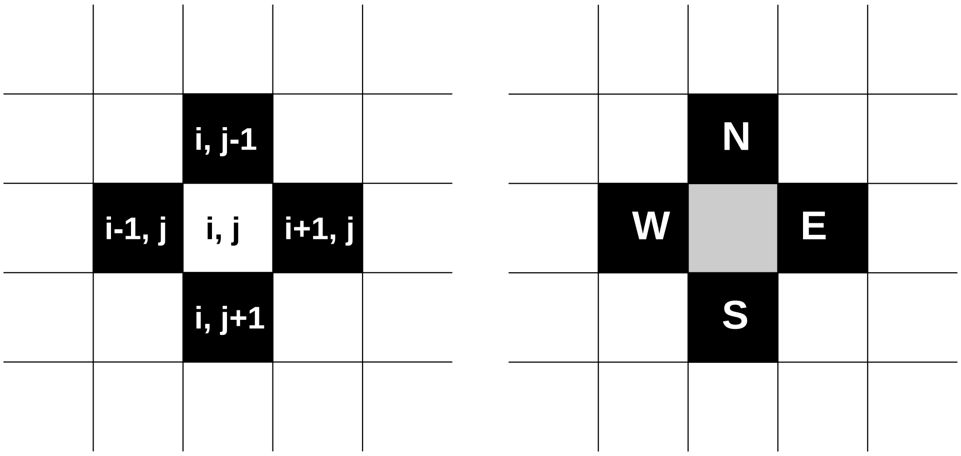

Figure 5. North, east, west, south relationship to `pixel(i,j)`.

2.4. Implementing the Hoshen-Kopelman Algorithm

As each row of the map is traversed, the current pixel value is compared to the previous pixel value in that row (west neighbor) and the pixel value in the same column of the previous row (north neighbor). Abiding by the NEWS neighborhood rule, if all pixels are in the same cluster, the smallest cluster label is assigned to all three pixels and any changes in the status of the west and north neighbors are recorded in the working arrays. The smallest of the two values is chosen to reduce the memory requirements of the **`csize`** array where the maximum size is a function of the density of the map and the pixel clustering patterns. For the best use of memory by the **`csize`** array, the best case scenario is a map entirely of 1's or 0's which would contain only 1 cluster of size $n \times m$. The **`csize`** array size would be 1. Conversely, the worst case scenario is a checkerboard pattern, alternating 1's and 0's throughout the map, resulting in $(n \times m)/2$ clusters each containing only 1 pixel. The **`csize`** array size would be $(n \times m)/2$. When the algorithm completes, each index in the **`csize`** array holds one of three values: a zero, the number of pixels in the cluster identified by the **`csize`** index value, or a pointer to another **`csize`** index. The last two values are differentiated by the arithmetic sign of the value—positive represents the pixel count in the indexed cluster whereas negative represents a pointer to another **`csize`** array index. De-referencing the pointer, which could follow a non-circular, multi-step path, is accomplished by indexing into the **`csize`** array using the absolute value of the pointer. The HK algorithm does not guarantee a properly labeled map; however, relabeling, or adjusting pixel values to accurately reflect cluster membership, can be accomplished in linear time using the working array produced by the HK algorithm. Figure 6 illustrates the results of the working array **`csize`** after merging the temporary cluster labeled 2 from the top row into the temporary cluster labeled 1. Before the merge, temporary cluster 1 contained 9 pixels, and temporary cluster 2 contained 2 pixels. After the addition of the current pixel (1 pixel) and the pixels from temporary cluster 2 (2 pixels), temporary cluster 1 now contains 12 pixels $(9 + 2 + 1)$, and temporary cluster 2 shows a pointer to temporary cluster 1 in the **`csize`** working array. It is possible to process extremely large maps in linear time using a divide-and-conquer approach to cluster identification because at a point in time, the algorithm only requires access to the working array **`csize`**, the current pixel, the north neighboring pixel, and the

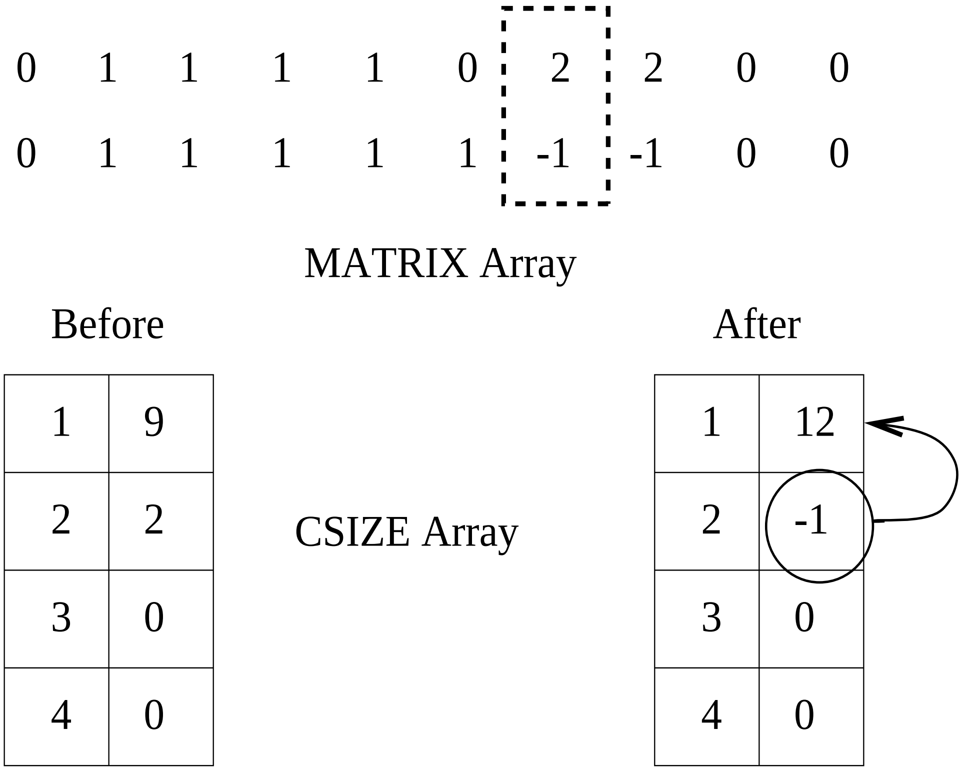

Figure 6. Example of working arrays **`csize`** and **`matrix`** manipulation during the implementation of the Hoshen-Kopelman algorithm. The *After* image of the **`csize`** array illustrates the results of merging the temporary cluster labeled 2 into the temporary cluster labeled 1.

pixel value of the west neighbor. In this work, both an improved sequential implementation of the HK algorithm and a parallel implementation on the Thinking Machines CM–5 are presented.

