DATA-PARALLEL IMPLEMENTATIONS OF MAP ANALYSIS AND ANIMAL MOVEMENT FOR LANDSCAPE ECOLOGY MODELS

Ethel Jane Comiskey

Computer Science Department

CS-93-207 August 1993

Data-Parallel Implementations of Map Analysis and Animal Movement for Landscape Ecology Models

A Thesis
Presented for the
Master of Science Degree
The University of Tennessee, Knoxville

Ethel J. Comiskey
August 1993

Acknowledgments

I thank my thesis advisor, Dr. Michael Berry, for his guidance, support and encouragement throughout this project. I also thank Dr. Monica Turner and Dr. David Straight who served on my thesis committee. I am very grateful to Dr. Turner and to Dr. Yegang Wu for the opportunity to work with their Northern Yellowstone Park ungulate model. I thank Dr. Bob Gardner and Karen Minser for the use of their serial mean squared radius program, a modified version of which was used for timing comparisons with my parallel algorithms. I also thank the Joint Institute for Computational Science (JICS), which supports the MasPar MP-2 used for this study, and MasPar representative Sally Chase for her advice and help in using the MasPar MP-2.

Abstract

In this thesis, improved sequential and data-parallel implementations of landscape ecology model components are presented. Parallelization efforts on the (SIMD) MasPar MP-2 focus on three model components: cluster identification, mean squared radius computation (cluster geometry), and animal movement.

The NOrthern YELlowstone Park ungulate model (NOYELP), developed by Drs. Monica Turner and Yegang Wu of the Environmental Sciences Division, Oak Ridge National Laboratory, serves as the example landscape ecology model for the model components studied. Modifications made to the original Fortran-77 NOYELP program as part of this thesis project resulted in a revised serial version which executes 11 times faster than the original (CPU time on a Sun SPARCstation 2).

Parallel implementations were tested and compared to functionally comparable serial algorithms using both random maps and maps extracted from runs of the NOYELP model. Speed improvements of MasPar MP-2 parallel kernels over serial implementations on Sun SPARCstations on the order of 9 and 150 for cluster identification and mean squared radius computation, respectively, were measured on 512×512 random maps with a resource probability of 0.85. Speed improvements generally increased with map size and density. For landscape maps tested, speed improvements were somewhat lower, due largely to the inclusion of map pixels outside the study area (54% of total map pixels) in the data maps analyzed.

Results of this study indicate that parallel adaptation of kernels for cluster identification and geometry is straightforward, but that effective parallelization of animal movements in the NOYELP model and similar individual-based models will involve reconceptualizing the movement rule. Issues involved in the parallelization of landscape ecology models are discussed and suggestions are made for future work in this area.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The impracticality of large scale experimental perturbations of natural systems has made computer modeling an important research tool in landscape ecology ([TWWR+93]). Computer simulations are becoming increasingly important in assessing the degree of habitat fragmentation (clustering phenomena) and its ecological implications in many different contexts and at varying spatial and temporal scales. Unfortunately, most landscape ecology models rely on sequential programming, which imposes practical limitations on the size and density of maps which can be analyzed. Parallel computing can expand the capability of these models to simulate spatial and temporal pattern in large ecological systems ([Haef92]).

Individual-based ecological models represent population dynamics by simulating the behavior and interaction of individuals or small groups of individuals responding as discrete units to pattern changes in the environment. Individual movement is simulated in two- or three-dimensions depending on the particular ecological context ([Lomn92]). These models generally have greater computational requirements than population models and can potentially benefit greatly from parallel programming ([Haef92]).

The NOrthern YELlowstone Park ungulate model (NOYELP), the example landscape ecology model used in this thesis, was developed by Drs. Monica Turner and Yegang Wu of the Environmental Sciences Division, Oak Ridge National Laboratory. NOYELP is an individual-based stochastic model which simulates ungulate population dynamics by representing the movement and foraging behavior of small groups of elk and bison in response to changes in the environment that impact the availability of forage and the distance a group can travel in a single day. Computing time for the original serial NOYELP model increases with the number of animals included in the model, as well as with the level of available resources. Parallel computing can be used to increase the number of individuals and the range of environmental conditions considered in individual-based models like NOYELP.

1.2 Objectives

An important objective of this research effort is to produce scalable map analysis algorithms for the identification and characterization of clusters for large, complex maps on massively-parallel SIMD computers. In landscape ecology models, cluster analysis kernels are often called at the end of each model time step to record and evaluate pattern changes, making them key components of many models. For example, the NOYELP model performs cluster analysis twice per model day over 180 days and over 5 replications, for a total of 1800 calls per execution cycle. Cluster radius, a measure of the compactness or density of clusters, has many potential uses in landscape ecology. However, determining radius measures can be computationally intensive, resulting in their exclusion from many applications because of constraints associated with computing time. An efficient parallel kernel for radius computation could make the use of cluster radius (and other measures derived from radius) feasible in landscape ecology modeling.

A second thesis objective is to investigate issues associated with parallelization of the animal movement component of landscape ecology models. Animal movement is typically implemented in serial programs as nested loops of activity repeated over each time-step of a model cycle, a situation for which parallel processing appears well-suited. However, many subjective decisions are made in formulating the rules governing the search and movement of animals in serial models. Some of these rules are based on the serial paradigm and may be unsuited to the constraints of parallel computing. Reconceptualization of movement rules may be necessary for parallel implementations of individual-based models.

1.3 Thesis Overview

Improved sequential and data-parallel implementations of landscape ecology model components are presented in this thesis. Parallel kernels are implemented on the MasPar MP-2, a single instruction, multiple data (SIMD) massively parallel machine. Three model components are examined: cluster identification, mean squared radius calculation (cluster geometry), and animal movement. The first two components are not model-specific and could serve as kernels or modules in other landscape ecology models. The animal movement component is specific to the NOYELP model. Parallel implementations of these landscape model components are tested on random maps and on landscape maps extracted from runs of the NOYELP model. Performance of parallel kernels is compared to that of optimized serial programs.

Chapter 2 of this thesis briefly describes the MasPar MP-2 system, the MasPar

Programming Language (MPL) and the two data-mapping strategies (i.e., hierarchical and cut-and-stack) supported by MasPar. Chapter 3 provides background information about the NOYELP model. Procedures and concepts common to the development and testing of all kernels are presented in Chapter 4. Chapters 5 through 7 address cluster identification, cluster geometry, and animal movement, respectively. Both serial and data-parallel algorithms, along with performance comparisons, are discussed in these chapters. Chapter 8 states conclusions drawn from parallelization efforts and suggests future work in this area. Supplementary information is provided in Appendices A–D.

Chapter 2

Data-Parallel Programming on the MasPar MP-2

The MasPar MP-2 is a massively data-parallel distributed memory processing system consisting of a front end machine and a Data Parallel Unit (DPU). The front end of this single instruction, multiple data (SIMD) system is a DECstation 5000 model 200 workstation with an ULTRIX operating system, windowing capabilities, and standard I/O devices. The DPU, which handles all parallel processing, consists of the Array Control Unit (ACU), the processor (PE) array, an 8-way X-net communication mesh and a global router. Figure 2.1 is a schematic diagram of the MasPar MP-2 system.

The ACU has 24 32-bit registers for user-declared register variables, 128 KBytes of data memory, and 1 MByte of physical instruction memory (RAM), expandable to 4 GBytes of virtual instruction memory. The ACU performs operations on *singular* (shared) variables which are visible to all processors, and controls the PE array, sending data and instructions to each PE simultaneously via the dedicated ACU-PE bus.

The MasPar MP-2 used for this study[1] has 4096 processors arranged in a 64×64 grid. Each PE is a RISC-based processor with 32 32-bit registers available for user-declared variables and 64 KBytes of private (unshared) memory. Each PE has a 16-bit datapath connecting local memory to PE registers. During program execution, all PEs receive the same program instruction from the ACU. All PEs which are active (enabled) at that point in the program execute the same operation simultaneously on their private data. PEs are connected by an 8-way X-net communications mesh and by a global router. In order for one PE to access the private data of another PE, special communication constructs must be used.

[1]supported by the Joint Institute for Computational Science (JICS) at the University of Tennessee, Knoxville.

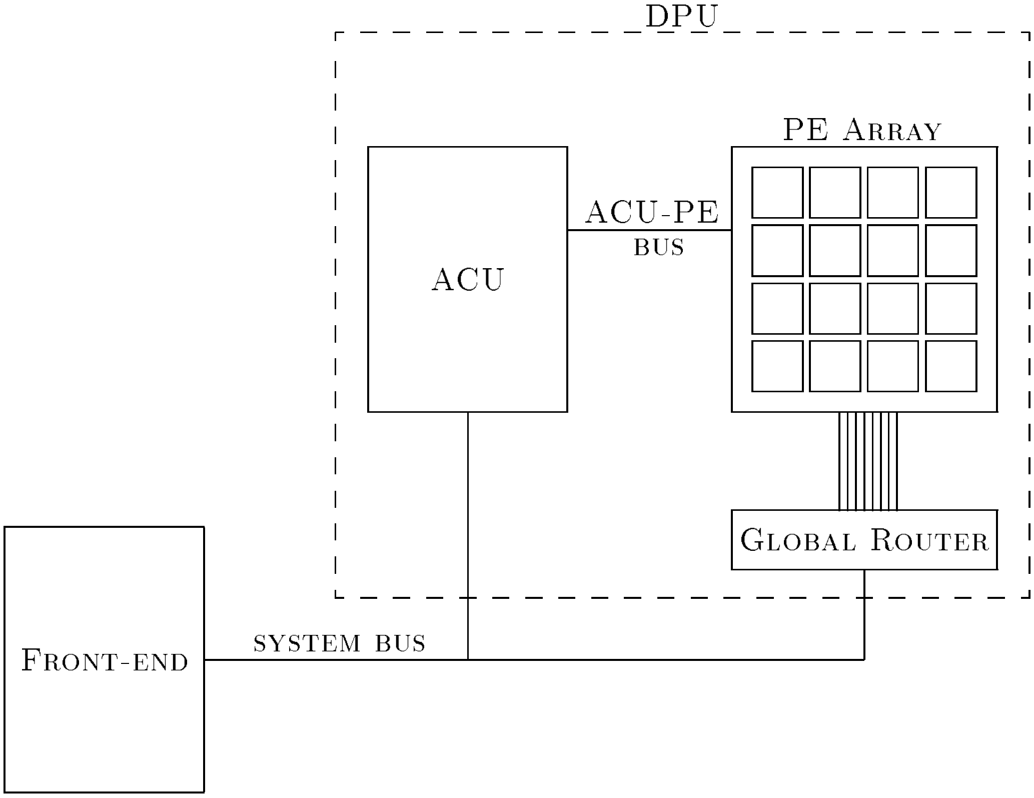


Figure 2.1: Diagram of the MasPar MP-2 system.

2.1 MasPar Programming Language

All kernels developed for this thesis were written in the MasPar Programming Language (MPL), MasPar's ANSI-C compatible language for programming the DPU. MPL is the most efficient and flexible language supported by MasPar. It is also MasPar's lowest level language, allowing more user control over communication and data mapping to processors than can be obtained with MasPar Fortran (MPF). MPL extensions to ANSI C include the capability to allocate *plural* variables across PEs and the ability to perform operations on these variables. MPL adds the keyword `plural` to specify that the associated variable is parallel. An example is provided below.

```
int i;          /* allocates 4 bytes in the ACU's memory */
plural int j;   /* allocates 4 bytes in each PE's memory */
```

Three communication constructs are provided for sending and receiving values between sets of PEs: `iproc`, `xnet`, and `router`. The `iproc` construct allows access to a plural variable on a single PE. The `xnet` construct is used to access processors which are a uniform distance away from active processors in one of eight directions: north, south, east, west, northeast, northwest, southeast, or southwest, requiring both a distance and a direction specifier. Automatic toroidal wraparound is employed with `xnet` to allow circular shifting of data values. East-west borders and north-south borders are connected for shift purposes. For example, the bottom row of PEs is 1 xnet shift north of the top row of PEs. The `global router` is used for communication between a particular PE and any other member of the PE grid.

2.2 Data Virtualization on the MasPar MP-2

Typical cluster analysis applications involve data sets larger than the size (64×64) of the MasPar MP-2 processor array. In situations such as this, where individual processors must handle more than one data point (i.e., map pixel), data must be mapped onto the processor array in some fashion. MasPar systems provide two general data mapping strategies for allocating multiple data points to individual processors: cut-and-stack and hierarchical. Figures 2.2 and 2.3 show in schematic form MasPar MP-2 cut-and-stack and hierarchical virtualization, respectively.

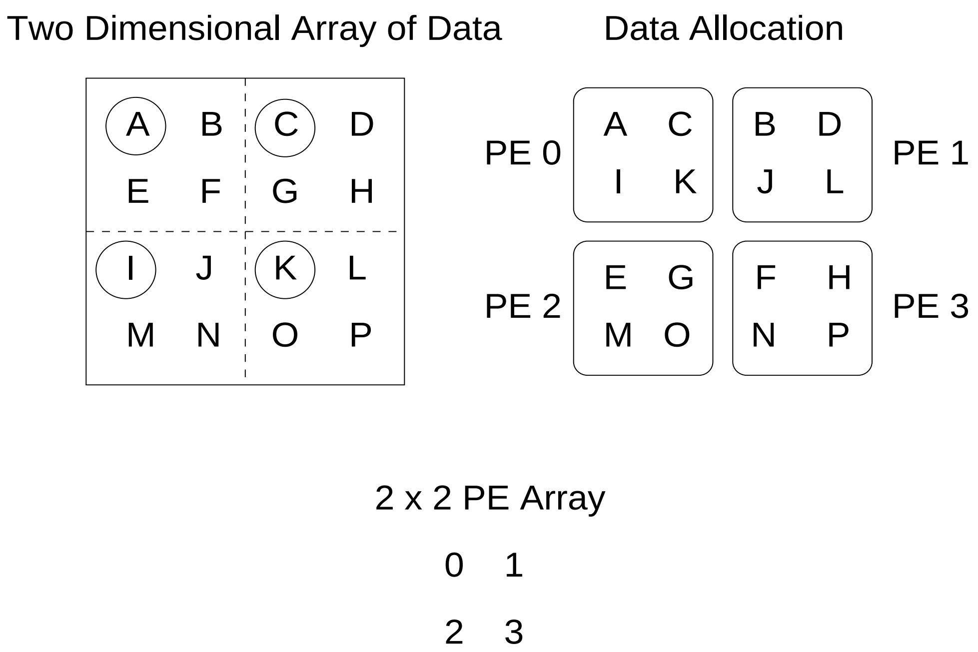


Figure 2.2: Cut-and-stack virtualization on the MasPar MP-2.

With cut-and-stack mapping (Figure 2.2), the data set is divided into a number of segments, called *pages*, equal to the total number of pixels divided by the total number

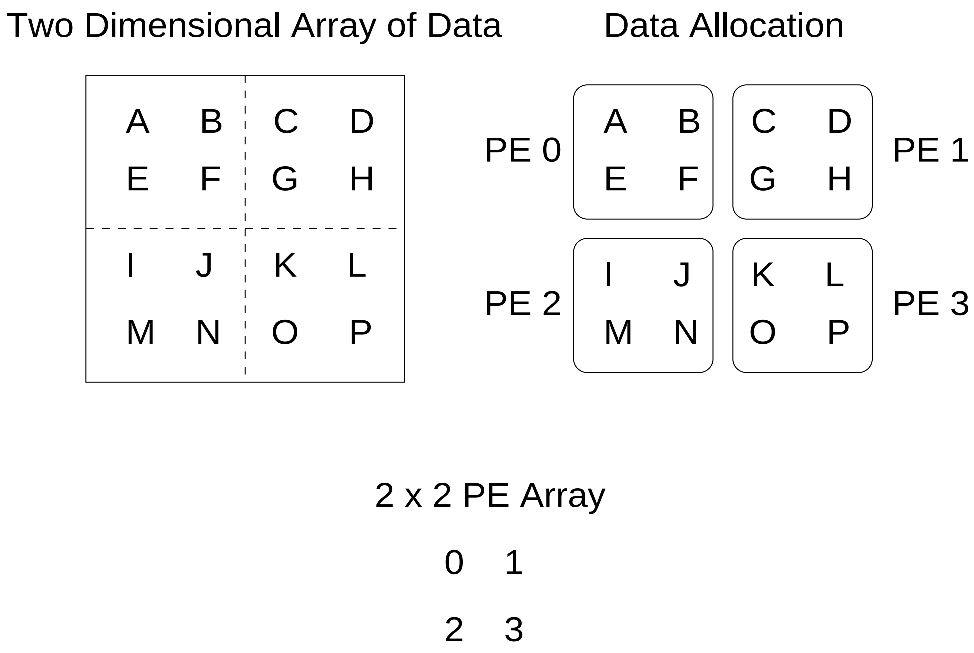


Figure 2.3: Hierarchical virtualization on the MasPar MP-2.

of PEs (i.e., 4096). Each page is the size of the PE array and each PE receives one piece of data from the same relative position in each page. Therefore, if the data pages were stacked, the data assigned to each PE would be an array whose elements form a column running vertically through the stacked pages. In cut-and-stack mapping, logically consecutive data points (i.e., adjacent map pixels) are assigned to physically adjacent processors, accessed by using the MPL communication construct `xnet`. In Figure 2.2, circled data items are assigned to PE 0.

To implement hierarchical data mapping, the data set is divided into as many equally-sized rectangular blocks of adjacent elements as there are PEs, with each PE being allocated one logically contiguous block representing a sub-grid of the original data set. No communication constructs are required for operations within each PE's sub-grid of values. The `xnet` construct is used to communicate between border row and column elements of adjacent sub-grids. As with cut-and-stack mapping, for maps greater than 64×64 processors are assigned more than one pixel, and these pixels are stored as data arrays at each processor.

In choosing between these two strategies, one tries to maximize processor utilization by balancing the workload across processors, while at the same time minimizing communications between processors. Hierarchical mapping is generally more efficient when communication needs are localized in subareas of the data map. Algorithms implementing cut-and-stack mapping are generally simpler to encode, and are often more efficient when there is no advantage to having adjacent data in each PE memory (i.e., when communication requirements are not localized) because work is distributed more evenly across processors.

There exists a family of MasPar `mpi` conversion functions which allow data configurations to be changed within a program. If different data mapping strategies are more efficient for different parts of a program, the programmer can switch between cut-and-stack and hierarchical mapping as needed. See Appendix A for a list of these functions.

Data-parallel cluster analysis kernels employing both hierarchical and cut-and-stack data mapping were developed on the MasPar MP-2 for this study and performance comparisons were made for the two strategies. These results are discussed in Chapters 5, 6, and 7.

Chapter 3

Northern Yellowstone Park Ungulate Model

The programming efforts discussed in this thesis use as an example the spatially explicit, individual-based NOYELP model. The NOYELP model simulates the search, movement, and foraging activities of individuals or small groups of free-ranging elk (*Cervus elaphus*) and bison (*Bison bison*) on the part of their winter range which lies in northern Yellowstone National Park (NYNP). The model was developed to explore the effects of fire scale and pattern on winter foraging and survival of ungulate populations on the heterogeneous, multi-habitat NYNP landscape. The information presented in this chapter is a summary of [TWWR+93], which provides a detailed description of the model and its application to the assessment of impacts of fire on ungulate survival in the context of related studies. A summary list of NOYELP subroutines and formulas used for computation of forage biomass and animal energetics are included in Appendix B.

NOYELP simulations are conducted for each of 180 days during the (approximate) period of November 1 through April 30. Within a day, an animal group makes one to several moves in its search and foraging activities. Available forage biomass varies as a function of foraging activity and snow cover. Ungulate body weight is decreased whenever daily forage intake does not meet energy requirements. Starvation during winter, the main factor influencing ungulate mortality in the study area, occurs when calculated body weight falls below survival thresholds. The model does not project ungulate reproduction or plant growth. For each year simulated, new data (e.g., weather conditions during the 180-day period, number of ungulates present at the beginning of winter, and amount of forage in kg/hectare present in each habitat category at the beginning of winter) are input to the model. Because NOYELP is a stochastic model, five replications of the 180-day simulation period are run with each set of input conditions, and results are summarized statistically over the set of replicates.

3.1 Description of the Study Area

Yellowstone National Park (YNP) was established in 1872 as the nation's first National Park. It covers 9000 km^2 (900,000 hectares) of the landscape in the northwest corner of Wyoming and immediately adjacent parts of Montana and Idaho. The NYNP study area encompasses 77,020 ha in the north central part of YNP. Approximately 83% of the elk winter range is included within the NYNP study area. Ecological dynamics on the winter range largely control ungulate survival and population sizes in YNP.

YNP is characterized by long, cold winters and short, cool summers. The climate is somewhat warmer and drier in the study area compared to the rest of the Park. The northwestern-most part of the study area lies in a precipitation shadow. Snowfall in this area is typically lower than in the winter range as a whole. While elevations in YNP range from 1500 m to more than 3000 m, those characteristic of the NYNP study area are in the lower end of the range. The vegetation of the study area consists primarily of lower-elevation grassland or sagebrush steppe interspersed with aspen and conifer woodlands.

3.2 Model Description

In the NOYELP model, the NYNP landscape is represented as a gridded irregular polygon with a spatial resolution of 1 hectare. The irregular shape of the study area requires a 285 by 584 grid (166,440 grid cells) to span the 77,020 1-hectare grid cells for serial implementations.

Spatial heterogeneity across the NYNP landscape is represented by a series of data maps. Some of the environmental data (e.g., elevation, slope, aspect) are constants, while others (e.g., baseline snow depth and forage biomass) may vary from simulation to simulation. Many of the abiotic data were obtained from the YNP geographic information system (GIS). Elevation is used primarily to initialize bison locations at the beginning of the simulation. Slope, aspect and baseline snow depth are used to estimate effective snow depth.

Initial quantities of pre-winter forage assigned to each habitat grid were derived from data collected during late summer and early fall of 1990. Unlike other habitat variables, forage biomass values are influenced by animal foraging activities and effective snow depth. Ungulate groups search the study area for forage according to a detailed serial movement rule. When snow depth exceeds the brisket height of the particulate ungulate category, foraging cannot occur. Daily forage intake is balanced against daily energy expenditure in estimating weight loss by ungulate individuals. When weight drops below a survival threshold, mortality occurs.