2.5. Finite State Machine Implementation

The objective of the Hoshen-Kopelman finite state machine (FSM) implementation is to reduce the number of integer comparison operations thus reducing the overall cluster identification time for a particular map. Particular emphasis is given to the integer comparison operations found in the temporary label determination and the cluster-merge functions of the algorithm. The objective is reached by encapsulating the label value and cluster membership status of the west neighbor in a state of the FSM and by compressing the recursive pointer path in the **`csize`** array formed during the merge-cluster function of the algorithm implementation. Table 2 details the information encapsulated in each state of the Hoshen-Kopelman finite state machine.

Table 2. Cluster information encapsulated by each state of the FSM.

State	Encapsulated Information
S_0	Not currently in a cluster
S_1	Currently in a cluster on this line
S_2	Currently in a cluster on previous line
S_3	At a map boundary
S_4	Finished

2.5.1. Temporary Label Determination

When evaluating a pixel for membership in a cluster, one must first determine if the west and north neighbors have already been assigned a temporary cluster label and, if so, determine their minimum cluster label. This requires at least two comparison operations: (1) is the west neighbor's pixel value equal to 0, and (2) is the north neighbor's pixel value equal to zero. Once the cluster membership has been determined, the appropriate **`csize`** array values must be adjusted to reflect the changes in the status (the pixel label value) of the west and north neighbors. In the FSM implementation, the value of the west neighbor's pixel is stored in a local variable and the west neighbor's cluster membership status is encapsulated in a state of the FSM.

2.5.2. Search Path Compression

The HK algorithm traverses the map assigning temporary cluster labels to non-zero pixels in the order in which the pixels are encountered. Once it has been determined that two temporary clusters are actually the same cluster, a merge operation is performed. Figure 7 illustrates the effects of merging temporary cluster 2 into temporary cluster 1 and path compression on the working array **`csize`**. The resulting temporary cluster 1 then contains 34 pixels. Temporary cluster 1 contributed 12 pixels and temporary cluster 3 contributed 22 pixels. Temporary cluster 2 contains a pointer to cluster 1 which is differentiated by the negative arithmetic sign. During the merge operation, the algorithm followed a 5-step pointer path ($2 \rightarrow 4 \rightarrow 7 \rightarrow 6 \rightarrow 3$) to determine that temporary cluster 2 had previously been merged into temporary cluster 3. Pointer paths are non-circular and contain a finite number of steps. Path compression reduces the number steps in any given path by replacing each pointer along the path with a pointer to the destination of the path. After path compression, the next time the algorithm encounters a merge involving cluster 2 the algorithm will only have to follow a 1-step path. This approach will always reduce the steps of a given path to only 1, which decreases the number of integer comparisons and overall cluster identification time.

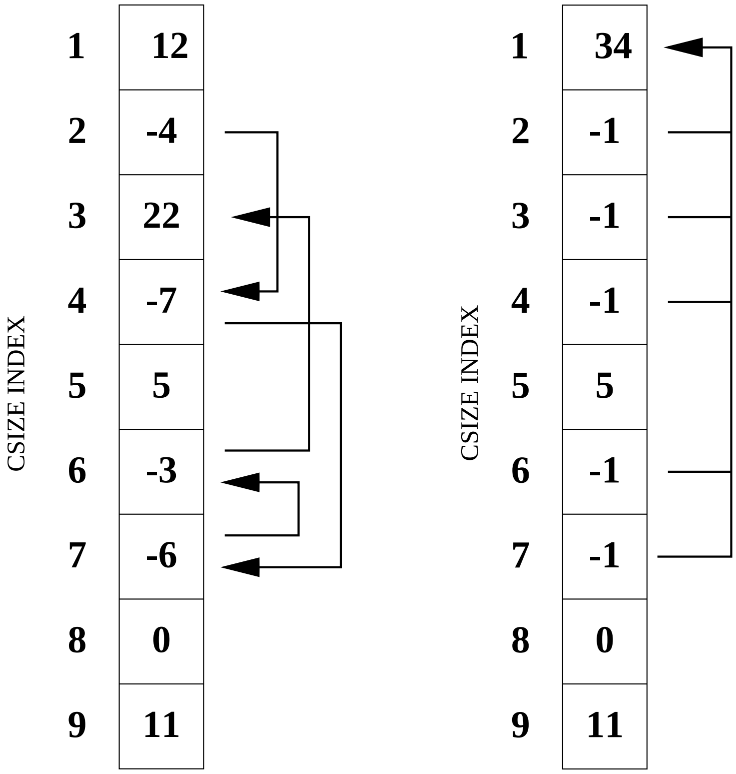

Figure 7. Example of search path compression in the **csize** array before (left) and after (right) a merge operation. The operation merged the temporary cluster labeled 2 into the temporary cluster labeled 1. This operation followed a 5-step pointer path ($2 \rightarrow 4 \rightarrow 7 \rightarrow 6 \rightarrow 3$) to determine that temporary cluster labeled 2 was actually temporary cluster labeled 3 and contains 22 pixels.

2.6. The Finite State Machine

A FSM, *finite automaton*, is a logical construct composed of a set of *states*, a finite input *token alphabet*, and a transition function to map *states* $\times$ *tokens* to *states*. According to [5], a *finite automaton* is formally denoted by

> a 5-tuple (Q, Σ, δ, q_0, F), where Q is a finite set of *states*, Σ is a finite *input alphabet*, q_0 in Q is the *initial state*, $F \subseteq Q$ is the set of *final states*, and δ is the *transition function* mapping $Q \times \Sigma$ to Q. That is, $\delta(q, a)$ is a state for each state q and input symbol a.

The *transition function*, δ, is defined as a set of 3-tuples (S_c, a, S_n) where S_c is the current machine state, a is an element of the *token alphabet*, and S_n is the new machine state. Table 3 formally defines the Hoshen-Kopelman finite state machine (FSM) used in this work. The FSM begins in S_0 and continues until a final state is reached. Section 7 of the Appendix lists the FSM cluster identification (**cluster_id**) and the cluster relabeling (**reLabel**) source code. The states in Q encapsulate whether or not the west neighbor had been assigned a temporary cluster label eliminating the redundant analysis of the west neighbor. Eliminating redundant conditionals increases the efficiency

Table 3. Formal definition of the Hoshen-Kopelman finite state machine.

FSM = ($\mathcal{Q}$, Σ, δ, q_0, F) where	
$\mathcal{Q}$ =	{ S_0, S_1, S_2, S_3, S_4 }
Σ =	{ -1, B } ∪ { 0, . . . ,Maximum Pixel Label }
δ =	{(S_0, $0/x$, S_0), (S_0, $-1/0$, S_1), (S_0, $-1/!0$, S_2), (S_0, B/x, S_3), (S_1, $0/x$, S_0), (S_1, $-1/0$, S_1), (S_1, $-1/!0$, S_2), (S_1, B/x, S_3), (S_2, $-1/!0$, S_2), (S_2, $0/x$, S_0), (S_2, $-1/0$, S_1), (S_2, B/x, S_3), (S_3, $!B/x$, S_0), (S_3, B/x, S_4) }
q_0 =	{ S_0 }
F =	{ S_4 }

Assumptions:
B is the pixel value of the map boundary,
! is the unary negation operator,
x is north pixel value, and
$x \in$ {0, . . . , Maximum Pixel Label and B}

of the FSM implementation. Table 4 is a working example of the FSM implementation showing the formal state changes for one row of a sample map.

3. Performance Methodology

This section presents the computing environment and performance methodology for the sequential and parallel implementation of the Hoshen-Kopelman cluster identification algorithm. Whenever the sequential and parallel implementations differ significantly, they are discussed independently.

3.1. Computing Environment

Sequential results were obtained using a Sun Microsystems SPARCstation 10, model 30 with an internal chip speed of 33 MHz and an on-chip cache of 36 KB. The processor architecture is the SuperSPARC™. The operating system was SunOS Release 4.1.3 and the machine had 128 Mb of physical memory.

The multiple-instruction-multiple-data (MIMD) architecture used for parallelizing cluster identification was the Thinking Machines 32-processor CM–5. The CM–5 is a scalable parallel processing computer using the CMOST operating system which provides both time-sharing and space-sharing. The CM–5 consists of groups of processing nodes (PNs) under the control of a control processor (CP) also known as the partition manager (PM). The group of PNs and the PM are collectively called a partition with a size ranging from tens to thousands of processors. The processors in a 32-node CM–5 are denoted as $PN_0, \ldots, PN_{31}$. Each partition runs the CMOST operating system—an enhanced version of UNIX where time-sharing is the natural mode. Although not exploited in this study, each PN has four vector unit accelerators (VUs) for enhanced arithmetic performance.

Programming on the CM–5 utilizes one of two paradigms, *host/node* and *hostless*. In the host/node paradigm, the CP acts as the host for the program controlling the program flow and the workload assigned to each PN. The host is also responsible for message-passing control and servicing the input/output (I/O) requests generated by the PNs. Furthermore, access to the partition's PNs is only accomplished through the CP. In this paradigm the program is not limited to executing

Table 4. Incremental state changes for the FSM working example.

Transition Function	**`Matrix`** Array Values							**`Csize`**	
STEP 1									
$(S_0, 0/x, S_0)$	$\boxed{1}$	1	0	0	2	2	0	1	$\boxed{2}$
	$\boxed{0}$	-1	-1	-1	-1	-1	0	2	$\boxed{2}$
STEP 2									
$(S_0, -1/!0, S_2)$	1	$\boxed{1}$	0	0	2	2	0	1	$\boxed{2}$
	0	$\boxed{-1}$	-1	-1	-1	-1	0	2	$\boxed{2}$
STEP 3									
$(S_2, -1/0, S_1)$	1	1	$\boxed{0}$	0	2	2	0	1	$\boxed{3}$
	0	1	$\boxed{-1}$	-1	-1	-1	0	2	$\boxed{2}$
STEP 4									
$(S_1, -1/0, S_1)$	1	1	0	$\boxed{0}$	2	2	0	1	$\boxed{4}$
	0	1	1	$\boxed{-1}$	-1	-1	0	2	$\boxed{2}$
STEP 5									
$(S_1, -1/!0, S_2)$	1	1	0	0	$\boxed{2}$	2	0	1	$\boxed{5}$
	0	1	1	1	$\boxed{-1}$	-1	0	2	$\boxed{2}$
STEP 6									
$(S_2, -1/!0, S_2)$	1	1	0	0	2	$\boxed{2}$	0	1	$\boxed{8}$
	0	1	1	1	1	$\boxed{-1}$	0	2	$\boxed{-1}$
STEP 7									
$(S_2, 0/x, S_0)$	1	1	0	0	2	2	$\boxed{0}$	1	$\boxed{9}$
	0	1	1	1	1	1	$\boxed{0}$	2	$\boxed{-1}$

the same program on each of the PNs. A different program may be executing on independent data on each of the PNs in the partition (MIMD).

In the hostless paradigm, the CM–5 is operating in SIMD mode. All of the PNs used by the program will be executing the same code on either the same or different data. The actual amount of work performed by each PN is determined either by data inconsistencies or direct programming references. Because access to the PNs requires a CP, the CMOST operating system provides a limited host program to control the message passing and I/O service request generated by the PNs.