The NOYELP model was originally calibrated by adjusting the values of two parameters, maintenance energy (`enmb`) and the upper threshold of snow equivalent at which foraging is precluded (`swhi`), for which field data were not available ([TWWR+93]).

3.2.1 Habitat types

To simulate ungulate foraging and survival, the NYNP landscape is discretized into six habitat types (Figure 3.1). Four of these six habitat types are grasslands, differentiated primarily on the basis of moisture availability, species composition and biomass production. At elevations characteristic of NYNP, the sagebrush-grassland habitat types form a patchwork mosaic with the two woody habitat types, aspen stands and coniferous (canopy) forests (dominated by pine and fir). In general, ungulates appear to respond to forage quantity rather than quality or subtle community differences, which simplifies the modeling process. At the start of the model year, forage is distributed within each habitat type by assigning to each grid cell in the habitat type a forage biomass value drawn randomly from the 95% confidence interval (i.e., $\pm$ 2 standard errors) of a normal distribution around the mean for the particular vegetation class, as determined by field sampling.

3.2.2 Ungulate Categories and Distribution

Six ungulate categories were defined for simulation purposes. Bison groups consist of 9 cows, 9 calves, or 2 bulls, while elk cow, calf and bull groups each include four individuals. Since calf groups follow the foraging pattern of the cow groups to which they are assigned, the model effectively simulates groups of 8 and 18 combined cow/calf groups for elk and bison, respectively. The model places no constraints on the number of groups that may occupy a grid cell at one time. All animals within the same ungulate group are assigned the same initial body weight.

Elk are initially distributed randomly within grid cells containing forage (i.e., resource sites) across the winter range. Bison groups are initially distributed randomly in grassland habitats at elevations $\leq 2100m$, resulting in approximately 90% being assigned to the eastern portion of the study area.

3.2.3 Snow Simulation

Snow conditions are extremely important in determining the winter dynamics and survival of ungulate populations in the NYNP. Both foraging and movement can be affected by snow. Snow depth and snow density (%) are used to determine energetic costs of travel and maximum daily moving distances. Snow water equivalent (`swe`), the product of snow depth and density, influences daily forage intake. When snow depth exceeds brisket height, ungulates generally cannot forage. Even shallow, densely packed snow may limit foraging. There is an upper limit of snow/water equivalent (`swhi`) above which no foraging can take place. This limit is category-specific, depending largely on the size of the animal (i.e., brisket height). For example, bulls can travel and graze in deeper, denser snow than calves.

To simulate snow conditions, the northern range is first subdivided into two regions (the snow-shadow area and the rest of the study area), based on amount of winter precipitation, and baseline snow depths and densities are then projected within each area, assuming no slope or aspect effects. These baseline projections are subsequently modified according to the slope and aspect of each grid cell. Snow conditions are updated at 3-day intervals.

3.2.4 Foraging

Daily ungulate forage intake on a grid cell is a function of maximum daily intake. This intake is a product of two constants, initial body weight (`bw`) and maximum daily foraging rate, (`feed`), and one of two negative feedback terms representing the amount of available forage at the site (`fbbio`) and the depth and density of snow (`fbswe`). Each feedback term is a number between 0 and 1. Whichever feedback term is smaller has the greater impact on foraging and is allowed to operate.

The hyperbolic forage availability feedback term (`fbbio`) reflects a direct relationship between the amount of available forage and the instantaneous rate of feeding. As ungulates graze, the feedback term decreases, reflecting a decrease in the amount of available forage on the grid cell. Because no regrowth of vegetation occurs during the winter (dormant) season, and no other sources of forage attrition are considered in the model, biomass can only remain the same or decrease. The feedback function utilizes the concept of a refugium value of biomass not available to ungulates. When forage biomass falls to the refugium value (13% of Fall biomass), the value of the feedback term is set to zero. When the refugium value is greater than 0, the feedback term will be less than 1.

The snow water equivalent (`swe`) feedback term reflects the effect of both snow depth and snow density on the ability of ungulates to obtain forage. Two thresholds (the value at which foraging is set to zero and the value at which limitation of foraging begins) are used in defining the snow feedback term. At `swe` values greater than those which limit foraging, an animal can forage at its maximum rate. Between the two thresholds of `swe`, a linear change in the value of the feedback term is assumed.

3.2.5 Search and Movement Rules

The NOYELP model utilizes a simple algorithm to simulate an ungulate's search and movement strategy. If an ungulate group is located on a grid cell containing available forage at the start of the day, the animal grazes. If forage intake on that cell is less than the daily maximum, the animal searches for another grazing site. Because the forage feedback term has an upper limit at 0.87 (with the refugium level set to 13%), the ungulate group cannot attain its maximum daily intake of forage from one grid cell, and must move at least once per day. An ungulate group is prohibited from remaining at the same grid cell, revisiting a grid cell during the course of one day's movement, or moving to a grid cell outside the boundary of the study area. Searching procedures are described in detail in Chapter 7.

During fall and early winter, when forage is generally available, ungulate groups will typically move only once per day. Later in the winter, when forage becomes less available due to foraging and/or the presence of snow cover, an ungulate group may move several times per day in its search for food. Maximum movement distance (and, therefore, the number of moves an ungulate group can make in one day) decreases as snow conditions become more extreme because of increases in energy costs associated with travel in snow.

3.2.6 Energetics

Daily energy balance is the difference between daily energy gain, `engain`, and daily energy expenditure, `encost`. `Engain` is the product of total intake of forage in kg (`fd`) and habitat type specific forage energy content in kcal/kg (`enpk`). `Encost` is the sum of maintenance energy cost (`enme`) and travel energy cost (`enmov`). Maintenance energy cost, represented as a power function of current body weight, includes the energy costs associated with all the animal's daily activities which occur within the grid cells. For initial model parameterization, estimates of maintenance energy cost obtained from the literature were used, but these estimates were subsequently adjusted during model calibration. Travel energy cost is computed by first calculating the (per unit distance) energy cost of travel in the absence of snow (a function of body weight), and then modifying this value to account for the relatively higher travel costs associated with travel in snow (a function of snow depth and snow density). These costs increase exponentially as a function of relative sinking depth, and may limit maximum daily distance traveled and (consequently) the number of cells a group can search during the day.

Whenever forage intake is insufficient to meet the animal's energy expenditures, ungulate body weight is adjusted downward. No weight gain is permitted. Death by starvation is assumed to occur when ungulates lose both 70% of their fat and 30% of their non-fat body weight. Since no predators are included in the model, death by starvation is the only significant source of population attrition on the winter range.

3.3 Example Data Set

For the example data set used in simulations for this study, 19,972 animals in 5,015 groups were input to the model. Greater than 96% of the total number of ungulates were elk. Table 3.1 presents a listing of ungulate cow, calf and bull groups, with

Table 3.1: Sample composition of animal groups in a NOYELP model simulation.

Category	Fraction of Population	Size of Groups	Number of Individuals	Number of Groups
Elk:				
Cows	0.65	4	12524	3132
Calves	0.16	4	3084	771
Bulls	0.19	4	3660	915
Total	1.00		19268	4819
Bison:				
Cows	0.38	9	270	30
Calves	0.18	9	126	14
Bulls	0.44	2	308	154
Total	1.00		704	196
All ungulates			19972	5015

a count of individuals of each species belonging to each of the categories. Since movement of calf groups follows that of the cow groups to which they are assigned, the model effectively simulates the movement of 4,230 groups.

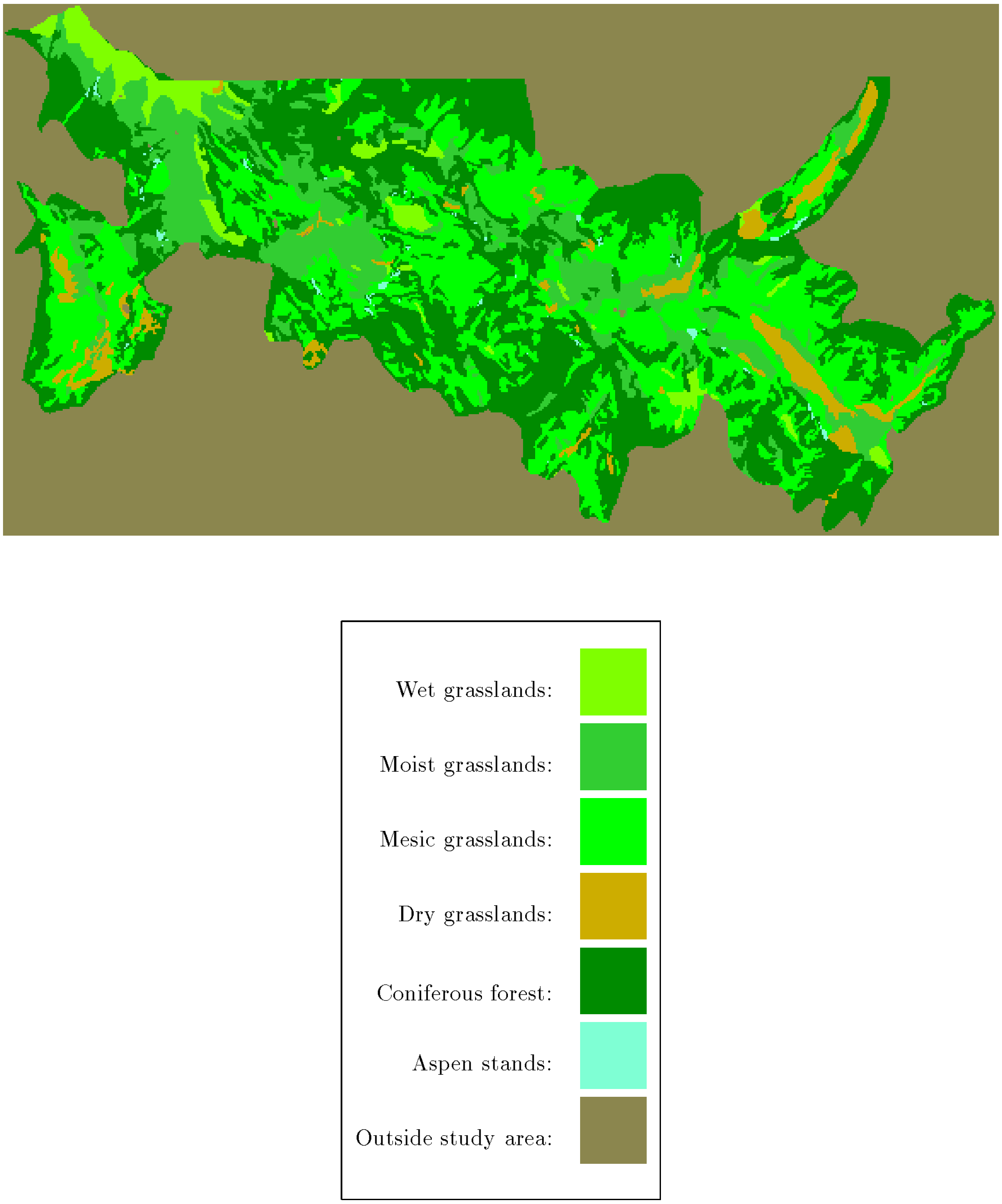


Figure 3.1: Map of NOYELP study area showing habitat types.

Chapter 4

Preliminaries

In developing kernels for diverse applications requiring cluster identification and geometry, the programmer must make the kernels adaptable to the needs of the application. Clusters might represent resources, animals, landscape patterns, or dispersal patterns of pollutants ([TWWR+93]). Map size and density may be static or may change during program execution.

To simulate cluster analysis on landscape maps, random maps are generated having a proportion, p, of 1's. The proportion of 1's (or non-zero elements) in a map is called the p *value* of that map ([StAh91]). These random maps are then used to facilitate algorithm development and to test parallel implementations for accuracy. Using random maps, performance of an algorithm can be predicted on prospective real world maps of various sizes and densities from actual landscape ecology models. For testing purposes, $m \times m$ random maps of density p are generated, where $m =$ 64, 128, 256, 512, 768, 1024, and $p = 0.10$, 0.30, 0.62, 0.85, and 1.00 for each value of m.

In landscape ecology models, the non-zero pixel elements generally represent an assigned level or range of some specific habitat parameter (e.g., moisture less than 10%, biomass greater than 100 kg/hectare, or a flammability index of 7), animal group density, or some aggregate of parameters representing habitat suitability on the unit area of landscape.

For this study, 1 hectare landscape units with available resource (i.e., forage biomass) above a pre-assigned threshold level is represented by setting the grid elements of the random maps to 1 with a probability of p. For these random maps, a setting of 0 (with probability $1 - p$) represents habitat with resource levels below the threshold. Thus, pm^2 is the number of pixels of suitable resource habitat (or 1's) within the map. For irregularly-shaped real world study areas, such as NYNP, the rectangular or square data grid will include pixels outside the study area, thereby decreasing the p value of the map as a whole.

4.1 Map Density

Table 4.1 summarizes cluster size distribution, size of the largest cluster, and total number of clusters for the five p values and six map sizes used in this study. All clusters for maps with $p = 0.10$ and $p = 0.30$ have fewer than 100 members (or pixels), while maps with $p \geq 0.62$ have a large dominating cluster, along with smaller clusters. This change in maximum cluster size is explained by percolation theory ([StAh91]). According to this theory, maps with p values greater than a threshold of 0.5928 are characterized by a large dominating cluster that *percolates* across the map from boundary to boundary. A random map with $p = 0.10$ is a sparse map with small, isolated clusters; $p = 0.30$ yields a map with many fragmented clusters; a map with $p > 0.59$ is dominated by one large cluster.

Figure 4.1 illustrates the differences in cluster numbers and sizes associated with p values of 0.30 and 0.62 for 64×64 maps.

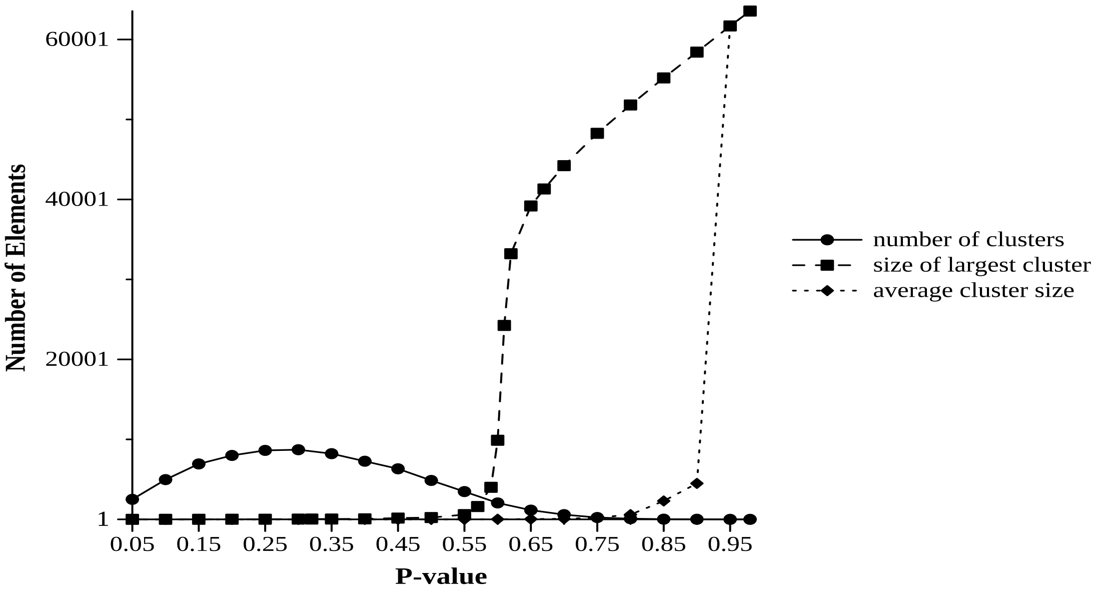
(a) (b)

Figure 4.1: Sample 64×64 random maps with (a) $p = 0.30$ and (b) $p = 0.62$.

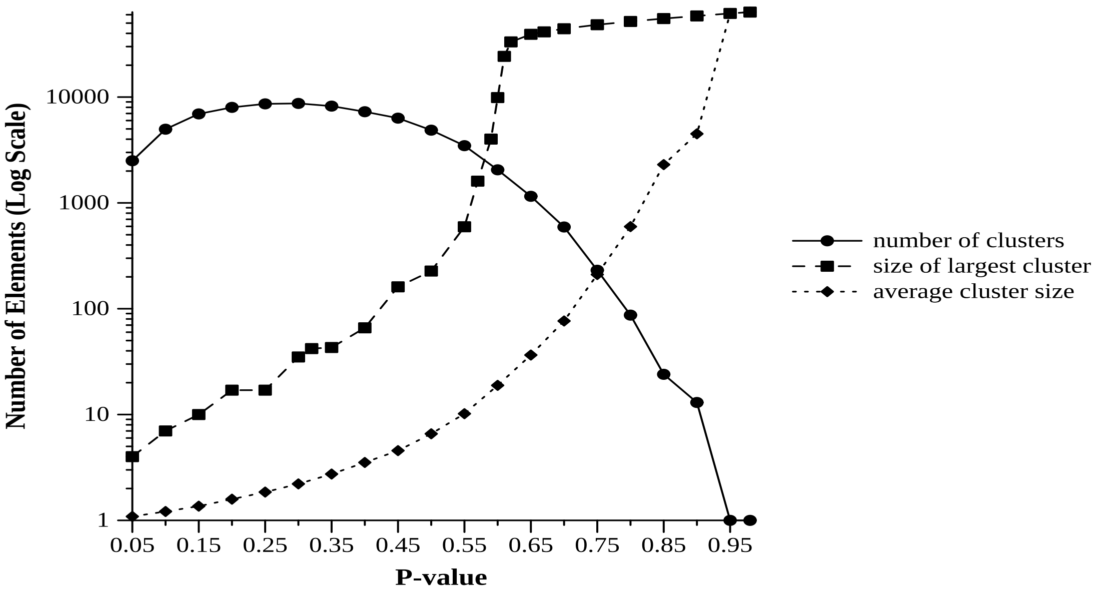
As the p value of a map increases, the amount of the work involved in cluster identification and geometry computations changes. The relationships of number of clusters, maximum cluster size, and average cluster size to p value for 256×256 maps are illustrated in greater detail in Figure 4.2. The number of clusters gradually increases to a peak around $p = 0.33$, after which it gradually decreases to 1. The average cluster size remains small until p values exceed 0.80 and the number of clusters drops to about 100. Maximum cluster size stays relatively small until the percolation threshold of $p = 0.5928$ is reached, after which the size of the largest cluster increases dramatically and remains large. For larger p values, the average cluster size

Table 4.1: Distribution of cluster sizes for randomly generated maps of six sizes and five p values.

		Size of	Total	Cluster Size						
	Map	Largest	No. of		101-	501-	1001-	10001-	100001-	
p	Size	Cluster	Clusters	<100	500	1000	10000	100000	500000	>500000
0.10	64	8	325	325						
	128	6	1305	1305						
	256	7	5227	5227						
	512	12	20917	20917						
	768	8	47281	47281						
	1024	15	84140	84140						
0.30	64	25	534	534						
	128	29	2157	2157						
	256	33	8484	8484						
	512	42	33891	33891						
	768	44	75941	75941						
	1024	55	135122	135122						
0.62	64	1981	110	109			1			
	128	6609	382	376	4	1	1			
	256	34363	1400	1393	5	1		1		
	512	141190	5503	5490	10	1	1		1	
	768	323676	11989	11965	22	1			1	
	1024	577501	21141	21099	37	4				1
0.85	64	3455	4	3			1			
	128	13862	9	8				1		
	256	55523	30	29				1		
	512	222417	129	128					1	
	768	500409	259	258						1
	1024	890080	510	509						1
1.00	64	4096	1				1			
	128	16384	1					1		
	256	65536	1					1		
	512	262144	1						1	
	768	589824	1							1
	1024	1048576	1							1



(a)



(b)

Figure 4.2: Comparison of cluster characteristics across p values for 256×256 maps (a) standard scale and (b) log scale.

approaches the size of the largest cluster as the number of clusters approaches one.

Execution times for cluster identification and geometry programs can be related to changes in cluster characteristics as a function of map density. Different cluster analysis algorithms developed during the course of this study generally reach performance peaks at characteristic p values. Some algorithms performed well on sparse maps (p values below about 0.15) or handled large numbers of small clusters well (p values near 0.30), but did not process dense maps containing a small number of large clusters efficiently. Algorithms which perform well on the large dominating clusters found in maps with p values above 0.59 often were not well adapted to handle the numerous smaller clusters found in sparser maps. Testing with maps of widely varying densities is necessary to identify the combination of algorithms which provide optimal performance over the entire range of densities found in real landscape maps.

The relationship between cluster characteristics and p values is somewhat different for non-random landscape maps. While the size of the largest cluster (and hence computational complexity) is predictably tied to the p value in random maps, this is not the case with landscape maps. For example, calculation of the mean squared radius of a cluster of size n requires computing $\mathcal{O}(n^2)$ squared row and column differences. The random map of size 64×64 (4096 pixels) with $p = 0.30$ (1229 resource pixels) generated for this study contains 534 clusters, all with fewer than 25 pixels and an average cluster size of 2.3. A total of 6103 row and column comparisons are required for all cluster geometry computations. By comparison, a 64×64 (non-random) landscape map with a p value of 0.30 could have 1229 clusters of 1 pixel each, requiring no row/column comparisons for mean squared radius calculation or, at the other extreme, could have 1 cluster with 1229 pixels, requiring $1229 \times 1229 = 151044$ comparisons. Because the relationship of p value and cluster size is not predictable for landscape maps, the size of the largest cluster is a more accurate indicator of performance than p value for these maps.

While results from random maps might not be directly comparable to those from landscape maps of the same size and density, the trends in algorithm performance over a set of random maps of varying sizes and densities can serve as a general guide to performance on landscape maps with known cluster characteristics. The similarity between performance results for random and landscape maps increases as the p value increases, since the range of possible cluster configurations decreases as maps become more dense.

Some modeling applications require analysis of clusters in a static map of known density while in other applications, such as NOYELP, changes in a given landscape feature (i.e., available biomass) over time must be evaluated by identifying and characterizing clusters at each time step. A typical dynamic application might start with a dense map (high p value) and analyze progressively sparser maps as resources are depleted; other applications will involve a map that is initially sparse, with the p value increasing as complexity is added. In these dynamic applications, a small improvement in cluster analysis time could save hours in total computing time.

4.2 Map Size

The relationship between random maps and landscape maps is complicated by the irregular shape of real landscape unit, and the resulting need to include pixels in real maps which are outside the study area to attain a regular (i.e., rectangular) grid for spatial analysis. For example, the 77,020 hectare NYNP study area is an irregularly-shaped polygon. Because of its irregular shape, the study area must be represented in the serial NOYELP simulation model as a 285×584 grid. If the study-area pixels were contained in a square, it could be represented as a 278×278 grid, with a spatial resolution of 1 hectare. Pixels outside the NOYELP study area constitute 54% of the model grid and are assigned the value -1 to distinguish them from habitat pixels.

When NOYELP input files are adapted to the 4,096-processor MasPar MP-2 implementations, the model grid must be further extended to 320×640, so that all rows and columns are multiples of 64, the row and column dimensions of the MasPar PE grid. This increases the proportion of non-study area pixels to 62% of the total map area. For MasPar MP-2 implementations, each processor is assigned 50 pixels, with an average of only 19 of these representing study area pixels. As a result, speed improvements for parallel NOYELP implementations will be smaller than for random maps of the same size. Processors which have been assigned map segments outside the study area will be idle, causing performance degradation. For this reason, the shape of the study area is a much more serious concern for parallel simulations than for serial models.

Algorithm performance for NOYELP maps will also differ from that of random maps of the same size and density because of the non-random distribution of forage biomass, a product of environmental heterogeneity and non-random foraging activities. This clumping of biomass distribution will affect different cluster analysis algorithms in different ways (see discussion in Section 4.1 above).

4.3 Verification

In developing parallel cluster analysis and ungulate movement algorithms, testing with maps of varying size and density is important not only for predicting performance, but also for verification of results. Error detection in parallel programming is not as straightforward as it is with serial programming, especially when data virtualization (via MPL) is involved. Care must be taken to exercise all permutations of inter-layer communication to ensure that no data are lost. Implementations which function correctly for dense maps (i.e., which "turn on" most or all virtualized layers to the active state) may fail when maps are sparse. On the other hand, these dense maps may generate large intermediate sums or products which test the limits of variable sizes and the accuracy of variable casting. During the algorithm development phase of this thesis effort, several errors appeared in only 1 of the 30 size/density test combinations, pointing to the need for testing over a range of combinations.

4.4 Serial Computing Environment

The computing environment used for serial program development and testing for this thesis consists of two architectures: the Sun SPARCstation 2 and the SPARCstation IPX. The SPARCstation 2 has a SunOS 4.1.2 operating system, a clock speed of 40 MHz, and 64 Mbytes of RAM. The Sun SPARCstation IPX is very similar, with a SunOS 4.1.3 operating system, a clock speed of 40 MHz, and 16 Mbytes of RAM. Both are capable of a peak computation rate of 4.2 Mflops (Millions of FLOating-Point operations per second). Performance of these machines on SPEC benchmarks is shown in Table 4.2.

Table 4.2: Performance specifications for architectures used in the sequential computing environment.

Machine	Sun SPARCstation IPX	Sun SPARCstation 2
MIPS	28.5	28.5
SPECmark89	24.4	25.0
SPECint92	21.8	21.8
SPECfp92	21.5	22.8
Mflops	4.2	4.2

The serial timings presented in the results sections of Chapters 5 through 7 were obtained from one of these two architectures, as specified in discussions and in table and figure captions. The Sun SPARCstation 2 was generally used, except for programs which compute mean squared radius. These programs required many hours of computing time, and were run on the Sun SPARCstation IPX because of the availability of dedicated time. Dedicated wall-clock times are used when times for serial implementations are compared to those for parallel implementations. When serial implementations are compared with other serial implementations, CPU times for programs run on the same machine (Sun SPARCstation 2 or Sun SPARCstation IPX, as specified) are used. When computing times are specified as excluding *I/O* time, *I/O* refers to reading data from input files and writing results and times to a standard output device (screen).

Chapter 5

Cluster Identification

Cluster identification is not unique to landscape ecology. It is important in such diverse fields as image processing and lattice field theory in physics ([ApCM92]). In physics, cluster identification is performed on n-dimensional maps and is referred to as *connected component labeling*, with map elements considered as boolean variables set to *on* or *off*. The goal is to have the same unique label on all connected sites and a different label for each disconnected cluster.

In landscape ecology applications, cluster identification typically involves locating and labeling clusters in a 2-D grid, and determining cluster characteristics such as total number of clusters, size of each cluster, size of the largest cluster, and average cluster size. Adjacent pixels are considered to belong to the same cluster if they have the same value (e.g., habitat or resource level), as defined by a particular nearest-neighbor rule. The neighbor rule implemented in the serial and parallel algorithms investigated in this thesis considers pixels containing the same value to belong to the same cluster if they are north, east, west or south (NEWS) neighbors of each other or of some other element in the cluster. Diagonal adjacency is not considered in this rule. In the following discussion, grid cells or pixels having a positive value indicating membership in the map class being analyzed are called resource pixels, consistent with the NOYELP model example utilized in this effort (where forage biomass is the resource variable of interest). Figure 5.1 shows a simple grid with cells belonging to each cluster (according to the neighbor rule) in a common enclosure.

5.1 Serial Algorithms

Serial cluster identification algorithms fall into one of two classes: (1) those which build entire clusters in sequence (i.e., one at a time) in the order in which they are encountered during grid traversal, and (2) those which build clusters incrementally, as members are encountered in grid traversal.

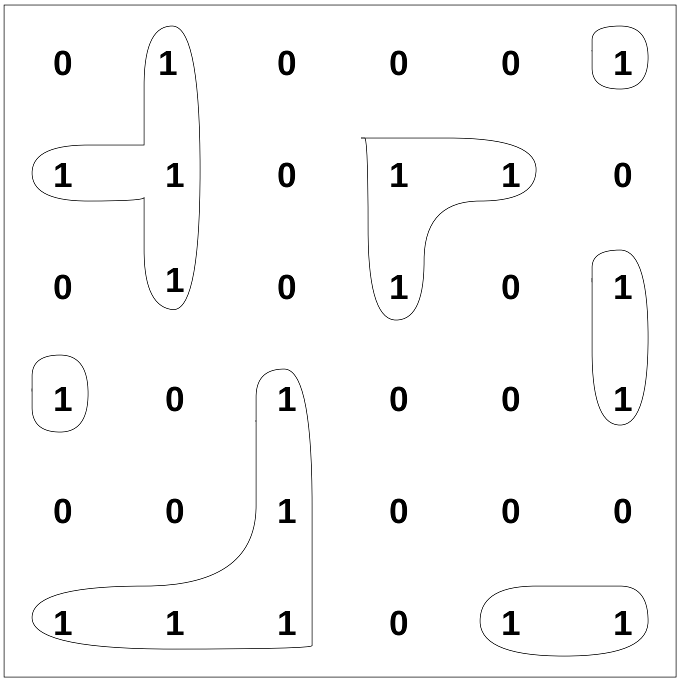


Figure 5.1: Two-dimensional spatial grid showing showing 7 individual clusters.

Recursive and pseudo-recursive cluster identification programs fall into the first class (i.e., they build cluster in sequence). Recursive programs have a cluster-building function which labels one pixel, then calls itself recursively with the location of any nearest neighbors of that pixel. Pseudo-recursive programs simulate recursion by storing pixel locations in arrays, which serve as *stacks* of cluster elements from which clusters are sequentially built. Cluster labeling is accomplished by traversing the grid one element at a time. When a pixel containing a resource value is encountered, it is labeled with a unique cluster number. Then all nearest neighbors of this pixel are examined and added to the growing cluster if they are also resource pixels. This process continues with examination of nearest neighbors of nearest neighbors, until all pixels in a particular cluster are identified and labeled. Traversal of the grid (and building of the next cluster) then continues with the next unlabeled pixel and the process continues until all clusters members are labeled.

Algorithms which fall into the second class embody an alternate approach to cluster identification, incremental cluster building. The grid is traversed from top down and left to right. Pixels are given temporary labels as they are encountered, and labels are updated as clusters take shape concurrently. Examples of this approach are the Hoshen-Kopleman algorithm, discussed in more detail below, and the *local diffusion* or *label propagation* method ([ApCM92]).

In the following subsections, (5.1.1 to 5.1.3), five serial cluster algorithms are discussed. Three of these (the original NOYELP function, a revision of this function, and the Hoshen-Kopleman algorithm) implement incremental cluster identification, while the other two, based on recursion and pseudo-recursion, respectively, process clusters sequentially.

5.1.1 Original NOYELP incremental algorithm

The original cluster identification algorithm used in the serial NOYELP model is a type of *local diffusion* algorithm, wherein repeated local nearest-neighbor comparisons (using the NEWS rule) result in correct labels diffusing throughout the grid. As the resource grid is traversed one element at a time, each element compares its resource value with those of its nearest neighbors. Elements with matching values are included in the same cluster by assigning to all the lowest (numerical) cluster label in the cluster. Cluster labels are updated as cluster membership changes. In the original NOYELP version, grid traversal is repeated four times, with four comparisons made for each pixel on each pass. It was assumed that all clusters had been properly labeled at this point.

As implemented in the original NOYELP model, this cluster identification algorithm had several deficiencies. One involved a minor array index error which would have led to aberrant results, but was easily corrected. The other, which involved the inefficient and incomplete implementation of the basic algorithm, was more significant. Testing with several 10×10 data files representing several known levels of cluster complexity showed that this algorithm correctly identified simple cluster patterns, but complex clusters were incompletely labeled. Testing with larger random maps revealed that errors in cluster identification began to appear as resource pixel density approached the percolation threshold ($p = 0.59$). Large clusters which spread from border to border were incorrectly labeled as several smaller clusters. These results indicated that four traversals of the grid were not sufficient for complete cluster identification when densities were near the percolation threshold.

The original algorithm was modified to identify clusters correctly by adding an activity flag and making repeated passes through the data until no cluster-building activity was detected. As many as 32 4-comparison (NEWS) passes through a 1024×1024 map with $p = 0.62$ were required to accurately label all clusters, resulting in a substantial increase in execution time. Clearly, a more efficient algorithm was needed to deal with the range of map densities which might be encountered in modeling natural environments.

Yegang Wu, author of the original NOYELP program, subsequently developed another incremental cluster-building Fortran-77 algorithm (ORFOR) which correctly identified all clusters and showed significant speed improvements over the original algorithm (modified for accuracy by adding the activity flag, described above) but required large arrays and performed poorly on maps with large clusters.

5.1.2 Recursive and pseudo-recursive algorithms

Cluster identification is most easily conceptualized in recursive terms, but recursion is not available in Fortran-77. Nonrecursive versions of recursive algorithms are often more efficient, even in recursive languages, because they lack the overhead of repeated parameter passing, and because nonrecursive code is more easily optimized by compilers ([HoSa83]).

A two-step approach was taken to develop this more efficient Fortran cluster identification algorithm:

1. A recursive C program (RECRC) was written and tested for accuracy. This program provided the conceptual basis for development of a functionally comparable pseudo-recursive Fortran program.

2. Following a procedure outlined in Horowitz and Sahni ([HoSa83]), the recursive C program was translated into a pseudo-recursive Fortran-77 program (PRFOR). The pseudo-recursive version builds clusters in the same way as the recursive program, but uses only iteration to control program flow. Arrays of row and column numbers (x- and y-coordinates) are used to simulate a stack. Pixel coordinates are pushed onto the stack by adding elements to the arrays; coordinates are popped from the stack by decrementing the count of items, which serves as the maximum array index.

After testing for accuracy and performance on random maps of various sizes and densities, the revised algorithm proved to be an acceptable alternative for cluster identification. PRFOR was included in the revised NOYELP model and is used for performance comparisons with the parallel kernels discussed below.

Figure 5.2 traces the performance of PRFOR for 1024×1024 random maps as p value increases from 0 to 1. A direct linear relationship exists between execution time and map density. Memory requirements for arrays which record the number of elements per cluster peak near $p = 0.32$, while working stack array sizes increase with increasing maximum cluster size and p value.

5.1.3 Implementation of the Hoshen-Kopleman algorithm

Another serial algorithm for incremental cluster identification, the Hoshen-Kopleman algorithm ([HoKo76]), was implemented later in the thesis effort to serve as a basis for the hierarchically-mapped parallel version of cluster identification. This algorithm (HKFOR) traverses the map to be analyzed pixel by pixel, assigning pixels to temporary clusters as they are encountered. Two working arrays, `level` and `label`, are maintained to keep track of clusters-in-progress. `Level` is the length of a row in the

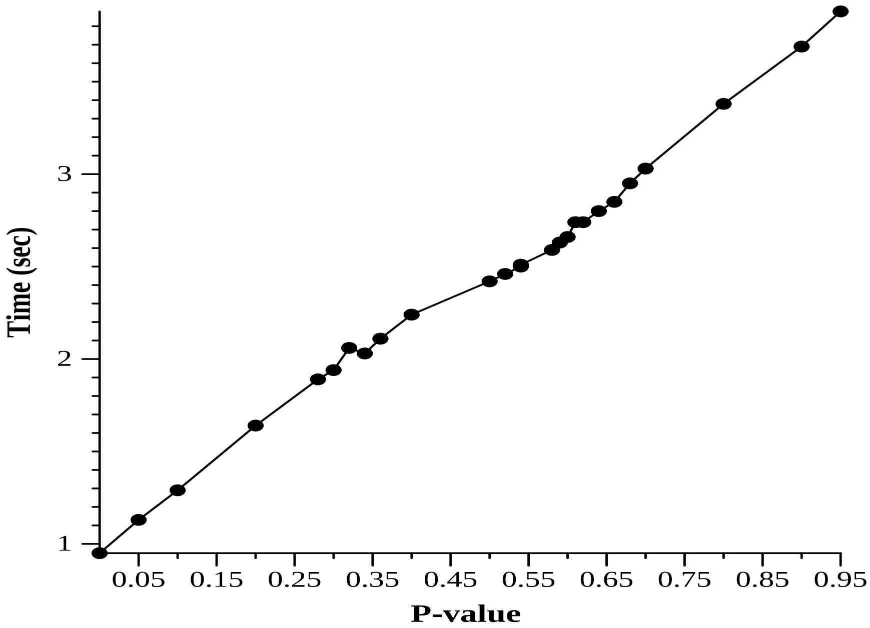


Figure 5.2: CPU time versus p-values ranging from 0 to 1 for 1024×1024 random maps using the serial PRFOR algorithm on a Sun SPARCstation 2 (excluding I/O).

main map, and records the cluster identification label of elements in the previous row analyzed. `Label` is an array which has an entry for each cluster and records an accumulating total of cluster membership. Figure 5.3 illustrates the status of these arrays after one row of the grid has been analyzed. Shaded circles in the `level` array indicate pixels which are not in the map class being analyzed.

Each cluster identification label stored in the `level` array serves as an index into the `label` array, pointing to either a positive or negative number. If the number is positive, it is the number of members to date in that cluster; if it is negative, the absolute value of the number represents the true cluster label/index. As each row of the map is traversed, the ID number of each map element is compared to that of the previous element in that row (west neighbor) and that of the element in the same column of the previous row (north neighbor). If they are all in the same cluster, the smallest label is assigned to all three elements, and any changes in the status of the west and north neighbors are recorded in the working arrays. When the map has been completely traversed, the `label` array holds all final assignments of cluster labels and the number of pixels in each cluster. This innovative 1-pass algorithm is efficient and does not require the large amount of stack space which makes recursive approaches for maps with very large clusters prohibitive on many machines.

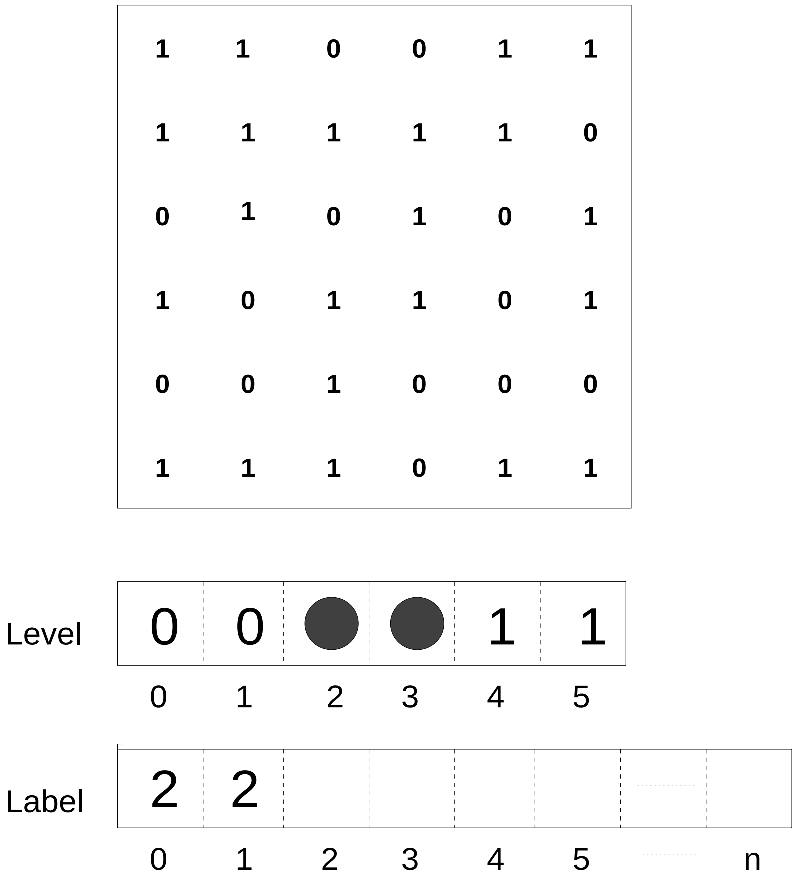


Figure 5.3: Status of *level* and *label* arrays after one row of grid has been traversed using the Hoshen-Kopleman algorithm.

5.1.4 Comparison of serial algorithm performance

Table 5.1 compares the elapsed CPU times (in seconds, excluding I/O) for these four serial cluster identification algorithms on a Sun SPARCstation 2 for seven map sizes and five p values. Execution times for all algorithms increase with map size and density.

The relative efficiency of the four cluster identification algorithms was consistent across map size and density. The incremental cluster identification HKFOR was consistently the most efficient of the four algorithms. The pseudo-recursive PRFOR sequential algorithm demonstrated efficiency comparable to that of HKFOR for maps with the lowest densities, and was marginally but consistently slower at high p values. The recursive C sequential cluster identification algorithm, RECRC, was comparable in efficiency to PRFOR and HKFOR for maps with low densities, but was clearly slower at higher p values. The incremental cluster identification algorithm ORFOR was consistently the slowest over all p values and map sizes.

Cluster identification methods which build clusters incrementally are preferred for SIMD parallel implementations because they embody more inherent parallelism. Even the *local diffusion* method, which is very slow in serial applications, is more efficient as the serial component of SIMD parallel implementations on the MasPar MP-2 than either the recursive or pseudo-recursive algorithms developed in this thesis effort. A modification of the Hoshen-Kopelman algorithm ([FlTa92]), the more efficient of the two serial incremental approaches evaluated, was chosen as the serial component in the parallel cluster identification kernels developed for this thesis.

5.2 MasPar MP-2 Algorithms

Cluster identification algorithms implemented on the MasPar MP-2 employed the two data mapping (virtualization) strategies supported by MasPar in order to handle maps larger than the 64×64 PE grid: cut-and-stack and hierarchical (see Chapter 2 for a discussion of these strategies). For both implementations, work is divided into two distinct tasks: (1) labeling cluster elements and (2) collecting information from the PEs. In Sections 5.2.1 and 5.2.2, the cluster labeling task is discussed for cut-and-stack and hierarchical data mapping, respectively. Collection of cluster data, which is similar for both mapping strategies, is discussed in Section 5.2.3.

5.2.1 Cluster labeling in implementations with cut-and-stack data mapping

In implementing cluster labeling with cut-and-stack mapping, each map pixel with a resource value greater than 0 is initially assigned a unique ID number based on its position in the map. Repetitive comparisons of north-south and east-west pairs of pixels are then made. This is very similar to the serial *local diffusion* method (also called *label propagation* ([ApCM92]), except for the fact that adjacent map pixels are on adjacent PEs, and label comparisons are made using the `xnet` communication construct. Contiguous map pixels in the same cluster are given the cluster ID corresponding to the smallest pixel label in the group of contiguous pixels.

Figure 5.4 shows a simple 4×4 map at four stages of the cut-and-stack labeling process: (a) starting state, (b) after labeling with a unique ID, and after each PE looks (c) north and (d) west. This process of comparison and relabeling continues until all adjacent cluster elements have the same label and no label updating activity is detected (using an activity flag).

Table 5.1: Comparison of CPU times for serial cluster identification algorithms on a Sun SPARCstation 2 (all times are in seconds).