In both situations communication between the CP and the PNs, or between individual PNs, is accomplished through message-passing routines that are supplied in a communications library, CMMD, as an enhancement to one of the supported programming languages. CMMD is primarily used for interprocessor communication—message passing between nodes in the hostless model and between host and nodes in the host/node mode [8]. Both the sequential and parallel implementations use the C++ programming language and version 2.6.3 of the g++ compiler.

3.2. Performance Methodology

The methodology used to asses the performance of cluster identification can be divided into three phases: algorithm selection, algorithm implementation and verification, and data collection and interpretation. The selection, verification, and data collection phases are applicable to both sequential and parallel implementations and are discussed simultaneously. Significant differences in the sequential and parallel implementations, however, are discussed independently.

3.2.1. Algorithm Selection

In [2], the Hoshen-Kopelman (HK) algorithm was shown to be the most efficient sequential cluster identification algorithm. The algorithm's one-pass aggregate mechanism does not require access to an entire input map at any given time, which makes it well suited for a divide-and-conquer approach for cluster identification on large-scale maps. A map can be partitioned along row boundaries allowing the algorithm to operate on a map iteratively—as if viewing through a sliding window. As the algorithm reaches the end of a partition, the sliding window reveals the next partition and allows the algorithm to continue as though it had access to the entire map. Furthermore, cluster identification on a partitioned map does not alter the performance nor the result of the algorithm when compared to cluster identification on a non-partitioned map. This inherent aggregate mechanism can be exploited to accommodate sequential cluster identification on large-scale maps or cluster identification using a coarse-grained, parallel approach on the CM–5. Hence, the inherent aggregate mechanism was the determining factor for the selection of the HK algorithm.

3.2.2. Algorithm Verification

The accuracy of the sequential and parallel implementations was verified using randomly generated test maps with known cluster statistics including the number of clusters, the size of the largest cluster, the average cluster size, and the map's non-zero pixel density. The square test maps contained either 64, 126, 256, 512, or 1024 rows and columns with a non-zero pixel density of 10%, 30%, 62%, or 85%.

3.2.3. Sequential Implementation

The algorithm implementation involved reading the header information from the input ERDAS/Lan map file, partition size determination, reading the input map, performing cluster identification, and tabulating the results. In the sequential implementation (SI), reading the header information and map data were performed using the C++ language file input/output functions. The map header contains sufficient information to determine the appropriate size of the data structures for the **`csize`** and **`matrix`** working arrays. The number of rows read and processed by the SI was limited[2] to 1024. If the number of rows exceeds 1024, the algorithm will partition the map and perform cluster identification on each of the partitions in sequential order. After the algorithm completes, the **`csize`** array on PN_0 contains the appropriate number of pixels in each cluster and the **`matrix`** array contains the temporary cluster label for each non-zero pixel in the input map. Final cluster statistics are obtained by traversing the **`csize`** array and tabulating the positive non-zero values.

3.2.4. Parallel Implementation

The algorithm was parallelized by partitioning the map across the PNs of the CM–5, performing cluster identification on each partition, and merging the cluster results of the partitions. The parallel mechanism for cluster identification involved four processes: mutually exclusive cluster identification on a partition assigned to each PN, building a list of clusters that intersect partition boundaries, merging the partitions' cluster information, and performing final cluster merges and statistics calculations.

Partitioning the map across the PNs of the CM–5 involved the basic input/output functions of the operating system. In the parallel implementation (PI), I/O functions were performed in either Synchronous Broadcast Mode (SBM) or Synchronous Sequential Mode (SSM). SBM requires all PNs to issue a synchronized I/O function call, that is implemented by having one PN perform the physical I/O operation followed by a global message broadcast to the other PNs using the CM–5's internal message passing facilities. This procedure results in each PN receiving/writing the same information and helps to avoid physical disk contention.[3] SSM allows an input data set to be sequentially divided across the PNs. For example, given 32 PNs and $32 \times b$ bytes to read: PN_0 would read the first b bytes, PN_1 would read the next b bytes, etc., so that PN_{31} would read the last b bytes. While SBM was used to read and distribute the 128-byte header to all PNs, SSM was used to read and partition the input map file across all PNs.

Cluster identification on each PN was a mutually exclusive process—totally independent of other PNs and partitions. Each PN, using a **`csize`** array that is sufficient to hold the statistics for the entire map, is allowed access to a specific range of indexes in their **`csize`** array. The beginning cluster label number for each PN must agree with the starting index of the assigned range. Mapping PN numbers to a unique range of indexes in the **`csize`** array protects against the assignment of duplicated temporary cluster labels across partitions. The absence of duplicate label numbers is significant in order for the CMOST's global integer reduction operation (GIRO) to yield meaningful results. The GIRO reduces 1 object on each PN returning the result of the operation to the object on each PN. The GIRO requires access to the object on each PN and a binary operator to apply during the reduction operation. For example, given an integer object i on p PNs, each with a value of 1 and a binary operator of *addition*, the result of the GIRO would have been $i = p$ on each PN. If the binary operator was *multiplication*, the result would have been $i = 1$ on each PN. The effective result of cluster identification is a **`csize`** array on each PN containing the complete map's temporary cluster statistics. Figure 8 illustrates a simple example of index range assignment on a 4-processor system. PN_0 is assigned the first 25% of the **`csize`** indexes, PN_1 is assigned the second 25%, PN_2 is assigned the third 25%, and PN_3 is assigned the last 25%. After the CMMD global integer reduction operation, all of the **`csize`** arrays contain the union of all of the **`csize`** arrays.