Algorithm	p-value	Map Size						
		64	128	256	512	768	1024	2048
HKFOR	0.10	0.00	0.02	0.07	0.27	0.61	1.25	5.27
	0.30	0.00	0.02	0.10	0.39	0.91	1.67	7.00
	0.62	0.01	0.03	0.13	0.61	1.37	2.60	10.79
	0.85	0.02	0.04	0.19	0.76	1.69	3.23	13.08
	1.00	0.02	0.05	0.21	0.88	1.96	3.73	14.74
ORFOR	0.10	0.01	0.02	0.17	0.70	1.60	3.00	13.45
	0.30	0.02	0.05	0.27	1.13	2.54	4.59	20.00
	0.62	0.02	0.10	0.44	1.79	4.13	7.23	30.34
	0.85	0.03	0.12	0.59	2.32	5.41	9.41	38.88
	1.00	0.04	0.14	0.69	2.74	6.35	11.11	45.72
PRFOR	0.10	0.00	0.01	0.06	0.31	0.74	1.30	5.22
	0.30	0.01	0.02	0.11	0.46	1.13	1.91	7.80
	0.62	0.01	0.03	0.17	0.68	1.56	2.75	11.09
	0.85	0.01	0.04	0.22	0.88	2.04	3.56	14.62
	1.00	0.02	0.06	0.25	1.08	2.54	4.49	18.03
RECRC	0.10	0.01	0.02	0.10	0.41	0.93	1.69	10.68
	0.30	0.01	0.03	0.13	0.52	1.18	2.08	12.26
	0.62	0.01	0.04	0.17	0.70	1.62	2.87	22.32
	0.85	0.02	0.16	0.59	1.62	5.32	7.21	*
	1.00	0.03	0.23	0.92	3.76	8.78	9.38	*

*exceeds stackspace limits

HKFOR: Hoshen-Kopleman (Fortran-77)
PRFOR: Pseudo-Recursive (Fortran-77)
ORFOR: NOYELP (revised) (Fortran-77)
RECRC: Recursive (C)

5.2.2 Cluster labeling in implementations with hierarchical data mapping

The hierarchical cluster labeling strategy adapted for parallel algorithm development was influenced by the work of Tamayo ([FlTa92]) in quantum physics. The process, which involves three-steps, is shown in the context of the entire cluster identification process in Figure 5.5 and is discussed below. In Figure 5.5, the two large boxes represent adjacent PE subgrids, each with 9 data elements (map pixels). The number in the center of each small box (one map pixel) represents the resource level in the *Starting State*, and the cluster label thereafter. The number in the lower right corner of each pixel represents the unique pixel label. The circled number in the upper left corner represents the total number of pixels in a cluster (stored at the head pixel of each cluster). The box in the upper right-hand corner denotes a local cluster head and contains a local cluster sum.

1. **Step 1.** As was the case for implementations employing cut-and-stack mapping, each map pixel is initially assigned a unique ID number (label) based on its position in the map. Clusters are first resolved locally (within the subgrid of each PE) using an adaptation of the Hoshen-Kopelman algorithm ([StAh91]) for incremental cluster identification. Modifications to the serial Hoshen-Kopleman algorithm required for parallel implementation included labeling each pixel with its local cluster number and storing local cluster sums at the local head pixels for local clusters. Each pixel is assigned a pointer to the local head of its cluster (i.e., the cluster member on its PE with the smallest label).

   Alternative recursive or pseudo-recursive algorithms for this serial step in cluster labeling are not well-suited to an SIMD approach and are much less efficient. With the sequential cluster labeling pseudo-recursive algorithm, subgrid traversal stops when a local cluster head is encountered and local building of that cluster takes place. As a consequence, some processors are idle while local cluster building takes place on one or more other processors, resulting in poor performance. Since the Hoshen-Kopleman adaptation adds new members to each cluster as they are encountered in local subgrid traversal, work is distributed more evenly over local subgrids for more efficient SIMD performance.

2. **Step 2.** For each PE (in parallel), comparisons are made of local cluster labels in border rows and columns with those in immediately adjacent rows and columns of adjacent PEs The smallest cluster label is assigned across PE boundaries to border elements of the adjacent PE which are in the same cluster. Label changes are then transferred (by pointer) to the local head of the cluster to which each border pixel belongs. An activity flag is set whenever relabeling occurs.

3. **Step 3.** The process of border comparison and relabeling is continued until an equilibrium state is reached in which corresponding border cluster elements on all adjacent PE's share the same cluster label. This state is detected when no activity flags are set recording relabeling activity. Total number of pixels in each cluster are stored at the cluster's head pixel.

   The number of iterations required for complete label propagation depends on sizes and densities of the clusters. For random maps, p values near the critical region (0.5928) result in cluster characteristics requiring the maximum number of iterations for resolution. When maps are sparse, clusters are smaller and labels do not have far to propagate. When maps are denser, most pixels belong to the same cluster, requiring little relabeling.

5.2.3 Collection of cluster data

In serial implementations of cluster identification, **collection** of cluster data is accomplished within the cluster labeling function. Cluster size is accumulated in arrays with one entry per cluster. For parallel implementation, where memory is somewhat limited on both the ACU and the DPU, potentially large arrays such as these cannot be maintained. Cluster data must be maintained on the PEs, and are not collected until after cluster labeling has been completed.

For both mapping strategies, collection of cluster data is accomplished by having all members of a cluster report to the local head pixel on their PE, which then reports to the global head pixel for that cluster (i.e., the cluster member with the smallest original label over all processors, whose ID number has been used to label all other members). Local collection for hierarchical implementations is shown in Step 1 in Figure 5.5. The boxes in the upper right-hand corner of each grid cell denote local cluster sums. Implementations of both mapping strategies would be expected to benefit from the local collection of cluster information for each cluster represented on a PE before sending the sums to the head element. Local collection would be expected to improve efficiency of the hierarchical algorithm more for sparse random maps ($p = 0.10$ and 0.30) than for dense random maps because of the high likelihood that clusters would be confined to individual PEs. Local collection would improve the cut-and-stack version only for denser maps ($p > 0.59$), since only the large clusters typically found in dense maps are likely to have a significant number of members on the same processor.

For either mapping strategy, each member of a cluster can calculate the address (PE number and subgrid position) of the head pixel of that cluster from its own final cluster label, as follows:

$$\text{PE number} = \text{label} \ \texttt{mod} \ \text{nproc},$$
$$\text{layer} = \text{label} \ / \ \text{nproc},$$

where PE *number* is a unique processor identifier, *layer* is a virtualized data layer on that processor (see Chapter 2 for a discussion of virtualization), and *nproc* is the total number of PEs, which is 4096 for the MasPar MP-2. The MPL communication construct `p_sendwithAdd()` is used to report membership of the pixel to the head element of the cluster to which it belongs (by sending a 1). Sums representing total cluster membership are maintained by each head element. Global collection is denoted by circled integers in the upper left-hand corner of cells, shown in Step 3, Figure 5.5. The MPL function `reduceMax()` is then used to find the size of the largest cluster, as follows:

```
largestcluster= reduceMax(clustersize).
```

where `largestcluster` is a singular variable and `clustersize` is a plural value allocated on all PEs. Number of clusters is determined by counting head elements, as follows:

Table 5.2: Speed improvement of MasPar MP-2 versions over pseudo-recursive Fortran version on a SPARCstation 2 for cluster identification (excluding I/O).

Mapping strategy	p value	Map Size 64	128	256	512	768	1024	2048
Cut-and-stack	0.10	0.00	0.62	4.00	2.48	1.74	1.27	0.41
	0.30	0.83	1.25	2.35	2.19	1.87	1.34	0.55
	0.62	0.23	0.35	0.47	0.54	0.60	0.50	0.34
	0.85	0.18	0.51	1.13	1.33	1.42	1.36	1.24
	1.00	0.39	0.73	1.47	1.95	2.14	2.14	2.23
Hierarchical	0.10	0.00	2.50	5.00	8.21	8.44	7.06	4.81
	0.30	0.83	2.50	7.50	7.12	6.89	5.31	3.24
	0.62	0.21	0.73	1.67	3.16	4.08	4.04	2.14
	0.85	0.20	0.65	3.14	8.98	12.68	15.08	12.40
	1.00	0.36	1.02	4.58	15.41	28.83	38.84	52.29

```
if(myclusterlabel==myoriginallabel) reduceAdd32(one).
```

The average cluster size is simply calculated by dividing the number of non-zero elements by the number of clusters.

In the early stages of parallel algorithm development, collection time dominated cluster identification time, largely because of the improper functioning of the initial MasPar MPL version of the `p_sendwithAdd()` function (used to collect data on head elements). The error was reported to MasPar, and a less efficient collection strategy which required all receiving PEs to be in an active state was devised as a temporary alternative. The `p_sendwithAdd()` error was corrected in subsequent software releases, and modifications to the collection algorithm utilizing the corrected function resulted in a substantial decrease in cluster data collection time.

5.3 Results

For both parallel cluster identification programs implemented on the MasPar MP-2 (i.e., hierarchical and cut-and-stack), total elapsed wall-clock time was compared with that of the pseudo-recursive serial Fortran version (PRFOR) on a Sun SPARCstation 2 for random maps of seven sizes and five densities (Table 5.2). For these comparisons, *work* time is defined as the sum of *label* and *collect* times for cluster identification, excluding time for reading data from a binary input file and writing results to the screen (I/O). Speed improvements are calculated by dividing work time for the serial program by work time for each parallel implementation. Tables C.1 through C.3 in Appendix C list actual read, work, and total wall-clock times for both MasPar MP-2 versions and for PRFOR.

Trends in performance of the two parallel kernels on these random maps across

p values within a given map size and across map sizes within a given p value were generally consistent. For the smallest maps (i.e., 64×64), both parallel algorithms were generally less efficient than the serial algorithm (i.e., speed improvement values less than 1.00). The cut-and-stack algorithm was also slower than the serial algorithm for maps with p values near the percolation threshold ($p = 0.62$) regardless of map size, and for all 2048×2048 maps of densities $<= 0.62$. In contrast, the hierarchical algorithm showed speed improvements over the serial algorithm for all p values for map sizes larger than 128×128.

Figure 5.6 graphically presents the speed improvements of the hierarchically-mapped MasPar MP-2 implementation for these same mapsizes and p values. Of the five densities considered, the worst parallel algorithm performance was typically seen for maps with $p = 0.62$, near the percolation threshold (Table 5.2 and Figure 5.6). For both parallel algorithms, an overall trend of maximum speed improvement at smaller than maximum map size was evident for all but the densest maps (i.e., those with $p = 1.00$). The greatest speed improvement for the hierarchical algorithm (52.29) was observed for the densest and largest map, while the greatest speed improvement for the cut-and-stack algorithm (4.00) was observed for the sparsest map (i.e., $p = 0.10$) of size 256×256.

For all random maps which were virtualized (i.e., those larger than 64×64), the hierarchically-mapped kernel was more efficient than its cut-and-stack counterpart, with the relative performance of the hierarchical algorithm generally increasing with increasing map size and density (Table 5.2). For sparse maps ($p = 0.10$ and 0.30), hierarchical mapping benefited from the local collection of cluster data on PEs, which reduces the number of `p_sendwithAdd`s required to transmit data to the head element of each cluster when cluster elements are sparsely distributed. Local collection is useful for cut-and-stack mapping only when clusters are large enough to have multiple members on each PE. For dense maps dominated by a single large cluster ($p \geq 0.62$), the cluster labeling component dominates cut-and-stack execution time. The hierarchical mapping strategy outperforms cut-and-stack mapping for these denser maps because it labels large clusters more efficiently.

An increase in the speed of `xnet` communications projected for future MasPar releases would improve the relative performance of the cut-and-stack algorithm, which requires more inter-processor communication than the hierarchical algorithm.

Figure 5.7 shows how the labeling and collection components of cluster identification using the two mapping strategies perform as p values increase from 0 to 1 for maps of sizes 256×256 and 2048×2048. Elements (pixels) for these graphs were produced by MPL programs using the MPIPL routine `mpigenrand` to generate and distribute appropriate random values on MP-2 PEs for each p value. Note the y-axis (time) scale differences for graphs showing results from the two mapping strategies. This reflects the consistently better performance of the hierarchical algorithm across all p values and both map sizes considered. Hierarchical total time is superimposed on cut-and-stack graphs for comparison. Performance differences for the hierarchical and cut-and-stack algorithms are most evident for cluster labeling of maps with densities near the $p = 0.59$ threshold, especially for the larger maps.

A comparison of these results with the changes seen in cluster characteristics across p values for random maps (e.g., compare Figure 5.7 with Figure 4.2(a) in Chapter 4) illustrates their relative importance in determining execution times of cluster identification algorithms. For the random maps analyzed, peak execution (i.e., work) times for both parallel cluster identification algorithms occur at p values near the 0.59 threshold, when the maximum cluster size begins to climb dramatically and the number of clusters is still relatively large. This is the point at which the maximum amount of border updating and label reassignment within PE subgrids is required. Execution times decrease as map density increases and one cluster becomes dominant. Under these conditions, most PE subgrid elements and border elements on adjacent PEs belong to the same dominant cluster, requiring much less border updating and label reassignment.

Memory requirements for the two SIMD algorithms are constant for a given map size over all p values. This is in sharp contrast to results for the serial algorithm PRFOR, which is characterized by increasing CPU time and memory requirements as the maximum cluster size increases (see Figure 5.2). These differences in the relationship of execution time to cluster characteristics for serial and parallel implementations largely explains why speed improvements of parallel kernels were generally lowest for random maps with p values near the percolation threshold. In fact the cut-and-stack cluster identification algorithm was slower than the serial Fortran algorithm at $p = 0.62$ for all map sizes (Table 5.2). By studying these relationships during code development and by anticipating cluster characteristics of non-random maps for a particular application, parallel processing bottlenecks can be pinpointed, and an optimized cluster identification strategy can be developed for each parallel application.

A more efficient cut-and-stack approach to cluster identification has been proposed ([ApCM92]) which includes power-of-2 neighbors (on processors which are 2^n units away, $0 < n < m - 1$, where map size $= 2^m \times 2^m$) in the list of neighbors which are checked on each iteration of cluster label updating. This approach should reduce the number of iterations required for complete label propagation for the dendritic clusters characteristic of maps with p values near the critical threshold.

5.4 Performance of MasPar MP-2 algorithms on NOYELP maps

5.4.1 Test map characteristics

Figure 5.8 illustrates the temporal pattern of density variation (expressed as p value) in maps of available biomass (resources) generated during a 180-day cycle of the NOYELP model. Resource maps of the study area extracted from the same 180-day cycle of the NOYELP model at day 1, day 90, day 120, and day 180, respectively, are presented in Figures 5.9 through 5.12.

All portions of the maps which are light green represent available resource pixels. Pixels which have high resources levels, defined as sufficient available biomass to satisfy at least 50% of the daily forage intake requirements of a bison bull, are represented by red.

At the beginning of the model year (November), resource levels, p values, and maximum cluster size are all high (Figures 5.8 and 5.9). As winter progresses, resources are depleted by the grazing of ungulates or are made inaccessible to the ungulates by heavy snowfall. This is reflected in lower p values for the resource maps (Figure 5.8), smaller, more fragmented clusters, and lower levels of available biomass (Figure 5.10 and 5.11). The model year ends in April before spring regeneration of vegetation begins; however, melting snow exposes ungrazed resources, causing map density to increase near the end of the 6-month cycle (Figures 5.8 and 5.12).

5.4.2 Results

Cluster identification is performed twice at each time step in the NOYELP model, once for clusters of any available resource level and again for clusters with available resources above a fixed limit (high resources).[1] Table 5.3 compares performance of the two MPL cluster identification algorithms with that of the serial NOYELP algorithm (PRFOR) on 285×584 resource maps extracted from the NOYELP model runs for seven days in the 180-day cycle. Maps were expanded to 320×640 for input to the MasPar MP-2 kernels, so that row and column dimensions are multiples of 64, the size of the PE grid. The p values calculated in Table 5.3 are based on the total number of map pixels in the expanded resource map, and include pixels outside the study area, while the graph of p values in Figure 5.8 includes only study area pixels.

Wall-clock time for the hierarchical parallel implementation is typically 4 to 5 times faster than that for the serial algorithm, while the cut-and-stack parallel implementation is slower than the serial algorithm for most maps. These results are not inconsistent with the relative performance of the parallel kernels on random maps of

[1] This is a simplification of what the actual resource matrix represents. Cluster analysis is performed on the matrix of feedback modifiers which limit the maximum daily intake of forage by ungulates, based on limitations associated with levels of forage biomass and snow depth/density.

Table 5.3: Comparison of wall-clock times for parallel implementations of cluster identification with the serial NOYELP function PRFOR on Sun SPARCstation 2 on 285 × 584 resource maps extracted from NOYELP model runs (all times are in seconds).

Time step	Resource level	p value	Largest Cluster	Serial PRFOR	Parallel hierarchical	Parallel cut-and-stack
1	any	0.30	55554	0.41	0.08	0.41
1	high	0.27	51035	0.38	0.08	0.43
30	any	0.30	55554	0.40	0.08	0.41
30	high	0.27	51045	0.37	0.08	0.43
60	any	0.29	53800	0.39	0.07	0.40
60	high	0.22	29945	0.33	0.07	0.45
90	any	0.18	16515	0.31	0.06	0.50
90	high	0.04	4701	0.21	0.05	0.18
120	any	0.06	6158	0.23	0.05	0.23
120	high	< 0.01	394	0.19	0.03	0.04
150	any	0.03	6153	0.23	0.05	0.23
150	high	< 0.01	0	0.19	0.02	0.02
180	any	0.29	55190	0.39	0.07	0.41
180	high	0.14	4672	0.28	0.06	0.47

similar map size and maximum cluster size, except for the relatively poor performance of the cut-and-stack implementation compared to PRFOR. Future work in this area should involve a closer examination of the the range of cluster configurations and characteristics encountered in landscape maps, the relative performance of parallel kernels on these configurations, and methods of optimizing parallel performance over the range of configurations.

5.5 Conclusions

Speed improvements of SIMD parallel algorithms on the MasPar MP-2 over serial algorithms for cluster identification are modest, largely because of the high communication requirements associated with labeling of pixels. These results indicate that the serial algorithms developed and evaluated are efficient enough for many purposes. However, the hierarchical parallel algorithm consistently outperformed both the serial and cut-and-stack algorithms, and could be useful in applications such as the NOYELP model which call cluster identification functions many times within a single program execution. NOYELP identifies clusters twice at each time step (once for high resource patches and once for patches with any resource) over 180 time steps and over 5 replications of the 180-day cycle. This requires a total of 1800 calls to the cluster identification function. Modest time savings per cluster analysis can result in significant savings in total execution time for models such as NOYELP. For a typical serial NOYELP simulation involving 20,000 ungulates, approximately 20% of total execution time for the revised model is spent identifying clusters (see Chapter 7 for a discussion of serial modifications). A simulation involving fewer animals would require that less model time be spent in the animal movement component and proportionately more time be spent in cluster identification. The presence of fewer animals would also mean less resource depletion due to grazing, and hence more pixels with available biomass (i.e., higher p values) over the course of model execution. This would further increase the proportion of total execution time allocated to cluster identification. Therefore, the fewer animals input to NOYELP, the more important the efficiency of the cluster identification component becomes.

To utilize the parallel cluster identification kernels for NOYELP model simulations, the serial Fortran-77 NOYELP program could be run on the front end machine of the MasPar MP-2 (DECstation 5000-200 workstation) and make calls to the MPL cluster identification function running on the DPU, which would then *blockIn* the data matrix to be analyzed. Since cluster identification results do not feed back into the main NOYELP program (results are stored in an array for later output to a file), an asynchronous call via `callAsync()` would allow the main program to continue execution on the front end while cluster identification is being accomplished on the DPU. Output from the parallel cluster identification kernels could be input to other parallel modules to compute various cluster geometry indices, such as mean squared radius, on distributed data on the DPU. Mean squared radius algorithms are discussed in detail in the following chapter.

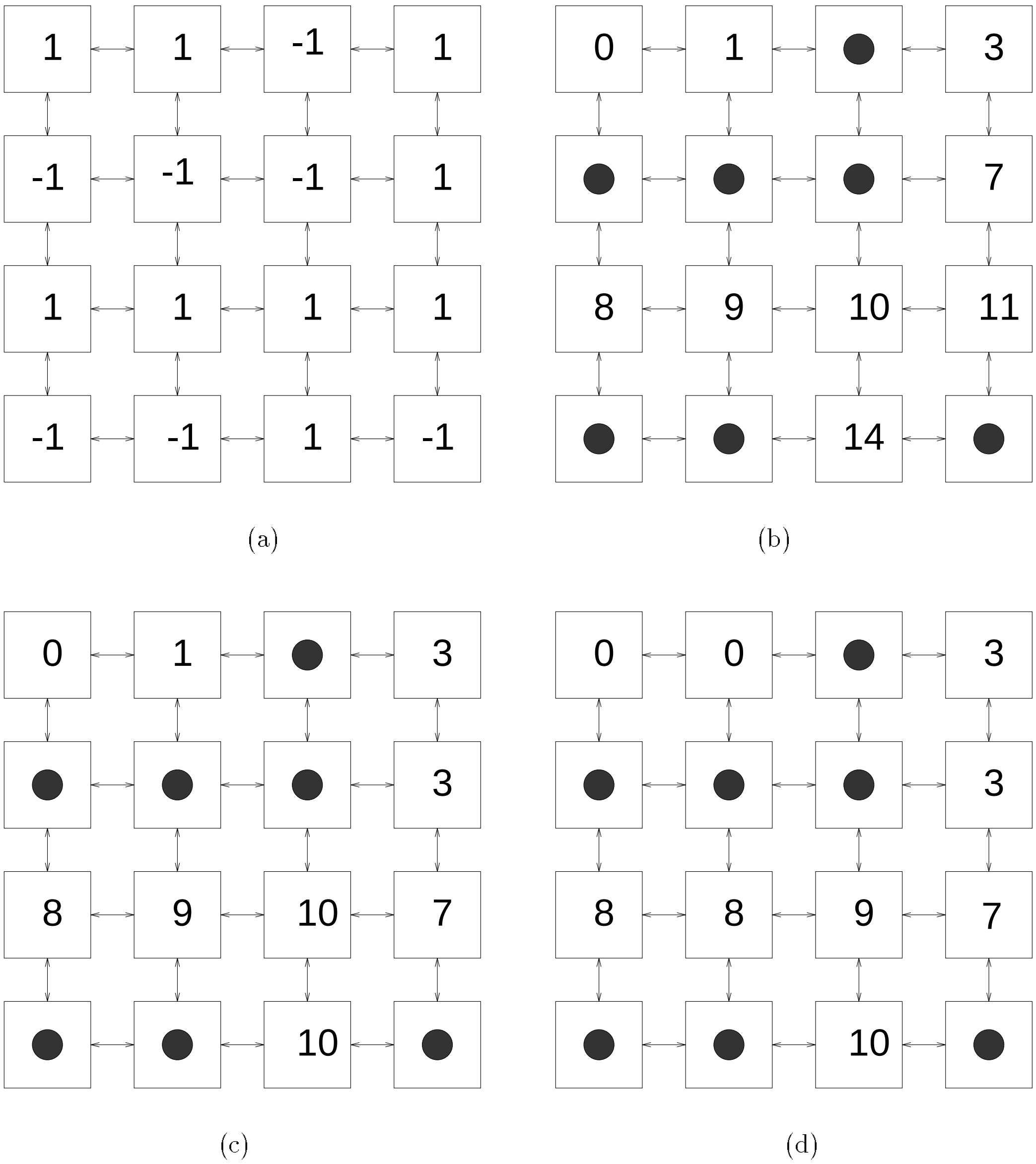


Figure 5.4: Four stages of the cut-and-stack labeling process: (a) starting state, (b) after labeling with unique ID, and after each PE looks (c) north and (d) west.