During cluster identification, all PNs have relabeled the first row (**`firstRow`**) and last row (**`lastRow`**) in its partition. Afterwards, each PN can then send a copy of **`lastRow`** to the PN with the next higher number. The relabel process uses the pixel value as an index into the **`csize`** working array and checks the arithmetic sign of the array stored value. If the stored value is positive, the pixel is correctly labeled. Otherwise, the correct label value is at the tail of the linked list created with the negative array values (see Figure 7). Each PN compares the row it receives to its relabeled **`firstRow`** and generates a list of *merge-clusters* ordered pairs. A merge-clusters ordered pair contains two temporary cluster labels and is maintained in an integer array, **`mergeArray`**, with each ordered pair occupying 2 successive array slots. Figure 9 illustrates the merge-clusters ordered pair generation by PN_1 after receiving **`lastRow`** from PN_0. PN_1 receives **`lastRow`** from PN_0 into **`messageRow`**, and then compares **`messageRow`** and **`firstRow`** on PN_1. Temporary clusters 12 and 13 from PN_0 overlap with temporary cluster 24 from PN_1; therefore, PN_1 would insert 12, 24, 13, and 24 respectively into **`mergeArray's`** next four available array slots. Figure 10 illustrates the resulting **`mergeArray`** on PN_1 for the example in Figure 9. Zero entries in indexes 4 and 5 are termination indicators that signal PN_0 to stop processing this array during the final merge operation. Using CMMD message receiving function (**`CMMD_receive_block`**), PN_0 receives the **`mergeArray`**

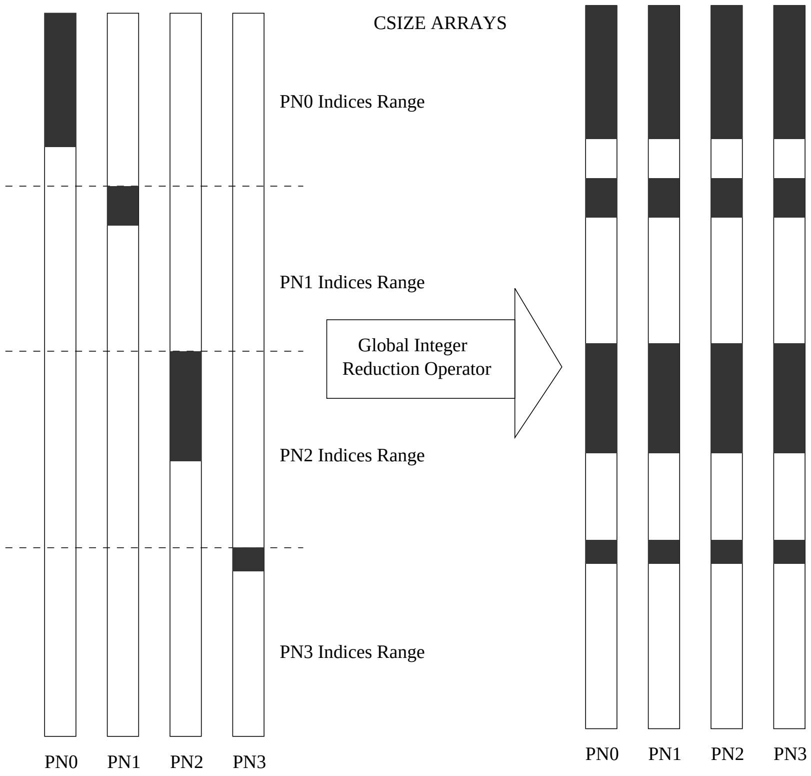

Figure 8. Illustration of index ranges in the **`csize`** array on a 4-PN system and the results of a global integer reduction operation (GIRO).

of merge-clusters ordered pairs from all other PNs and then performs a merge operation on the **`csize`** array for each merge-clustered ordered pair. For example, when PN_0 receives **`mergeArray`** from PN_1 containing 12, 24, 13, and 24, PN_0 will merge temporary cluster 12 into temporary cluster 24 and temporary cluster 13 into temporary cluster 24. The result of the identification process is a **`csize`** array on PN_0 containing the cluster statistics for the entire input map.

4. Performance Results

In this section, the performance of the sequential and parallel implementations of the FSM-based Hoshen-Kopelman algorithm is presented. Actual timing results obtained on a Sun SPARCstation 10 and Thinking Machines CM-5 are provided. The FSM sequential implementation is compared to the Hoshen-Kopelman cluster identification *textbook* implementation from [1], which was coded verbatim from the algorithm presented in [6]. The FSM sequential and parallel performance results will be compared for accuracy and *speedup*—the reduction in execution time due to the code parallelization.

PN	Row Pointer	`matrix` Array						
					⋮			
PN_0		12	12	12	0	13	0	10
PN_0	**`lastRow`**	0	12	12	0	13	13	0
				↓ **`CMMD_send_block`** ↓				
PN_1	**`messageRow`**	0	12	12	0	13	13	0
				↓ is compared to ↓				
PN_1	**`firstRow`**	0	0	24	24	24	0	0
PN_1		26	0	24	0	0	0	27
					⋮			

Figure 9. Illustration of the merge-clusters ordered pair generation. **`LastRow`** is passed from PN_0 and received by PN_1 into **`messageRow`**. **`messageRow`** is then compared to **`firstRow`** to determine overlapping clusters.

Index	Entry
0	12
1	24
2	13
3	24
4	0
5	0
⋮	⋮

Figure 10. Illustration of the merge-clusters ordered-pair array, **`mergeArray`**.

4.1. Comparison Maps

Two input maps were used for comparing the FSM sequential implementation to the HK implementation in [1] and to the FSM parallel implementation on the CM–5. The first map is a 454×454 ERDAS/Lan map (**`FIRE`**) containing fire patterns from the Northern Yellowstone National Park. The **`FIRE`** map is composed of ten map classes each with different pixel densities, number of clusters, and cluster sizes. Table 7 in Section 6 of the Appendix lists the map class statistics of the **`FIRE`** map which was supplied by Dr. Robert Gardner of the Oak Ridge National Laboratory, Environmental Science Division [4]. The second map is a 7570×7940 ERDAS/Lan map (**`FORD`**) containing pre-classification values, obtained from applying a *classification function* on four or five of the inputs from thematic mapper imagery [3]. The original inputs are sensor readings (wavelengths) from different parts of the light spectrum. The pixel values in the input map are artificial values that represent map class membership. The values are logically significant, but not numerically. A data value of 8 is not necessarily closer to a data value of 9 than it is to any other value (0–31). Table

8 in Section 6 of the Appendix lists the map class statistics for the **FORD** map, which was supplied by Dr. Ray Ford from the Department of Computer Science at the University of Montana and the Wildlife Spatial Analysis Lay directed by Dr. Roland Redmond at the University of Montana.