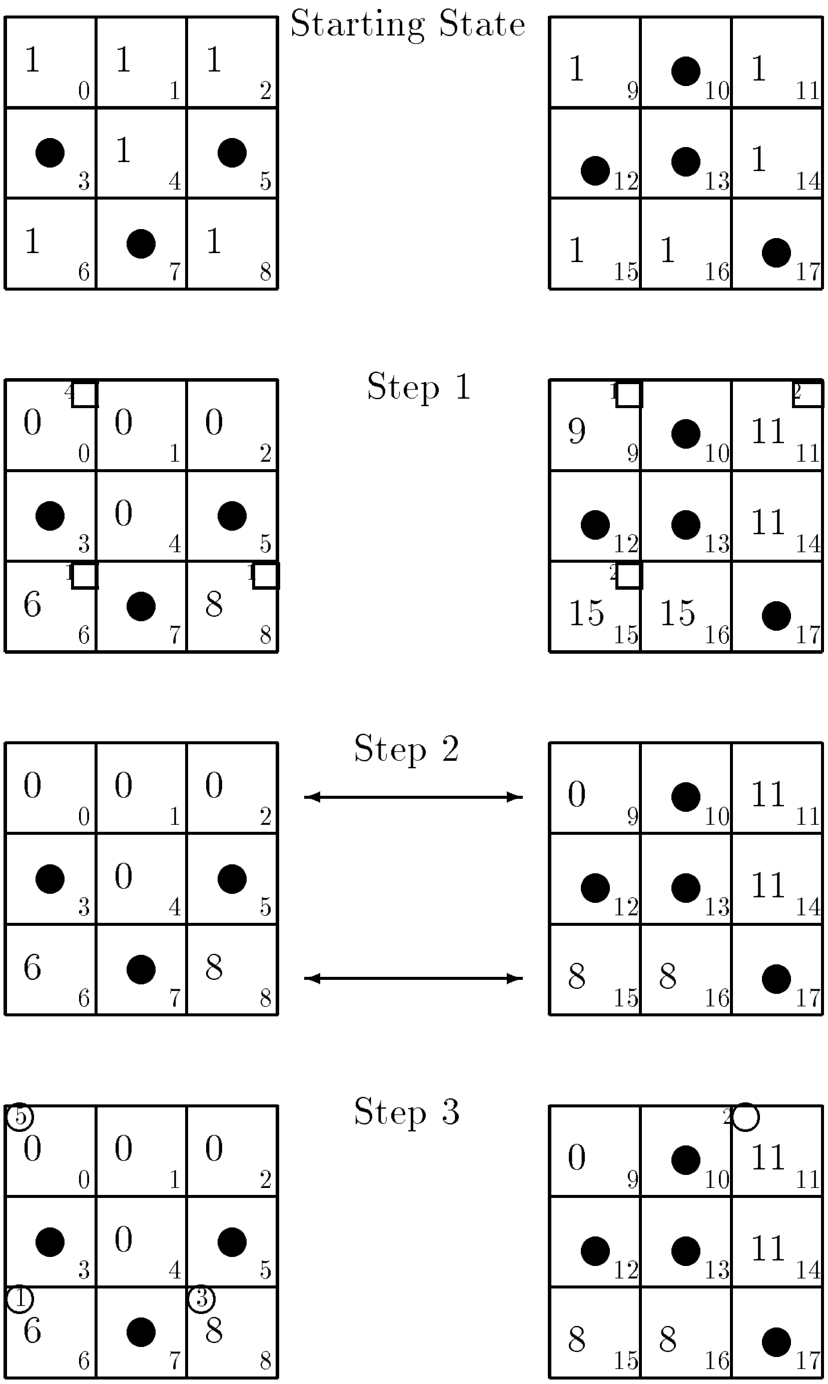


Figure 5.5: Step-wise procedure for cluster identification in hierarchically-mapped MPL implementation of cluster identification.

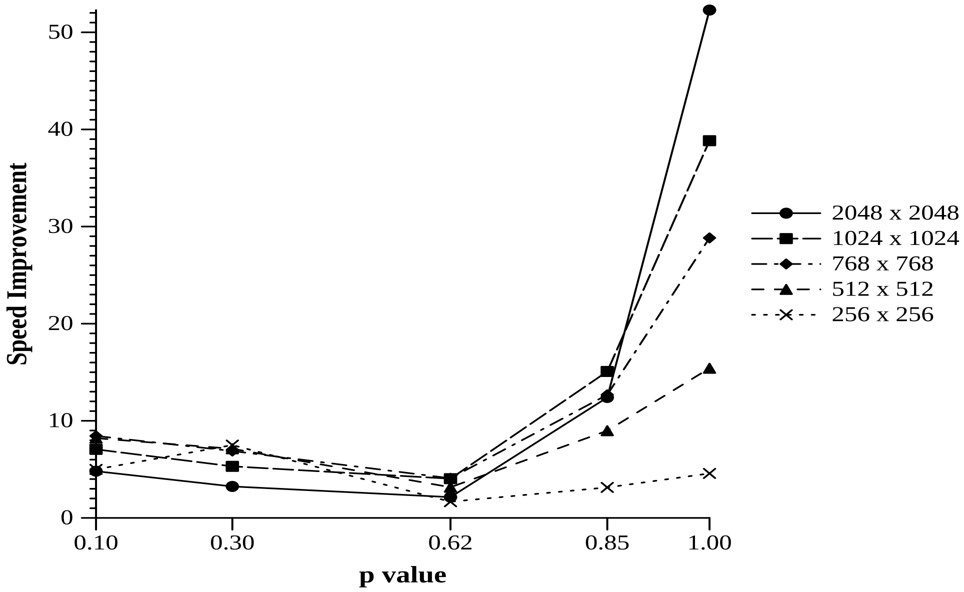


Figure 5.6: Speed improvement of MP-2 hierarchically-mapped implementation over the sequential PRFOR version on a SPARCstation 2 for cluster identification (work time, excluding I/O).

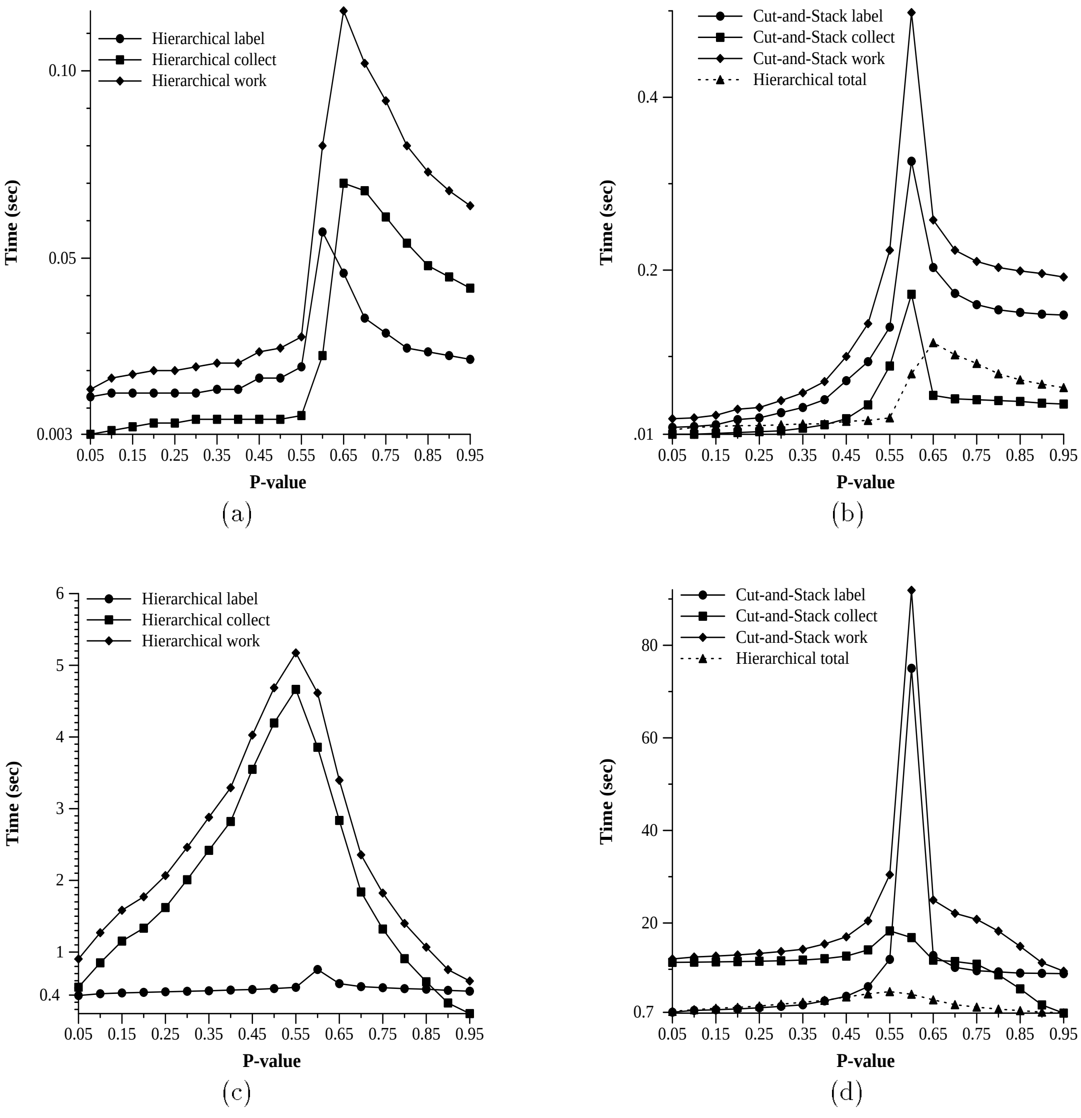


Figure 5.7: Comparison of wall-clock labeling, collection, and total work times versus p values for 256×256 maps ((a) and (b)) and 2048×2048 maps ((c) and (d)) using hierarchical ((a) and (c)) and cut-and-stack ((b) and (d)) virtualization.

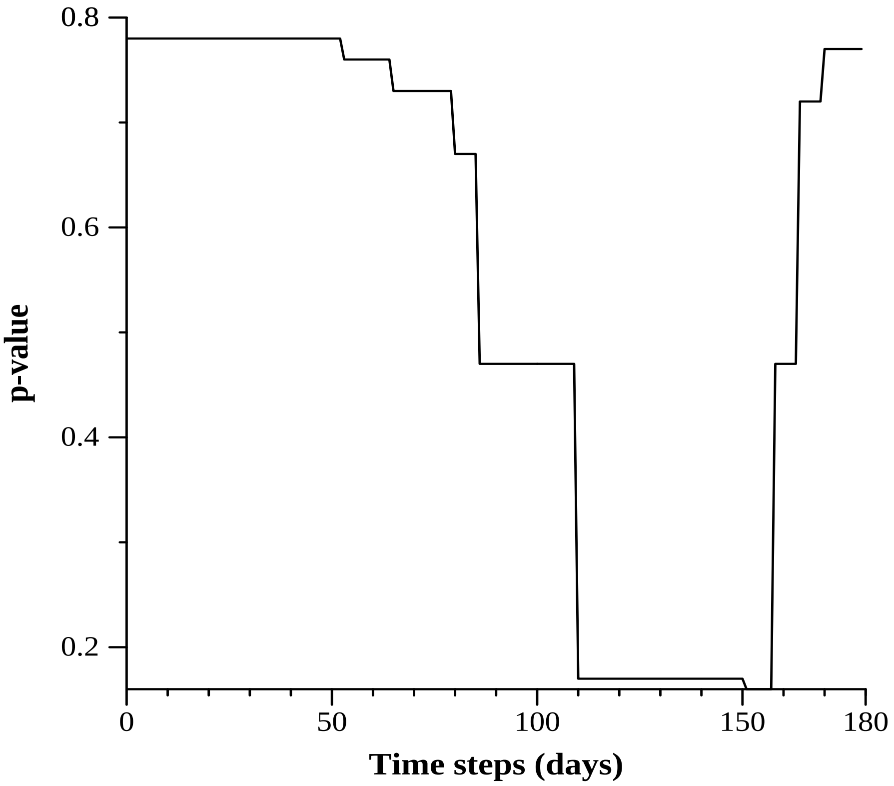


Figure 5.8: P values of the available resource matrix generated daily for a 180-d ay cycle of the NOYELP model (excluding pixels outside the study area).

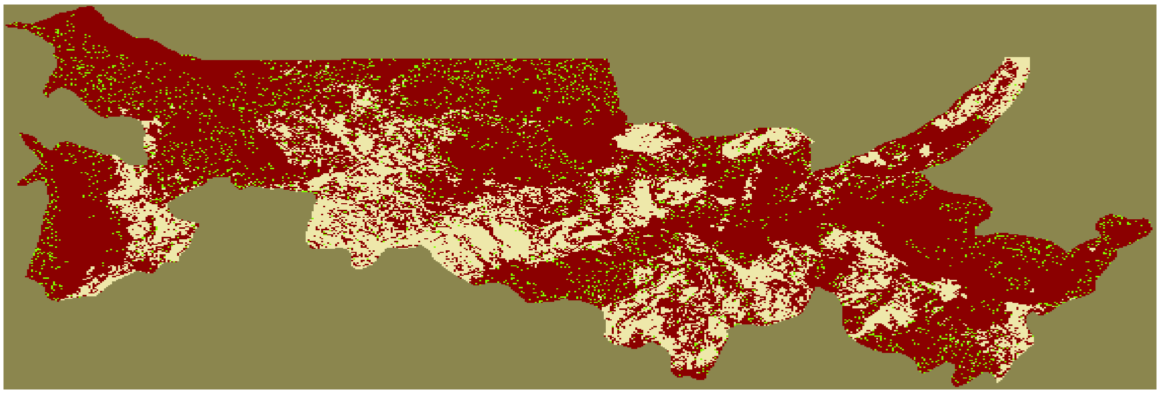


Figure 5.9: Resource map of the NOYELP study area extracted at day 1 from a 180-day cycle of the NOYELP model.

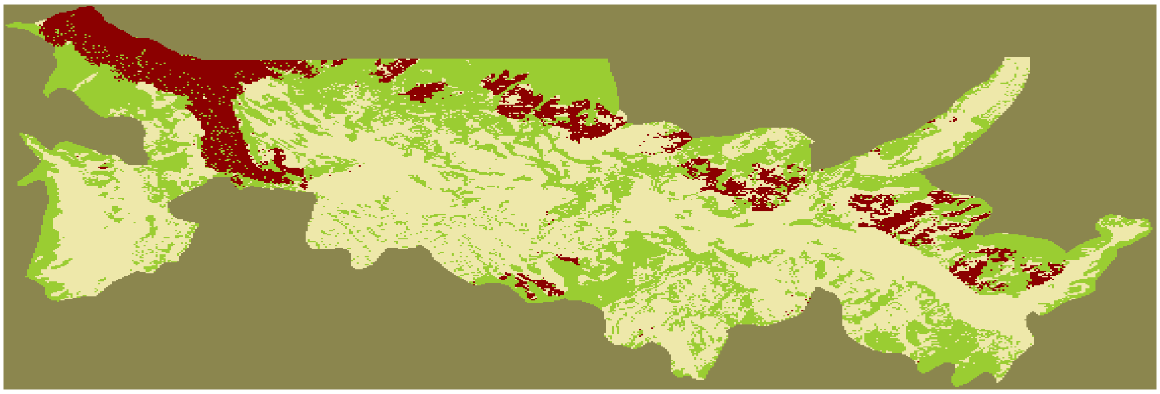


Figure 5.10: Resource map of the NOYELP study area extracted at day 90 from a 180-day cycle of the NOYELP model.

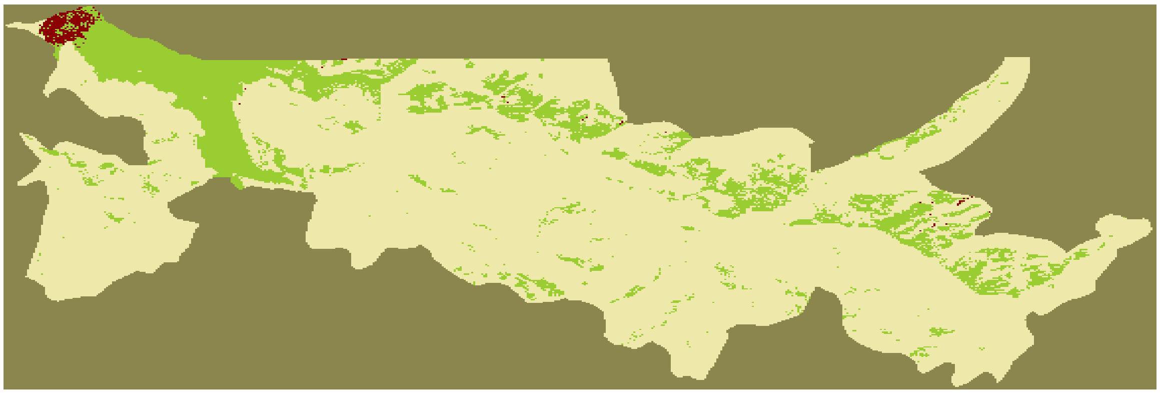


Figure 5.11: Resource map of the NOYELP study area extracted at day 120 from a 180-day cycle of the NOYELP model.

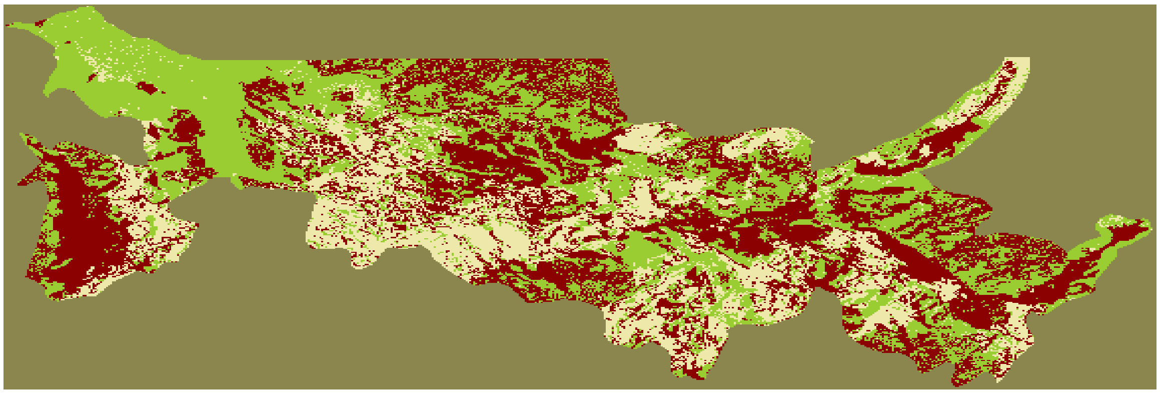


Figure 5.12: Resource map of the NOYELP study area extracted at day 180 from a 180-day cycle of the NOYELP model.

Chapter 6

Cluster Geometry

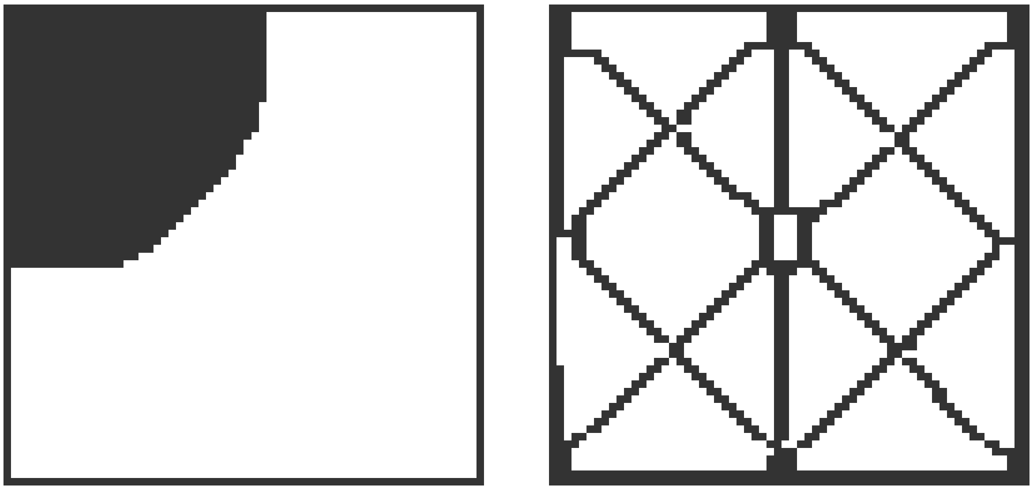
Once clusters in a map have been identified (i.e., labeled and counted) other descriptive statistics may be computed to describe the geometry of clusters, such as *radius*, *mass*, *perimeter*, and *correlation length* ([StAh91]). The radius measure, which is used to derive correlation length, is the focus of thesis efforts presented in this chapter. The radius of a cluster is the average distance between two cluster pixels, providing a measure of the compactness of a cluster. A cluster whose members are widely dispersed across a map will have a larger radius than a cluster whose shape approximates a regular polygon. Figure 6.1 shows two 64×64 maps, each with a single cluster having 1024 member pixels. The radius for the compact cluster on the left (a) is 14.12, while that of the more dispersed cluster on the right (b) is 31.09.