4.2. Sequential Performance Results

As stated in Section 1.1, the primary objective of the FSM sequential implementation (SI) was to optimize a sequential cluster identification algorithm for a subsequent parallel implementation (PI) on the CM–5. This objective was accomplished by implementing the HK cluster identification algorithm with efficient use of pointers and utilizing a finite state machine to eliminate redundant integer comparison operations. The FSM implementation was compared to the HK implementation in [1] and Figure 11 illustrates the *time improvements* for the FSM implementation. The time improvement is the ratio of the execution time for the original HK implementation in [1] to the FSM execution time and is attributed to the efficient use of pointers during array indexing, compression of the search path during cluster merge operations, and using a finite state machine to reduce redundant integer comparison operations by encapsulating the previous comparison result in a machine state. The FSM implementation achieved a minimum time improvement of 1.39 and a maximum time improvement of 2.00 on the **FIRE** map. Table 9 in Section 6 of the Appendix lists the time improvement for each map class of the **FIRE** map. The **FORD** map was not used for sequential comparisons because the previous implementation in [1] was designed to handle maps smaller than 1024×1024 pixels.

4.3. Parallel Performance Results

The success of the parallel implementation (PI) was evaluated by comparisons with the sequential implementation (SI) in two categories: accuracy and execution time speedup. Speedup is defined as the ratio of the sequential time to the parallel time and was calculated using the 7570×7940 **FORD** map. Because the HK algorithm's working arrays (**csize** and **matrix**) were distributed across the PNs of the CM–5, the PI functions **SEND**, **IDROWS**, **REDUCTION**, **MERGEFUN**, **ROW1**, and **CALCSIZE** were necessary to duplicate the functionality of the sequential HK algorithm. Table 5 details the responsibility of each of the functions involved in cluster identification in the FSM parallel implementation. The total time spent in cluster identification in the PI is the sum of the time spent in the **SEND**, **IDROWS**, **REDUCTION**, **MERGEFUN**, **ROW1**, and **CALCSIZE** functions (see Table 6).

The PI was able to achieve a speedup ranging from 5.41 to 5.90 on the **FORD** map. The average speedup was 5.71 for the 31 layers of the **FORD** map. Given the parallel potential of the HK algorithm, the performance of the PI met expectations. The HK algorithm only requires access to a small portion of the input map and the process of merging overlapping clusters is well defined. The PI takes advantage of the PN's local memory to partition the input map which enables cluster identification to work entirely in parallel. However, the separation of the data and working sets became a hindrance during the **MERGEFUN** process. The combination process, merging information from each PN, turned out to be more expensive than the cluster identification process.

The combination process consumes 48% of the total cluster identification time compared to the cluster identification's 35%. Future research concerning the combination process is discussed in the next and final section. The SI and PI results for the 31 map classes of the **FORD** map, which are detailed in Table 10 (see Section 6 in the Appendix) and Figure 12, demonstrate a constant speedup factor near 6.

Table 5. Functional responsibilities of the functions involved in cluster identification in the FSM parallel implementation on the CM–5.

Function	Responsibility
ROW1	correctly labels the first row of the PN's partition
CLUSTER	performs cluster identification of the PN's partition
SEND	sends a copy of the last row in each PN to the next PN
IDROWS	compares the row received during the **SEND** function to the first row in the current PN's partition and generates a list of merge-cluster ordered pairs (see Section 3.2.4)
REDUCTION	combines the individual **csize** arrays located on each PN into one **csize** array on PN_0 and is implemented using a CMMD global integer reduction operation
MERGEFUN	is responsible for resolving the list of merge-cluster ordered pairs generated by the PNs and is performed entirely on PN_0
CALCSIZE	uses the **csize** array to calculate the cluster statistics for the entire map and is performed entirely on PN_0

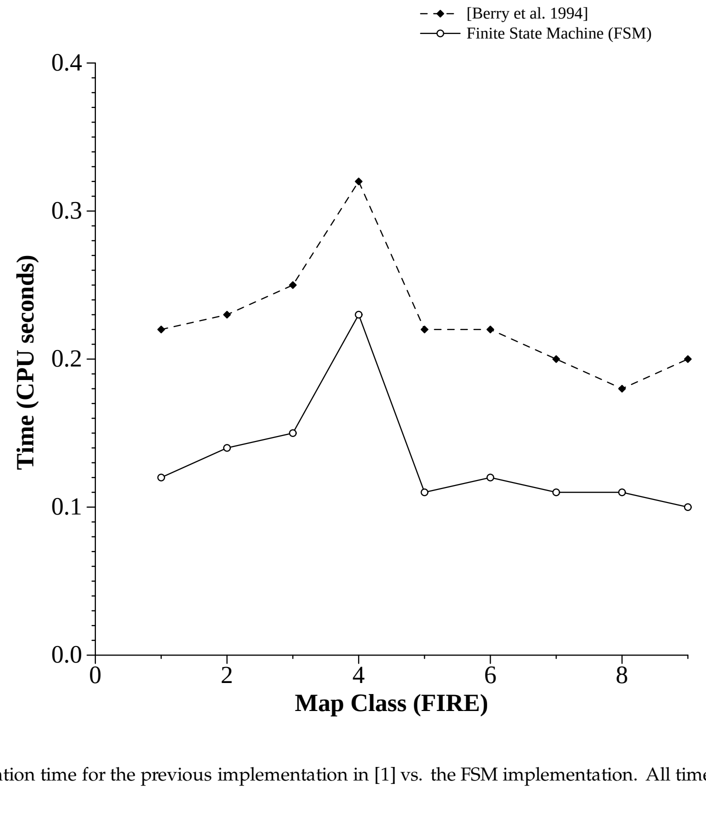

Figure 11. Cluster identification time for the previous implementation in [1] vs. the FSM implementation. All times are in CPU seconds.

Table 6. Functional profile of the Parallel Implementation (PI) of the FSM cluster identification algorithm applied to the **FORD** map.