This chapter describes the data-parallel implementation of the mean squared radius (R^2) computation for clusters, from which the *radius* measure is derived. R^2 of an individual cluster is defined as the sum of all squared intra-cluster distances between pixels, divided by two times the squared cluster size. Each squared distance is calculated by squaring the row and column differences between the x- and y-coordinates of two pixels. Because a 1 is added to coordinate differences before squaring, the absolute values of coordinate differences are used in the calculation of R^2. The formula for R_n^2 of a cluster (as derived from [StAh91]) may be given as:

$$R_n^2 = \frac{\sum_{i,j}(|x_j - x_i| + 1)^2 + (|y_j - y_i| + 1)^2}{2n^2}\ , \tag{6.1}$$

where x_i, y_i and x_j, y_j are the coordinates of pixels i and j, respectively, (for $1 \leq i \leq n$ and $1 \leq j \leq n$) and n is the number of elements in the cluster.

The computation of cluster radius ($\sqrt{R_n^2}$) has many potential uses in landscape ecology. For example, if it has been determined through cluster identification that a large fire had spread over 60% of a landscape, computation of cluster radius would provide insight into whether the fire is concentrated in one compacted area, or if significant amounts of unburned area exist within the area of the fire cluster.



(a) (b)

Figure 6.1: Example 64×64 maps with a single cluster of size 1024 and radius equal to (a) 14.16 and (b) 31.09.

The computationally-intensive nature of the derivation of R^2 has limited its usefulness in many applications. The serial computation of R^2 for all clusters in a 512×512 random map with a p value of 0.85 typically requires about 12 hours (CPU time) on a Sun SPARCstation IPX. Parallel implementations of mean squared radius algorithms offer a way to address this problem.

6.1 Serial Algorithm

The serial program for mean squared radius computation used for timing comparisons with data-parallel implementations presented in this chapter is a modified version of a Fortran-77 program developed by Dr. Robert Gardner of the Environmental Sciences Division at Oak Ridge National Laboratory and rewritten in C by Karen Minser of the Computer Science Department at the University of Tennessee ([Mins93]). This serial program implements an algorithm in which the grid is sequentially traversed for each cluster. The x- and y-coordinates for each pixel belonging to the current cluster are stored in arrays. After all coordinates have been stored, differences in coordinate distances are calculated, squared, and added to an accumulating total.

For each pixel, squared differences between its x- and y-coordinates and those of every other member of the cluster are computed as follows:

```
for i = 1 to n − 1 do
  for j = i + 1 to n do
    rx = abs(xcoord[j]-xcoord[i])+1
    ry = abs(ycoord[j]-ycoord[i])+1
    rsum = rsum + rx * rx + ry * ry
  enddo
enddo
```

where the x- and y-coordinates of all members of the cluster are stored in the arrays *xcoord* and *ycoord*, respectively, and *rsum* is the (accumulated) sum of squared differences for each pixel. Hence, the computational complexity of Equation 6.1 is $\mathcal{O}(n^2)$, where n is the number of pixels in the particular cluster.

The serial C program was optimized for performance as part of this thesis effort, and timing results for the optimized program are used for comparisons with MasPar MP-2 parallel kernels. Decreasing the size of dynamically allocated coordinate arrays, computing absolute values in place rather than calling the built-in C function `abs()`, and removing calculations from *for* loop indices reduced execution time substantially. Memory demands for processing maps with large clusters are lessened by computing squared coordinate differences directly from *for* loop indices rather than storing x- and y-coordinates in arrays. Optimizing modifications reduced CPU time for computing R^2 of all clusters in a 512×512 random map with $p = 0.85$ on a Sun SPARCstation IPX from 25 hrs. to 12 hrs. Further modification employed a look-up table of pre-calculated distances read from an infile. The use of look-up tables (with separate tables for each map size analyzed, stored in binary files) could provide significant speed improvements for some applications. Performance improvements gained by each of these serial program modifications are described in Appendix C.

Figure 6.2 shows the performance of the optimized serial algorithm on random 128×128 maps as the p value increases from 0 to 1. A comparison of this trend with that of maximum cluster size vs. p value (Figure 6.3) suggests that maximum cluster size is a major factor contributing to total computing time for R^2 computation for random maps. As maximum cluster size increases sharply at densities above the percolation threshold ($p = 0.5928$), there is a corresponding increase in CPU time required to compute R^2. This reflects the effective exponential nature of the relationship between cluster size and number of computations required.

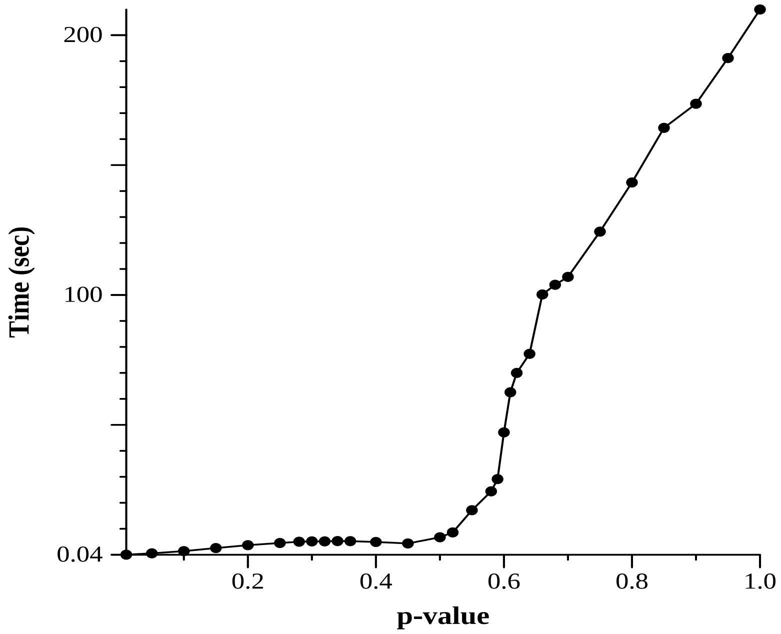


Figure 6.2: CPU times for serial mean squared radius computations across p-values for 128×128 random maps on a Sun SPARCstation IPX.

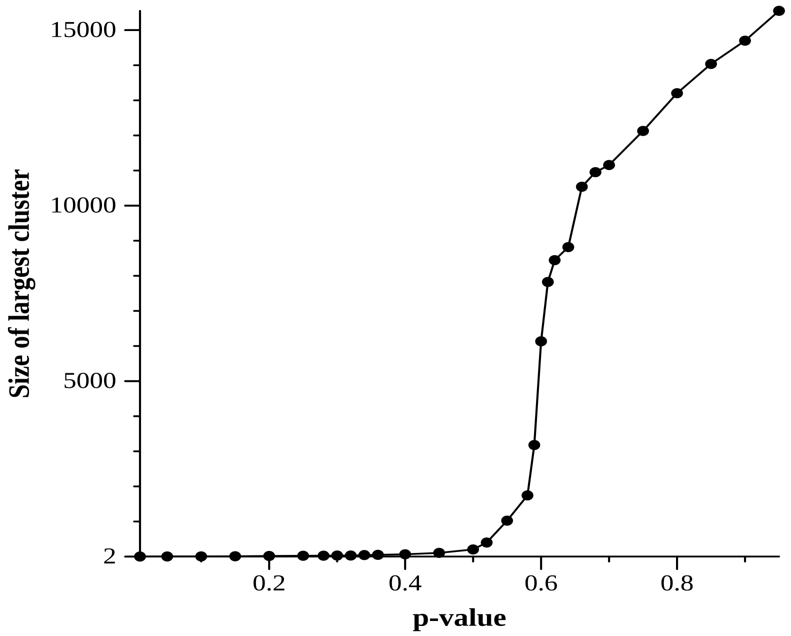


Figure 6.3: Size of the largest cluster as a function of p value for the 128×128 random maps from Figure 6.1.

6.2 MasPar MP-2 Algorithms

Two parallel algorithms for computing mean squared radius on the MasPar MP-2 were implemented, one for each of the two explicit data mapping strategies used: cut-and-stack and hierarchical. As discussed in Chapter 2, the two virtualization methods differ in how pixels are allocated to PEs. In hierarchical mapping, continuous blocks of the data map are assigned to each PE. Small clusters may be wholly contained in the subgrids of individual processors, and can be processed locally. For cut-and-stack mapping, contiguous map pixels are located on adjacent PEs, and must be accessed by the MPL communication construct `xnet()`. The number of pixels assigned to each PE is determined by the size of the map. Regardless of the method of virtualization, data for a specific variable relevant to cluster geometry computations (e.g., cluster size) for the pixels on the PE are stored in stacks or data arrays, with the length of the array (i.e., number of layers) being determined by the number of pixels assigned to each PE. Every pixel in the original map is represented in the array of one of the PEs, and the stacks or data arrays in all PEs are of the same length. For data such as cluster size, which is stored at the head pixel of each PE, these data arrays will contain many zeros. In the following discussion, *layer* refers to a 64×64 matrix of pixel data, all elements of which bear the same array index, and *sub-grid* refers to the contiguous chunk of the map allocated to each PE in hierarchical mapping. Because a layer has special relevance to cut-and-stack mapping, it is called by a special name, *page*. *Page* is an abstraction referring to a 64×64 block representing a given *layer* across all PEs in cut-and-stack mapping.

Both parallel R^2 computation algorithms are similar in overall approach, employing the same three-step process: (1) resolve small clusters in parallel, (2) resolve large clusters by copying pixel cluster labels and positions serially to shared data space, and (3) collect sums for each cluster across members (pixels).

Figure 6.4 illustrates these steps for the hierarchical version. The Starting State in Figure 6.4 represents the status of cluster analysis following completion of cluster identification (see Figure 5.5 in Chapter 5). The large boxes in Figure 6.4 represent two adjacent PE subgrids, each with 9 data elements (map pixels). The number in the lower right corner of each pixel box represents the unique pixel label.

The number in the center of each pixel box represents the cluster ID label. The circled number in the upper left corner represents the total number of pixels in a cluster (stored at the head pixel of each cluster), *psum* and *csum* in the upper right corner denote the sum of squared x- and y-coordinate differences computed for each cluster member (termed the partial sum of squares) and for the entire cluster, respectively, and *rad* in the lower left corner represents the mean squared radius computed for each cluster (stored at the head pixel of each cluster).

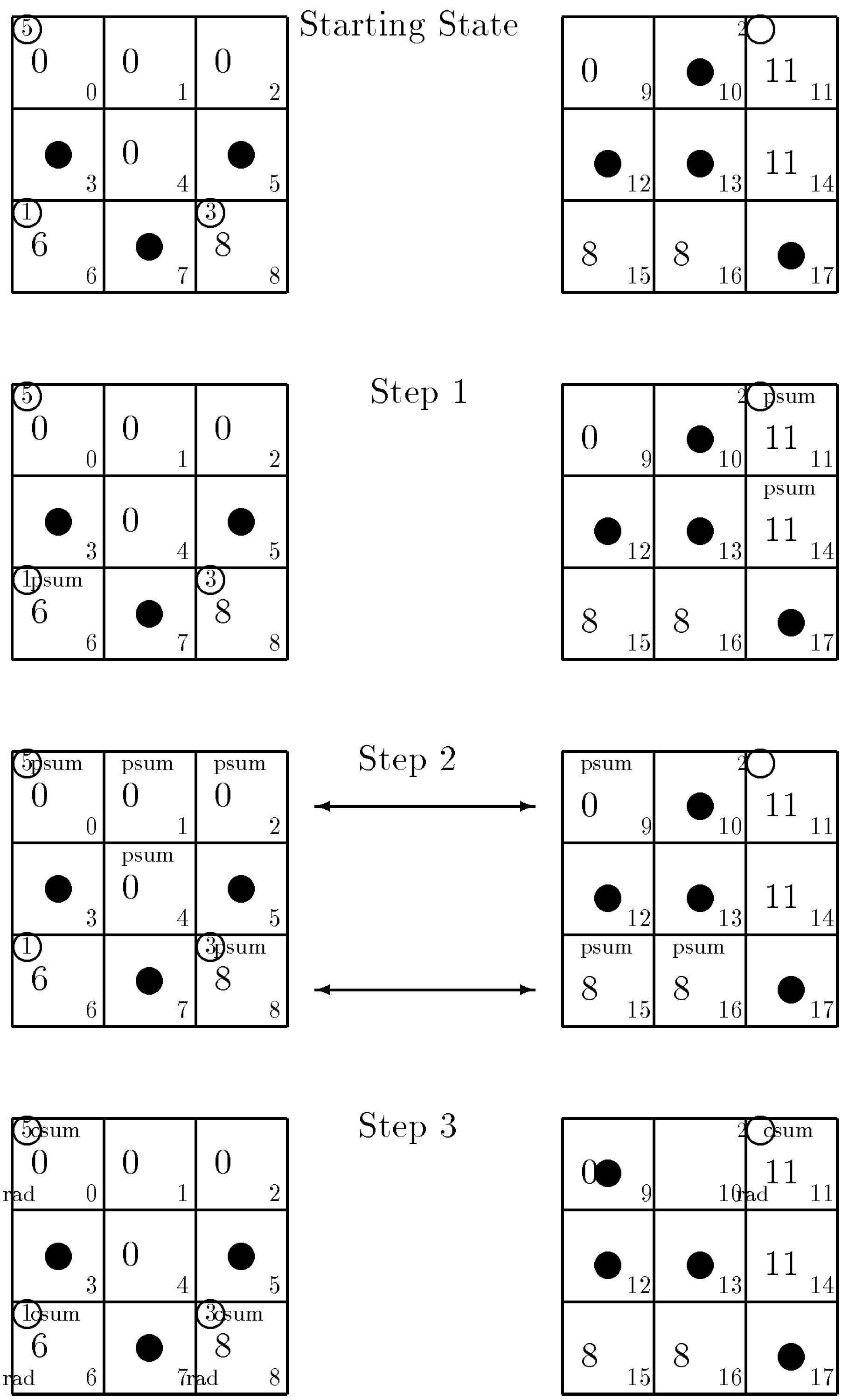


Figure 6.4: Step-wise procedure for cluster labeling in hierarchically mapped MPL implementation of mean squared radius computation.

Step 1: Resolve small clusters. Resolution of small clusters is implemented differently for the two mapping strategies.

- For the algorithm implementing cut-and-stack mapping, local cluster resolution (i.e., within a PE) is not effective, particularly for small clusters, since pixels on the same processor are not adjacent to each other. Instead, radius sums for clusters contained within one data *page* are computed, using an `xnet`-shift operation which allows each pixel on a page to view the value of every other pixel on that page. Data relevant to mean squared radius computations (e.g., cluster labels and x- and y-coordinates) for members of clusters found on more than one page are then copied to *singular* variables readable by all processors (see Chapter 2) for global comparisons in Step 2.
- For the hierarchical data mapping version, radius sums for clusters wholly contained in the sub-grid of a single PE are calculated on each PE using a local serial algorithm, and collected locally (in parallel) before global comparisons are made in Step 2. Data relevant to mean squared radius computation for only those pixels belonging to clusters represented on multiple PEs are then copied to singular variables for global comparison. This is very efficient for large, sparse maps ($p = 0.10$) because relatively few comparisons of x- and y-coordinates across PE boundaries are required to resolve clusters with hierarchical virtualization.

Step2: Resolve large clusters. Large clusters are resolved using shared variables readable by all processors. Let the map element e be an unresolved pixel whose relevant data values have been copied to shared data space. Since data are virtualized on each processor, each PE's data arrays must be searched for members of the same cluster to which e belongs. For every other element belonging to the same cluster, the absolute differences between its x-and y-coordinates and those of e are calculated (in parallel), and these differences are squared and summed for each cluster member. The distance from element a to element b is calculated only if the equivalent distance from b to a has not been calculated. If the element e is a member of layer i, the squared difference is calculated in layer i only for those cluster members whose original label is larger than that of e. For all other data layers, duplicate calculations are avoided (without a condition (if) test before each set of calculations) by searching only data layers i to n on each processor, where n = number of layers. For maps with a large dominating cluster ($p > 0.59$), this results in fairly effective load balancing, since many processors have members of the dominant cluster in their data arrays for both cut-and-stack and hierarchical virtualization strategies.

Step 3: Collect sums. Partial sums of squared differences are then summed across members (*csum* in Figure 6.4) within each cluster and stored at the head element of each cluster (the member whose original cluster label was adopted for all cluster members). The radius for each cluster (*rad* in Figure 6.4) is computed by taking the square root of this sum divided by the number of cluster members.

When pixels in the map class being analyzed are very sparse relative to the map as a whole, many processors will be intermittently idle during the traversal of the virtualized data layers during R^2 computation because they have been allocated few or no pixels belonging to the pertinent map class. This density problem is exacerbated by the need to include map pixels outside the study area. If a significant number of processors are idle, load balancing among processors, and hence performance, will be less than optimal. To enhance performance, a variation of the cut-and-stack algorithm was implemented which is more efficient when maps are very sparse ($p < 0.20$). Arrays of cluster labels are *compacted* on each processor before R^2 is computed by moving array elements associated with the study area forward in the processor's *label* array. This effectively eliminates array elements corresponding to pixels outside the study area or pixels in a different map class. In this approach, the length of the data arrays at each PE may differ, but there are fewer total active layers to traverse and more processors are active in each layer traversed. Original x-and y-coordinates for each map class member must be recorded in similarly compacted arrays, to preserve locality so that inter-pixel position differences may be calculated. This approach could also be implemented for hierarchically mapped data, but the advantage gained would not be as great as with cut-and-stack mapping, since study area pixels are more unevenly distributed (as subgrids of the landscape map) among processors with hierarchical mapping. For example, compaction of a 50-element array could result in some processors having 50 elements, while others might have none.

6.3 Results

For both parallel R^2 implementation on the MasPar MP-2, total elapsed wall-clock time was compared with that of the optimized sequential C program on a Sun SPARCstation IPX on both random maps and landscape maps extracted from runs of the NOYELP model. For these comparisons, total time is the sum of times for reading from a binary file, performing cluster identification, and computing R^2 for all clusters. Total time is dominated by R^2 computation. Speed improvements are calculated by dividing total elapsed wall-clock time of the serial program by that of the parallel implementation. Tables C.4 through C.6 in Appendix C list actual wall-clock times for both MasPar MP-2 versions and for the sequential C version.

As shown in Table 6.1, both parallel implementations show significant speed improvements over the sequential C program on a Sun SPARCstation IPX for random maps of all sizes and p values tested, performing total map analysis over 150 times faster than the serial algorithm for 512×512 random maps with $p \geq 0.85$. The se-

Table 6.1: Speed improvement of MasPar MP-2 versions over the sequential C version on a SPARC-station IPX for total map analysis, including mean squared radius computation.

Mapping strategy	p-value	Map Size 64	128	256	512
Hierarchical	0.10	0.88	4.22	21.16	84.40
	0.30	2.00	8.38	30.82	91.60
	0.62	9.88	23.97	71.24	101.17
	0.85	22.89	77.98	130.38	153.89
	1.00	40.09	97.46	154.89	175.92
Cut-and-stack	0.10	1.03	4.43	20.38	62.55
	0.30	1.81	8.58	41.53	148.23
	0.62	9.20	24.52	73.72	105.70
	0.85	20.76	72.64	126.49	152.31
	1.00	35.40	90.76	150.68	174.14

quential C program required over 16 hours of elapsed wall-clock time to analyze the 512×512 random map with $p = 1.00$ (including read time, cluster identification and geometry), while both parallel kernels resolved this same map in less than 6 minutes. Figure 6.5 presents these speed improvements for the 512×512 maps in bar graph form.

Speed improvements increased with map size and density, and were generally consistent for both parallel implementations. For the largest sparse map tested (map size of 512×512 with $p = 0.10$), the hierarchical implementation performed considerably better than the cut-and-stack version. This was attributable to the abundance of small clusters which can be evaluated locally on individual PEs (i.e., the ratio of cluster size to number of pixels per PE is small), and the dispersal of these same contiguous map elements across PEs in the cut-and-stack virtualization scheme for these large maps (with the consequent need for inter-processor communication). The cut-and-stack version is clearly more efficient for maps larger than 128×128 with $p = 0.30$. These are maps with the maximum number of clusters. These clusters typically have fewer than 50 members, but are large enough to overlap PEs when hierarchical mapping is used. For these maps, the relative efficiency of the cut-and-stack algorithm increased with increasing map size. Otherwise, performance of the two parallel implementations is generally comparable.

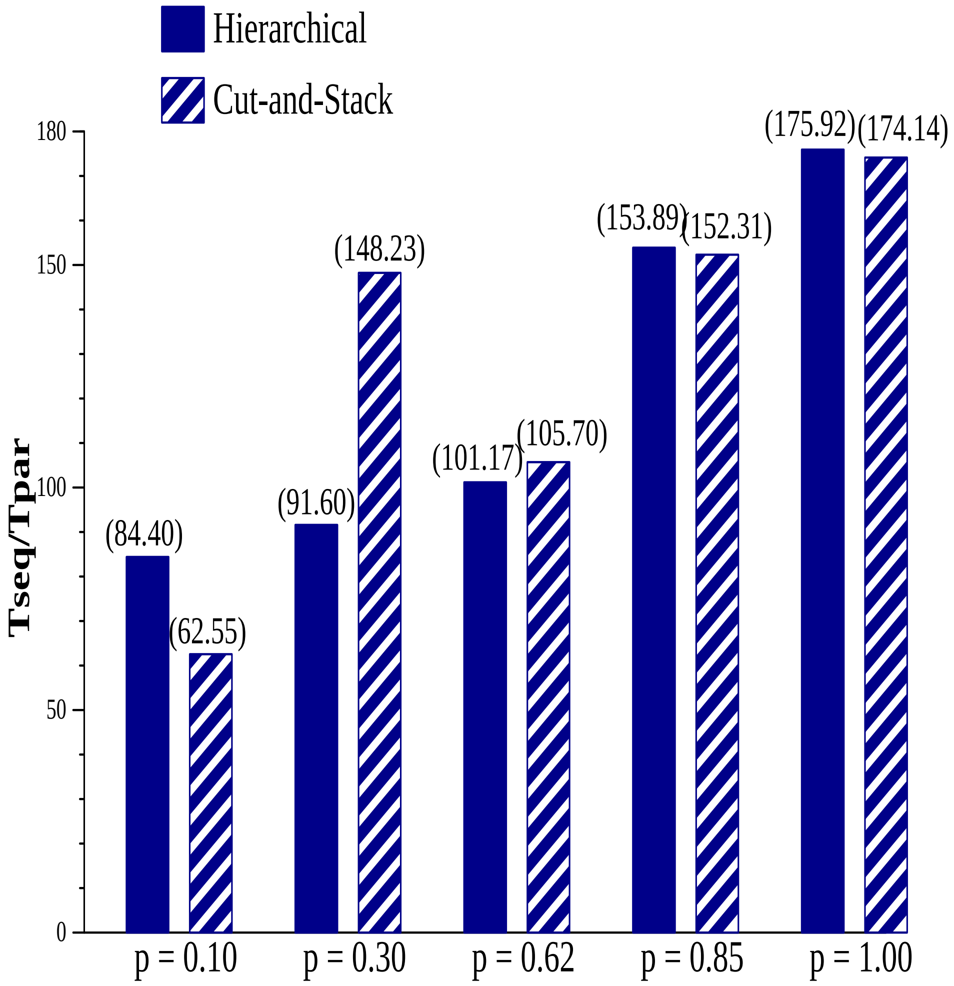


Figure 6.5: Speed improvements of MasPar MP-2 parallel implementations for total map analysis over the sequential C version on Sun SPARCstation IPX for 512×512 random maps.

Figure 6.6 shows in detail how the parallel kernels perform total map analysis relative to the serial C version over a range of p values for random maps of size 128×128. Figure 6.7 shows how the same parallel kernels perform total map analysis relative to each other over a range of p values for maps of size 512×512. Data for these graphs were produced by MPL programs using the MPIPL routine `mpigenrand()` to generate and distribute appropriate random values on each PE. Execution times (express in log units) for the serial program increase dramatically at densities near the $p = 0.59$ threshold (Figure 6.6), in response to the abrupt increase in maximum cluster size, while both parallel implementations show a relatively smooth, gradual increase (in log scale) across the range of p values.

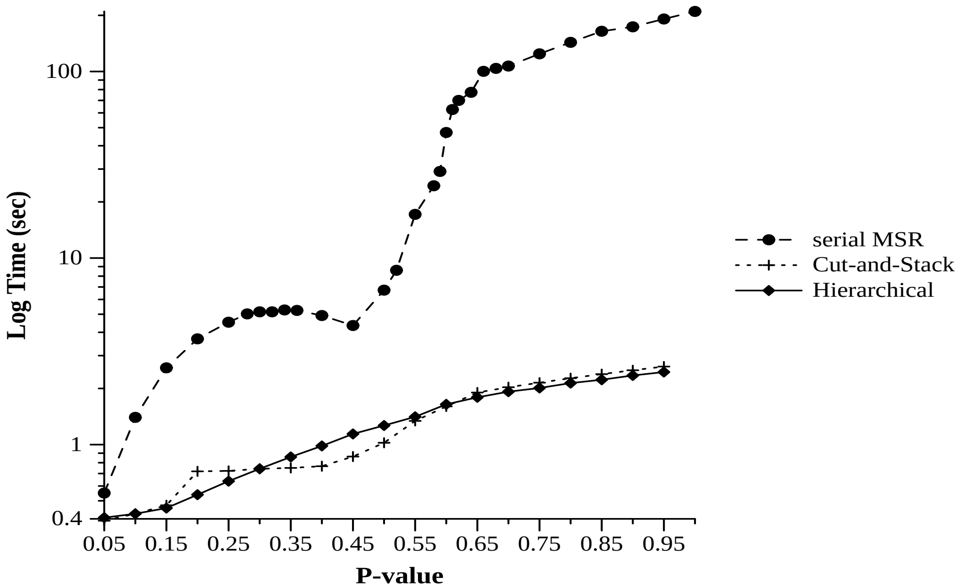


Figure 6.6: Elapsed wall-clock times (in seconds) for total map analysis (map generation, cluster identification and mean squared radius computation) across p-values for 128×128 maps for MasPar MP-2 parallel implementations and for the serial C program on a Sun SPARCstation IPX (log scale).

Figure 6.7 provides a clearer contrast for trends in total map analysis time in the vicinity of the percolation threshold for the two parallel kernels. While both implementations show distinct increasing trends, the increase for the hierarchical implementation begins at lower p values (i.e., $p = 0.30$) and is much more gradual than the trend for the cut-and-stack implementation, which increases sharply in the vicinity of the percolation threshold (i.e., 0.59). The cut-and-stack version is more efficient in the range $0.30 \leq p \leq 0.60$ because cluster sizes characteristic of maps with these densities overlap hierarchically-mapped PE subgrids, but can be resolved within cut-and-stack *pages*. When maximum cluster size increases near the percolation threshold, resolution of the largest cluster, which now overlaps page boundaries, dominates

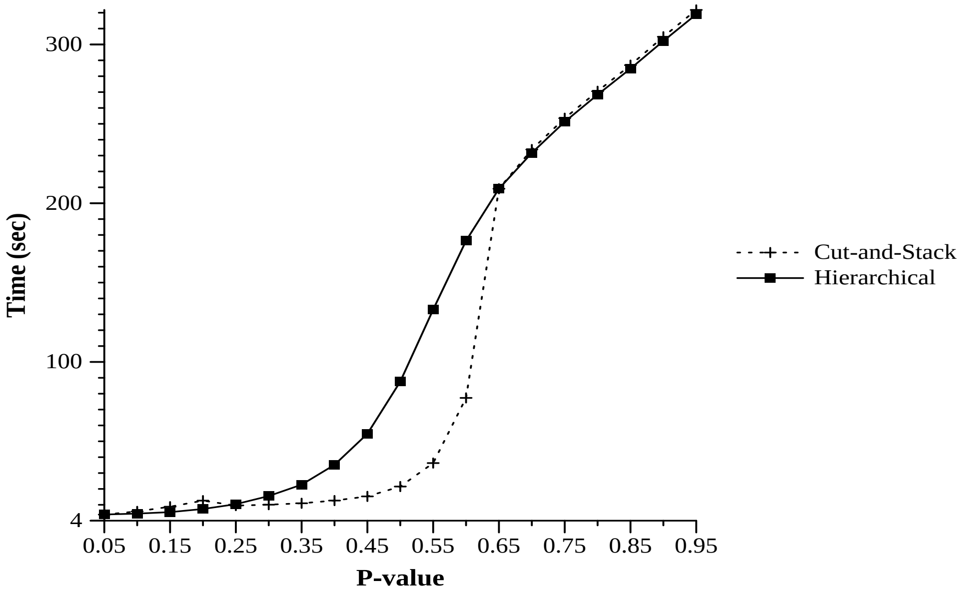


Figure 6.7: Elapsed wall-clock times (in seconds) for total map analysis across p-values for 512×512 random maps for MasPar MP-2 parallel implementations.

computing time, and performance is similar for the two mapping strategies.

As the size of a cluster approaches the size of its respective map class, possible cluster configurations become limited. Consequently, at high densities cluster radius becomes a less informative measure of cluster geometry compared to lower p values. With adequate planning, many applications could probably limit R^2 computation to clusters whose sizes are below a specific percentage of total map size. However, it is recognized that in other applications it may be desirable to compute the maximum radius possible for a given map class.

There exists a family of MasPar conversion functions (see Appendix A) which allow data configurations to be changed within a program. If alternate data mapping strategies are more efficient for different parts of a program, the programmer can switch between cut-and-stack and hierarchical mapping as needed.

6.4 Performance of MasPar MP-2 algorithms on NOYELP maps

Since the parallel kernels were developed to handle maps of varying sizes and densities, they proved successful in the analysis of maps extracted from runs of the NOYELP model. Table 6.2 compares performance of the MPL implementations for total map analysis, including R^2, with that of the serial C program on a Sun SPARCstation

IPX for 285 × 584 maps of available biomass extracted from NOYELP model runs for seven days of the 180-day cycle (see Figures 5.9 through 5.12 in Chapter 5). Maps were enlarged to 320 × 640 to make row and column dimensions divisible by 64, the dimensions of the MasPar MP-2 PE grid, contributing to the low p values for these maps (≤ 0.30).

Speed improvements for the cut-and-stack MPL version are consistently higher than those for the hierarchical version (Table 6.2) due to the more even distribution of study-area pixels across processors with cut-and-stack mapping. As was the case with random maps, speed improvements for both parallel algorithms increase with increasing density (Table 6.1).

Table 6.2: Comparison of wall-clock times for total map analysis on 285×584 resource maps extracted from NOYELP model runs for parallel MP-2 implementations and optimized serial program on a Sun SPARCstation IPX (all times are in seconds).

			Size of		Parallel			
Time	Resource		largest	Serial	Cut-and-stack		Hierarchical	
Step	Level	p-value	cluster	MSR	time	*S.I.**	time	*S.I.**
1	any	0.30	55554	2730.40	54.95	49.69	61.31	44.54
1	high	0.27	51035	2300.42	49.28	46.68	56.71	40.57
30	any	0.30	55554	2713.92	54.95	49.39	61.31	44.27
30	high	0.27	50145	2218.71	47.43	46.77	56.44	39.31
60	any	0.29	53800	2546.09	53.69	47.42	59.53	42.77
60	high	0.22	29945	829.93	28.06	29.58	44.49	18.66
90	any	0.18	16515	299.83	18.85	15.90	36.91	8.12
90	high	0.04	4701	24.99	6.49	3.85	10.79	2.32
120	any	0.06	6158	49.49	8.15	6.08	14.07	3.52
120	high	<0.01	394	1.96	2.73	0.72	3.00	0.65
150	any	0.03	6153	49.26	8.14	6.05	14.04	3.51
150	high	<0.01	0	1.02	0.12	8.72	2.55	0.40
180	any	0.29	55190	2674.64	54.51	49.07	60.93	43.90
180	high	0.14	4672	94.08	11.97	7.86	27.99	3.36

*S.I. denotes speed improvement of parallel implementation over serial program.

6.5 Conclusions

MasPar MP-2 data parallel implementations of total map analysis, which is dominated by R^2 computation, show substantial speed improvements over serial implementations (over 150 for large, dense random maps), making radius measurements a viable tool for many ecological applications. Speed improvements increased with map size and density for both kernels. Elapsed wall-clock times for data-parallel implementations do not increase as dramatically as those for the serial implementation at densities above the percolation threshold. Maximum speed improvements for landscape maps tested were lower (<50), due largely to the high percentage (62%) of map pixels outside the study area. These non-habitat pixels contributed substantially to the low p values for the NOYELP maps. As with cluster identification, size of the largest cluster (relative to map size) is a better indicator of performance than p value when comparing random maps with landscape maps. For the limited range of p values represented by NOYELP maps (<0.30), speed improvements increased with increasing density, a trend consistent with results for random maps.

The parallel implementations discussed in this chapter have advantages over the original serial versions other than speed. Data size is constant for a given map size in the parallel implementations, regardless of map density. No arrays which are dependent on the number or size of clusters in the map being analyzed are required. While the revised serial R^2 program does not maintain arrays of coordinate differences for each cluster element for large clusters, an array which records the size of each cluster is maintained.

As is the case with cluster identification kernels presented in Chapter 5, parallel kernels for R^2 computation could run on the DPU of the MasPar MP-2 and be called by a serial program running on the front-end DECstation 5000-200 machine. Results could either be returned to the calling program or be written to an output file.

Chapter 7

Animal Movement

The algorithm evaluation and development activities for animal movement for this thesis are specific to the NOYELP model. The initial objective was to duplicate the serial NOYELP movement rule on the MasPar MP-2. In the process of attempting to address this objective, revisions were made to the serial NOYELP program which improved its performance significantly (with no change in functionality). This revised serial program then provided the basis for parallel model implementation efforts. The following sections describe both serial and parallel efforts, including the initial scoping of a revised movement rule that is more amenable to parallel processing.

7.1 Serial NOYELP Implementation

The serial NOYELP model is implemented by a 4736 line Fortran-77 program consisting of 11 subroutines. Figure 7.1 illustrates the flow of program control through the NOYELP subroutines. A summary of these subroutines is provided in Appendix B.

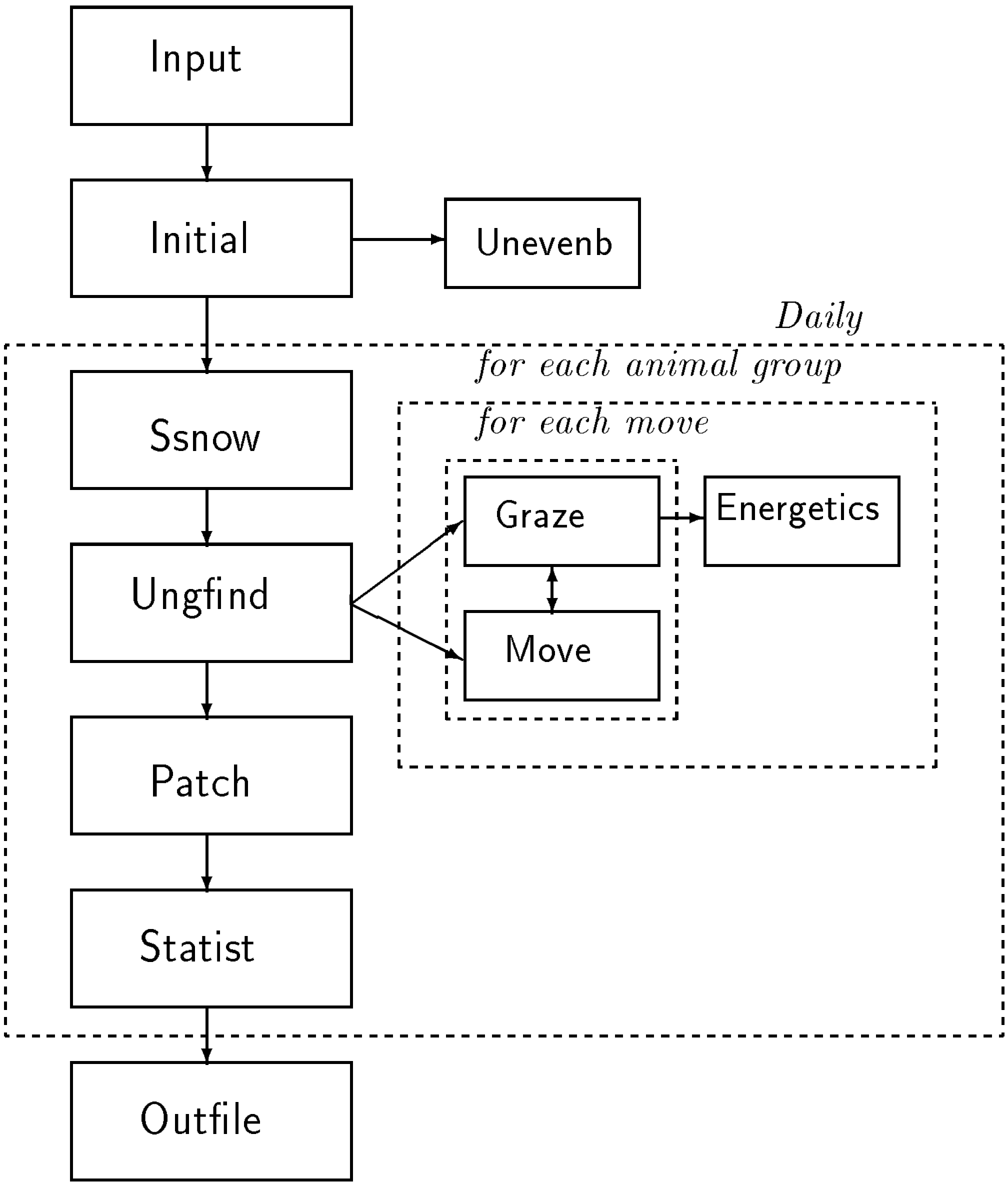


Figure 7.1: Control flow chart for the NOYELP model.

Abiotic data are input from files in subroutine INPUT; ungulate locations and biomass are initialized on the landscape in subroutine `initial`; snow conditions are updated every 3 days in subroutine SSNOW. At the heart of the NOYELP model are four subroutines which simulate the movement and grazing of animal groups on the landscape: `ungfind`, `move`, `graze`, and `energetics`. On each day of the 180-day model cycle, `ungfind` locates each animal group on the landscape and initiates a sequence of search/move/graze activities for each group which results in animal movement, biomass depletion due to foraging, and changes in ungulate body weight. These four subroutines are the central focus of the efforts discussed in this chapter. In subroutine `patch`, cluster analysis (see Chapter 5) is performed daily on the available biomass matrix, with results stored for output to a file (in subroutine `outfile`).

7.2 Model Description

In the following discussion *habitat pixel* refers to a pixel within the boundaries of the study area, animal *category* refers to one of the six types of animals (elk/bison calf, cow, or bull) and animal *group* refers to a single group of 2–9 elk or bison, as defined in Chapter 3. The input data set used in this model evaluation effort included 19,270 elk and 699 bison, forming 5015 ungulate groups. *Available biomass* refers to that part of total biomass which is available for consumption by animals. A *resource* pixel is a pixel containing available biomass (i.e., biomass above the preset refuge level and not rendered unavailable due to snow cover). The *movement rule* defines a set of constraints which govern the nature and sequence of events in the move-graze component of the model (i.e., subroutines `ungfind, move, graze`, and `energetics`). The *movement series* refers to the order in which ungulate groups move and forage on a given day, and is constant for all model days. Group i in the series makes all its moves for a day before subsequent groups in the series move (where $0 \leq i \leq n$, where n is the total number of groups in the model). This ordering of animal groups is established randomly when groups are initially distributed on the landscape (in subroutine `initial`). See Chapter 3 for a more detailed discussion of the NOYELP model, including the input ungulate data set.

Subroutine `ungfind` comprises a loop over each time step (day) for all animal groups in sequence, according to the movement series. For each group i, the amount of energy required to move one unit of distance (i.e., pixel) across the landscape and maximum moving distance for the day (`maxdist`) are calculated. The value of `maxdist` is governed largely by snow depth and density, which are updated every three days in the model cycle. Following the NOYELP movement rule, the pixel in which group i finished the previous day is first evaluated to determine if available biomass exceeds the threshold biomass. If sufficient resources are available on group i's current pixel at the start of the day, the group grazes before moving to a new site; otherwise, the day starts with a call to `move`. Since an ungulate group is not allowed to choose the same pixel twice in any day, every other grazing event except the last of the day is followed by a call to move.

In subroutine `move`, group i's first move of each day is preceded by the calculation of the group's search area boundaries, which are based on `maxdist`. Figure 7.2 shows a diagram of a 9×9 search area (`maxdist` $= 4$) in the context of the larger work map and NOYELP data map. All movements for group i over the course of the day are

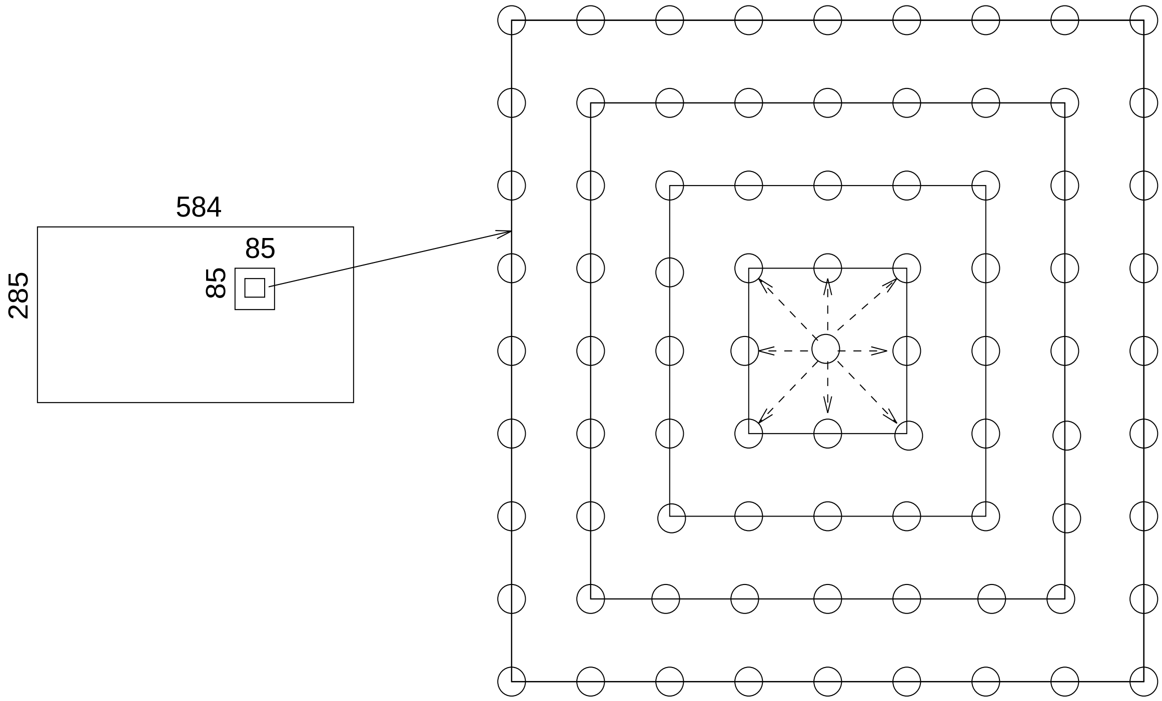


Figure 7.2: Typical search area for a NOYELP animal group, with maximum moving distance of 4 pixels.

restricted to this search area, as defined by `maxdist`.

For each habitat pixel in the search area, available biomass is calculated once per day using a complex 18-variable formula. Group-specific available biomass values are updated prior to each group's moves to reflect the grazing activities of all previous groups in the movement series. To implement a move, resource pixels along the perimeter of concentric squares of dimension $k \times k$ are searched in a stepwise process (where $k = d * 2 + 1$, $1 \leq d \leq$ `maxdist`, and d is the distance away from the current pixel location). For each k, a comparison of available biomass values is made for all perimeter pixels. If resource pixels are found, the search stops, and a move is made to the pixel with the largest available biomass. If multiple pixels have the same available biomass level, one pixel is chosen at random from among them. Once a destination pixel is selected, it becomes the center of a new search area (if another move is indicated). The size of this and any subsequent search areas for this group on the current day are constrained by the boundaries of the overall search area. The process of searching for resource pixels along the perimeter of concentric squares continues. If there are no resource pixels within the entire search area (i.e., d=`maxdist`), group i chooses at random from habitat pixels `maxdist` away (the perimeter pixels of the outermost search square) and starts the next day at that pixel.