Function	Elapsed Time
CLUSTER	1.30545
SEND	0.00961
IDROWS	0.00607
REDUCTION	1.80424
MERGEFUN	0.12524
ROW1	0.00423
CALCSIZE	0.44157
Total Cluster Id	3.69641

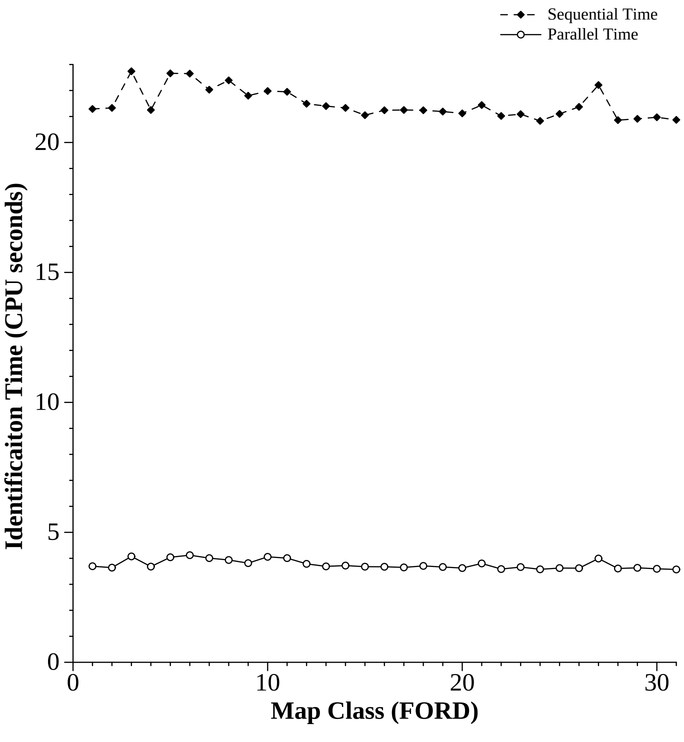

Figure 12. Comparison of parallel and sequential cluster identification times for 31 map classes of the **FORD** map. All times are elapsed CPU seconds.

5. Summary and Future Work

In this work, a faster sequential implementation (SI) of the Hoshen-Kopelman cluster identification algorithm and a parallel implementation (PI) on the CM–5 have been presented. The implementations involved using a finite state machine to maintain results of prior integer comparisons in order to reduce redundant comparison. The SI achieved a *time improvement* ranging from 1.39 to 2.00 over the sequential HK implementation in [1] and the PI achieved an average *speedup* of 5.71 over the fastest SI. Future optimization of particular PI functions, **`REDUCTION`** and **`MERGEFUN`**, is possible by examining the message passing mechanism in the CMOST operating system with a goal of reducing the total time spent in inter-processor communication.

Improving the combination mechanism for the FSM parallel implementation is another future concern. Since the PI creates a **`csize`** working array on each PN, the range of indexes into the array (based on the PN number) is limited. For example, in a 32 PN system, PN_0 would have access to the first 1/32 indexes in the **`csize`** array on PN_0. The rest of the indexes would remain unused. This mechanism was chosen to make use of the global reduction operations available in the CMOST operating system. The global reduction operation is responsible for 48% of the overall cluster identification time which is more expensive than the actual cluster identification function. Furthermore, the extra memory allocated in the **`csize`** working array on each of the PN could be better used to process larger input maps.

A possible solution could be to switch to a host/node paradigm whereas the host PN would allocate a **`csize`** working array large enough to support the entire map; but, the PNs would only allocate enough memory to support the PN's portion of the input map. The **`MERGEFUN`** would then be performed entirely on the host processor. The **`REDUCTION`** process would be more complicated because the host would have to map the individual **`csize`** array index numbers to a specific range of indexes; but, it should reduce the time spent passing messages.

Notes

1. Post-processed refers to the map resulting from pre-processing—filtering data values not corresponding to the chosen map class. The original input ERDAS/Lan map is read into a holding buffer, filtered, and moved into the **`matrix`** array for final processing. This three-step process is referred to as pre-processing.
2. This limitation is due to limited physical memory resources, i.e., random access memory and virtual memory swap space on the Sun SPARCstation's local hard disk drive.
3. Physical disk contention is a phenomenon observed when the number of read or write operations targeted at a physical hard disk drive are more than can be serviced simultaneously.

References

1. Berry, M., Comiskey, J., and Minser, K. 1994. Parallel Analysis of Clusters in Landscape Ecology. *IEEE Computational Science and Engineering 1,* 2, 24–38.
2. Comisky, E. 1993. Data-Parallel Implementations of Map Analysis and Animal Movement for Landscape Ecology Models. Technical Report CS–93–207 (August), University of Tennessee, Knoxville, Tennessee.
3. Ford, R., Running, S., and Nemani, R. 1994. A Modular System for Scalable Ecological Modeling. *IEEE Computational Science & Engineering 1,* 3, 32–44.
4. Hargrove, W., Gardner, R., Turner, M., Romme, W., and Despain, D. 1993. Simulating Fire Patterns in Heterogeneous Landscapes: the analysis and interpretation of landscape heterogeneity. Preprint.
5. Hopcroft, J. and Ullman, J. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, Reading, Massachusetts.
6. Hoshen, J. and Kopelman, R. 1976. Percolation and Cluster Distribution. I. Cluster Multiple Labeling Technique and Critical Concentration Algorithm. *Phys. Rev. B. 1,* 14 (October), 3438–3445.
7. Thinking Machines Corporation 1993a. *Connection Machine CM-5 Technical Summary*. Cambridge, Massachusettes: Thinking Machines Corporation.
8. Thinking Machines Corporation 1993b. *Connection Machine CM-5, User's Guide*. Cambridge, Massachusettes: Thinking Machines Corporation.

Appendices

6. Data Tables

Table 7. Statistics for the 9 map classes of the **`FIRE`** map (454 × 454 pixels).

Map Class	Clusters	Max Cluster	Avg. Cluster	Density (%)
1	29	10525	666	9
2	55	15403	470	12
3	66	10329	530	16
4	107	67920	903	46
5	73	1443	137	4
6	19	5395	999	9
7	19	9	1	0
8	1	1	1	0
9	0	0	0	0

Table 8. Statistics for the 31 map classes of the **FORD** map (7570 × 7940 pixels).

Map Class	Clusters	Max Cluster	Avg. Cluster	Density (%)
1	18762	191247	41	1
2	12255	20354	50	1
3	166045	9983	24	6
4	35191	8480	17	1
5	156346	22971	27	7
6	235361	31644	18	7
7	184364	7337	12	3
8	84961	41615	48	6
9	71257	52544	29	3
10	217712	11838	12	4
11	212811	12576	11	4
12	102649	3255	11	2
13	35033	5357	21	1
14	64058	3079	9	1
15	32157	4202	15	0
16	33378	4400	13	0
17	23302	8859	22	0
18	44676	3767	16	1
19	31155	7394	11	0
20	29602	1434	5	0
21	108455	3660	9	1
22	7116	381	3	0
23	40589	5934	6	0
24	380	1199	20	0
25	18550	6868	11	0
26	24688	3677	4	0
27	209065	8800	10	3
28	18938	1146	4	0
29	25183	3337	5	0
30	1165	3495	39	0
31	0	0	0	0

Table 9. Comparison of cluster identification time: original HK implementation in [1] vs. the FSM implementation on the **FIRE** map.

Class	Original HK (CPU sec.)	FSM (CPU sec.)	Speedup
1	0.22	0.12	1.83
2	0.23	0.14	1.64
3	0.25	0.15	1.67
4	0.32	0.23	1.39
5	0.22	0.11	2.00
6	0.22	0.12	1.83
7	0.20	0.11	1.81
8	0.18	0.11	1.63
9	0.20	0.10	2.00

Table 10. Comparison of cluster identification CPU time: FSM sequential implementation vs. the FSM parallel implementation on the **FORD** map.

Map Class	FSM Sequential Time (CPU seconds)	FSM Parallel Time (CPU seconds)	Cluster ID Time *Speedup*
1	21.29	3.70	5.75
2	21.33	3.64	5.86
3	22.74	4.07	5.58
4	21.25	3.68	5.77
5	22.66	4.04	5.61
6	22.65	4.12	5.50
7	22.03	4.01	5.50
8	22.39	3.94	5.68
9	21.80	3.82	5.71
10	21.98	4.06	5.41
11	21.95	4.01	5.47
12	21.49	3.79	5.67
13	21.40	3.70	5.78
14	21.33	3.72	5.73
15	21.05	3.68	5.72
16	21.24	3.68	5.77
17	21.25	3.65	5.82
18	21.24	3.71	5.72
19	21.19	3.67	5.77
20	21.12	3.67	5.75
21	21.44	3.80	5.64
22	21.02	3.59	5.85
23	21.09	3.66	5.76
24	20.83	3.58	5.82
25	21.10	3.63	5.81
26	21.37	3.62	5.90
27	22.21	3.99	5.57
28	20.86	3.61	5.78
29	20.91	3.63	5.76
30	20.97	3.60	5.83
31	20.87	3.57	5.85

7. Selected Procedures

Procedure `cluster_id` – FSM version

```
void
Map::clusterId(int tempRows)
{
   int *C, *N;         // pixel pointers: current, north
   int i, T1, T2;      // loop index and temporary variables
   int previous;       // temp variables
   int state = 0;

    N = firstRow;
    C = secondRow;     // set to first of second row in matrix

    for (i = 0; i < tempRows; i++) {
        while (*C != OB) {
            switch (state) {
                case 0:
                    if (*C == -1) {
                        if (*N == 0) {
                            *C = labelNum++;
                            state = 1;
                            }
                        else {
                            if (csize[*N] < 0) {
                                T1 = csize[*N];
                                while (T1 < 0) {    // chase negative
                                    T2 = -T1;
                                    T1 = csize[-T1];
                                    }
                                *C = T2;
                                while (*N != *C) {  // path compression
                                    T1 = -csize[*N];
                                    csize[*N] = -(*C);
                                    *N = T1;
                                    }
                                }
                            else         // *N > 0
                                *C = *N;
                            state = 2;
                            }
                        previous = *C;
                        csize[*C]++;
                        }
                    break;

                case 1:
                    if (*C == 0)
                        state = 0;
                    else  {
                        if (*N == 0)
                            state = 1;
                        *C = previous;
                        csize[*C]++;
```

```
                    }
                break;

            case 2:
                if (*C == 0)
                    state = 0;
                else if (*N != 0) {
                        state = 2;
                        *C = previous;
                        csize[*C]++;
                           // merge N into C operation
                        if (*N != *C && -csize[*N] != *C)  {
                            if (csize[*N] >= 0) {
                                csize[*C] += csize[*N];
                                csize[*N] = -(*C);
                                }
                            else {
                                T1 = -csize[*N];
                                csize[*N] = -(*C);
                                while (T1 > 0) {      // chase negative
                                    T2 = T1;
                                    T1 = -csize[T1];
                                    csize[T2] = -(*C);
                                    }
                                if (T2 == *C)         // no merge necessary
                                    csize[T2] = -T1;  // restore csize
                                else
                                       csize[*C] -= T1;
                                }
                            }
                        }
                else {
                    *C = previous;
                    state = 1;
                    csize[*C]++;
                    }
                break;
            }
        C++;
        N++;
        }
    C +=2;
    N +=2;
    state = 0;  // end of row return to state 0
    }
} // end of cluster_id
```

Procedure **reLabel**

```
void
Map::reLabel(int tempRows)
{
    int *C;                 // pointer to current pixel
    int i, j;               // loop index variables
    int T1, T2;             // temp holding variables
    C = matrix + cols + 3;  // init tmp to first important value in matrix

    for (i = 0; i < tempRows; i++) {
        for (j = 0; j < cols; j++) {
            if ((*C > 0) && (csize[*C] < 0)) {
                T1 = csize[*C];
                while (T1 < 0) {       // chase negative
                  T2 = -T1;
                  T1 = csize[T2];
                  }
                while (*C != T2) {      // compress search path
                  T1 = -csize[*C];
                  csize[*C] = -(T2);
                  *C = T1;
                  }
                }
            C++;        // move C pointer
            }           // inner while loop
        C+= 2;
        }
}
```