Subroutine `graze` is called after a destination resource pixel has been chosen. When group i grazes on the available biomass at the destination pixel, total biomass at that pixel is decreased by an amount determined by the ungulate body weight for group i and the daily foraging rate. This updated total biomass value is used by the next animal group in the series (if search areas overlap) for calculation of available biomass at that pixel.

Group i continues to search, move, and graze until one of two limiting conditions is met: maximum daily forage is consumed or maximum daily moving distance is traveled. Because it is not possible for an animal to consume its maximum daily forage at one resource pixel, even when resources are plentiful, and because the group is prohibited from visiting a pixel twice in the same day, each group must make at least one move per day.

After the last graze of the day for group i, the daily energy balance is calculated and body weight is adjusted downward for group members if the energy balance is negative (no weight gain is permitted). If body weight falls below a pre-assigned percentage of initial weight, mortality is simulated and the group is removed from the landscape. These computations are included in the `energetics` subroutine.

This process of searching, moving, grazing and energy adjustment is repeated for each ungulate group according to the movement series until all groups have completed moving and grazing for the day. At the end of each day, daily statistics are calculated, and cluster analysis is performed in subroutine `patch` for two levels of available biomass values (any resource and high resource).

7.3 Program Evaluation

7.3.1 Introduction

Computing time for the original serial NOYELP model is dominated by the simulation of animal movement, which accounts for 95% of computing time when the maximum number of 20,000 animals (composing 5015 groups) is input. Most of this time is consumed in the calculation of group-specific available biomass values on which animal groups base their movement decisions. These intensive computations are repeated in a program loop for each animal group for all pixels in its search area at each time step. This situation is typical of those for which parallel processing is well-suited, offering the potential for substantial speed improvements comparable to those for the computationally-intensive mean squared radius calculation (see Chapter 6).

Three model constraints related to the NOYELP movement rule make recalculation of available biomass values in search area pixels for each of the 5015 animal groups on a daily basis seem unavoidable:

1. One of the eighteen variables involved in the calculation of available biomass, `swhi`, which defines upper bounds for snow/water equivalent above which an animal cannot forage, is (ungulate) category-specific. As a result, each of the 6 animal categories (elk calf/cow/bull, bison calf/cow/bull) " sees" a different available biomass value at each pixel in the study area. Unfortunately, the value for `swhi` does not relate to the other variables in a scalar manner. As a result, removing these calculations from the loop would entail creating six separate available biomass maps (one for each category) and updating all maps after each group grazes. The extra work this would require would offset any time saved by removing the calculations from the loop, and could pose serious memory constraints (i.e., in storing six maps encompassing the entire study area).

2. According to the movement rule, each ungulate group must be aware of the results of resource depletion resulting from the grazing of previous ungulate groups in the movement series. Knowledge-based foraging requires that total biomass changes be recorded after each animal grazes, and available biomass values be recalculated (from total biomass values) within the loop.

3. Each animal group must set its daily foraging path to 0 (i.e., set the category-specific available biomass for each pixel visited to 0) to prevent multiple visits to the same pixel by the same animal on the same day. In the original version of NOYELP, a group can set its path to 0 on the available biomass map without affecting subsequent groups, because the next group will recalculate the values it requires from (updated) total biomass values. Removing available biomass calculations from the loop would necessitate setting flags in a separate array and checking these flags before each resource pixel comparison to see whether the current group has used the current pixel on a previous move on the same day. Again, the extra work required would probably offset any time saved by removing calculations from the loop.

7.3.2 Modification of the serial algorithm

In the process of designing a parallel movement algorithm, a detailed evaluation of the interrelations among the different parts of the movement rule was made. This evaluation led to the identification of several areas in which the original serial code could be improved. Alternative approaches were developed, implemented and tested to (1) map search area resource values to (smaller) work matrices, and (2) provide current available biomass knowledge to moving animals. Incorporation of these approaches into the NOYELP program improved serial performance significantly. The following changes were made to the serial algorithm without any change in functionality (i.e., all model outputs are identical).

1. Work maps the size of the maximum search area for any NOYELP ungulate group (i.e., 85×85) are used for storing category-specific available biomass computations. Indices of the work matrix are mapped to corresponding indices of the larger habitat map. Determination of the maximum resource value, the search for duplicate maximum values, and random selection of the destination pixel from duplicate maxima can be made directly from the work map. Biomass pixels are calculated iteratively within the work map as the distance d from the home pixel increases to `maxdist`. If a group finds a resource pixel at distance d, available biomass values in concentric squares n units away (where d<n<maxdist) will not have to be calculated. This differs from the original NOYELP program, in which biomass values for the entire search area ($(2*maxdist+1)$ by $(2*maxdist+1)$) are calculated before any searching begins.

2. At the start of each day, a partial computation of available biomass is made, up to the point at which category-specific information (`swhi`) is required. These partial available biomass values are stored in a study area-sized matrix (285×584), referred to below as the *shared map*, for use by all groups in the daily sequential calculation of category-specific available biomass. Within the daily movement loop, as each group makes available biomass comparisons for selecting a destination pixel, the partial available biomass value is read (from the shared map) and combined with the appropriate `swhi` value to calculate category-specific available biomass on a pixel by pixel basis. The available biomass values are then stored in a work map.

3. When a group grazes at a resource pixel, total biomass depletion is calculated for that pixel. Partial values for available biomass are then recalculated on the shared map for just the one pixel grazed. In preparation for the next ungulate group's search for a destination pixel, these updated values are read from the shared map, and new available biomass values are calculated on the work map using category-specific information for the next group in the movement series.

4. Each animal group marks its movement path on the work map with flags (i.e., available biomass is set to an out-of-range value). This prevents the revisiting of pixels on the same day without affecting the shared map which is read by all groups. As a group makes its moves for the day, the work-map coordinates of these moves are stored in an array, so that the flags which have been set to

Table 7.1: CPU time (in seconds) for the original and revised NOYELP serial model versions and speed improvement of the revised over the original version.

Version	DECstation 5000-200	Sun SPARCstation 2
ORIGINAL	12780.80	9174.25
REVISED	828.15	840.72
Speed improvement	15.43	10.91

mark one group's path can be selectively removed before the next group uses the map. The use of these flags and associated arrays is a crucial step in redesigning available biomass calculations, because it eliminates the need to clear the entire work map (i.e., set all elements to 0).

These modifications are interrelated, and full performance enhancement requires all four changes. Other minor changes were made to the serial code to facilitate the implementation of these modifications.

7.3.3 Comparison of performance of original and revised NOYELP model

The original and revised NOYELP models were compared for performance, based on speed improvement (i.e., the ratio of CPU time for the original model to that of the revised version). Both Fortran-77 programs were compiled with the `-O` option for the current `f77` compiler. The results for total CPU time are shown in Table 7.1. Figure 7.3 compares CPU time of the original and revised NOYELP models at each time step of the 180-day model cycle and Figure 7.4 shows these same CPU times for the revised model on an expanded y-axis.

Speed improvements of 10.91 and 15.43 were realized on a Sun SPARCstation 2 and DECstation 5000 model 200, respectively (Table 7.1). Most of this improvement was attributable to changes in the movement component of the model. Trends of CPU time over the 180-day model cycle for the original and revised versions were very different (Figure 7.3). The original version required more CPU time during the early part and at the very end of the NOYELP cycle, while CPU times for the revised model increased slowly over the first 120 days, were highest from day 120 to day 165, and decreased sharply during the last 15 days (Figure 7.4). The period during which times were fastest for the original model coincided with the period when times were slowest for the revised model.

The number or distance of moves made and distance traveled per day has little effect on execution time of the original model, since all available biomass values within a group's search boundary are calculated each day, regardless of number of moves, and biomass calculations consume the majority of CPU time. Instead, the major

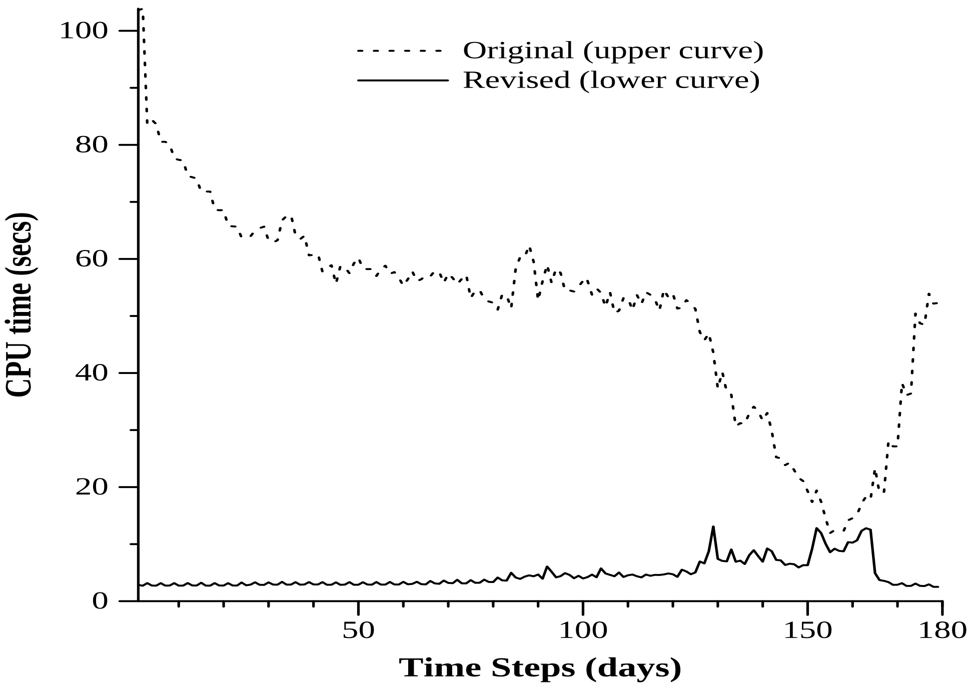


Figure 7.3: CPU time (in seconds) at each time step for the original and revised NOYELP model over the 180-day model cycle on a Sun SPARCstation 2 (excluding I/O).

factor influencing execution time for the original version is the availability of biomass on the NOYELP landscape (Figure 7.5). When components of available biomass are low, category-specific calculations for available biomass are truncated, requiring less execution time. The increasing availability of biomass in late winter–early spring (Figure 7.5) requires increasing numbers of biomass computations in the original NOYELP program, and computing time increases substantially from the mid-winter low (Figure 7.3).

In contrast, CPU times for the revised version (Figures 7.3 and 7.4) vary directly with the total daily distances traveled in ungulate movements (see Figure 7.6). Because biomass values are computed sequentially as the search area expands, both number of moves and the length of each move increases the required number of resource pixel comparisons and hence the number of available biomass calculations that must be made. Execution time is highest for time steps in which resources are sparse and many moves are required per day to satisfy daily foraging requirements. As snow cover melts in late winter and biomass becomes more available, CPU times decrease sharply as the average number of moves per day for the ungulate groups decreases.

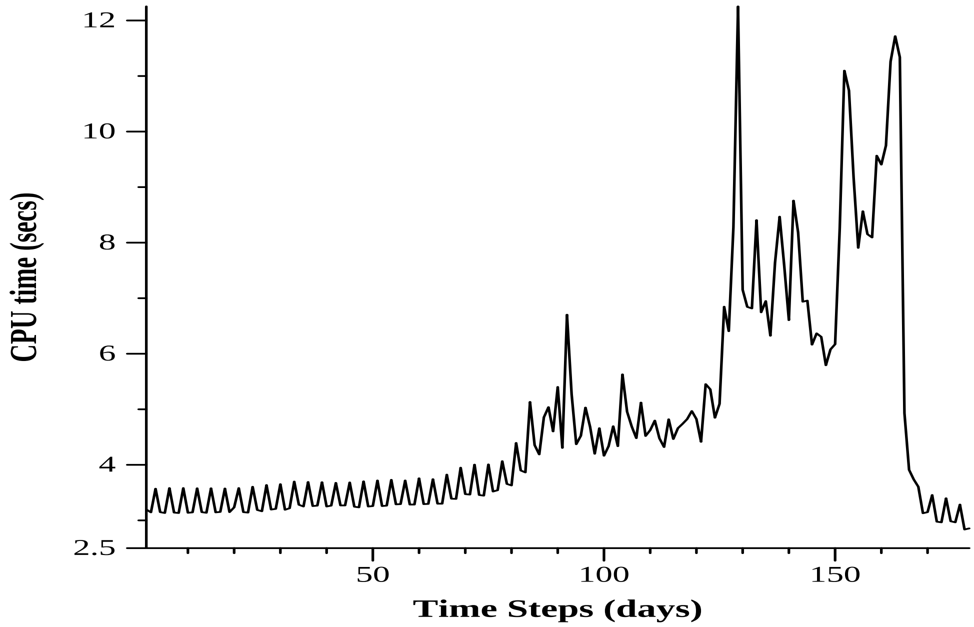


Figure 7.4: CPU time (in seconds) for each time step of the revised NOYELP model over the 180-day model cycle.

Performance gains from the truncation of biomass calculations are much smaller for the revised model because fewer biomass calculations are made.

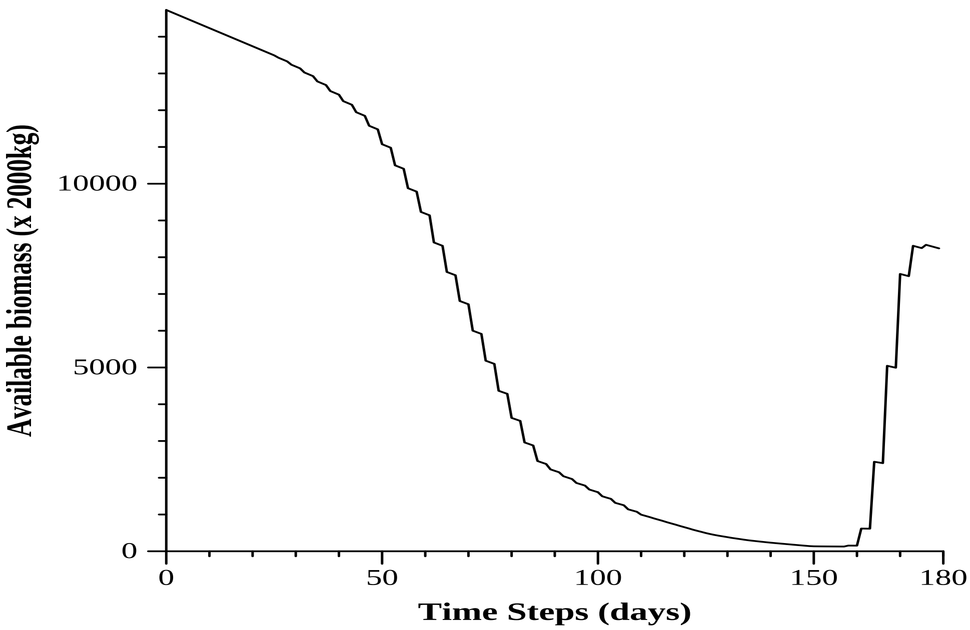


Figure 7.5: Available biomass (in 2000kg units) in the study area over the 180-day NOYELP model cycle.

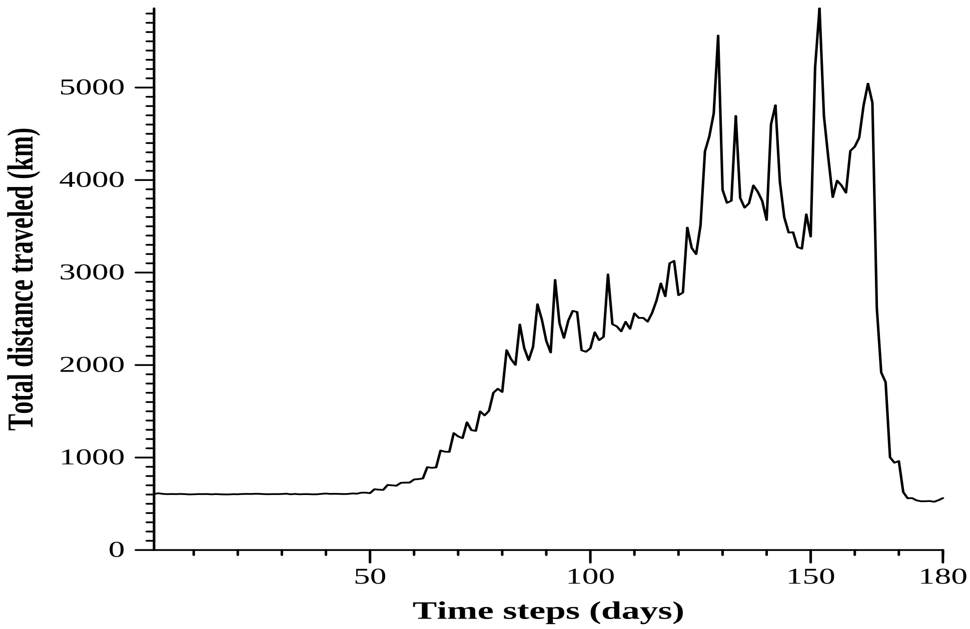


Figure 7.6: Total daily distance traveled (km/day) by all ungulate groups over the 180-day NOYELP model cycle.

7.4 Parallelization of Animal Movements

As a first direct step toward parallelizing the NOYELP animal movement rule, a data-parallel MPL version of the subroutine `move` was attempted. Cut-and-stack virtualization was chosen because of the many non-habitat pixels which surround the irregularly-shaped NYNP study area. While hierarchical virtualization clumps non-habitat pixels together on the same processors, cut-and-stack mapping allocates these non-habitat pixels more equitably among processors, leading to better load balancing. However, if an ungulate group's search area were limited to the subgrid of one processor, hierarchical mapping would have a strong advantage over cut-and-stack mapping, even with the presence of a significant number of non-habitat pixels, since no MPL communication constructs are required for comparison of pixels within a processor's subgrid.

A key issue in parallelization of animal movements in the NOYELP model is how to distribute the data for animal groups across the processors. Two basic approaches can be employed. In the first approach, group data are distributed evenly across processors, regardless of the actual location of the group on the landscape. Each group's descriptive data are stored in one stationary place, and the value of a location variable is changed when a move is made. This distribution would be advantageous if a significant amount of data were maintained for each group, or if a large number of groups reside on an individual pixels.

The second approach involves maintaining group data on the same processor and data layer as the landscape pixel on which the group resides. When groups move from one pixel to another, these data must be transferred to the appropriate destination pixel. This is feasible for NOYELP because the model maintains only three pieces of data about each animal group: category, location, and current body weight. If more information were maintained, the required amount of data movement would be prohibitive. The danger of memory constraints when multiple groups share a pixel must also be considered.

Both approaches to storing animal group data were tested in this parallelization effort. The first approach is more flexible and is recommended for a fully-implemented parallel NOYELP model. Although this approach has the disadvantage of requiring more inter-processor communication to access group data on a distant processor, its use is warranted because it can better handle the presence of multiple groups on a pixel and has the flexibility to accommodate additional group-specific data, as needed.

Execution times were measured at various stages of algorithm development, with somewhat disappointing results. Progressively simpler parallel versions of the movement rule were implemented in an attempt to identify a level at which speed improvements over the serial model could be achieved. In the most simplified version of the `move` subroutine, data for all categories of animals were read from the same available biomass map, with the median `swhi` value used for all calculations. With this simplification, all ungulate categories would see the same available resource values. The restriction from revisiting the same pixel on a given day was also removed. Unfortunately, execution time was still about 20% slower than for the fully-implemented serial model.

A parallel versions of the NOYELP movement component implementing the current movement rule would be fairly communication intensive, due to the inter-processor communication required to compare resource values from adjoining pixels within a search area for each ungulate group several times each day over a 180 day simulation period. Both the original and revised versions of NOYELP utilize nested loops to simulate these repetitive activities. To benefit from parallelization, the number of floating-point calculations performed inside these parallelizable program loops must be sufficient to support the amount of interprocessor communication necessary for parallel implementation of the movement rule chosen. The revisions made to the original serial NOYELP model decreased the computational complexity of the loops in the `move` subroutine to such an extent that inter-processor communication requirements associated with parallelization of the movement rule more than offset any gains from parallelization of computations within the program loops.

This attempt to parallelize the animal movement component of NOYELP indicates that development of a parallel version of the model will require fundamental re-conceptualization of the movement rule. The parallel movement rule would have to minimize communication/comparisons between processors to be effective in enhancing model performance. The re-conceptualized parallel model would have to be recalibrated from suitable ecological data, adjusting flexible components in an iterative process similar to the procedure used to calibrate the original serial model. The development and implementation of such a rule, including the calibration and validation required to realize an operational parallel model, are topics for future research.

One approach which appears to be feasible and potentially efficient is based on restricting of the movement of an animal group to the confines of a processor subgrid for one move. This rule would favor hierarchical over cut-and-stack mapping because the hierarchically-mapped subgrid represents a contiguous chunk of the landscape map, keeping communications costs low. At the start of the each move, animals which are on one of the border rows or columns of the subgrid could be moved to the adjoining processor sub-grid. This would allow wider-ranging movement, but would not require comparisons of pixel values across processor boundaries (which requires the use of time-consuming communication constructs). More moves per day might be required to obtain sufficient daily forage intake because of the restricted search area for each day. A single suitability index for each subgrid might be communicated to neighboring subgrids to guide direction of movements. Whether these changes to the movement rule are sufficiently consistent with ecological theory or could be compensated for by other adjustments to the model is difficult to judge at this point.

7.5 Serial vs. Parallel Updating of Biomass Levels

A major concern in designing a parallel NOYELP model would be the updating of total and available resource levels, which provides each ungulate group with knowledge of the grazing effects of animal groups which precede it in the movement series. The use of this movement series (which remains constant over the 180-day model period) is critical to the serial processing of foraging activity. Its use implies a serial process which, by definition, constrains concurrent grazing behavior. However, it is possible that sequential grazing and consequent partial knowledge of grazing effects may effectively simulate ecologically significant phenomena, such as unequal fitness and/or dominance ranking among population members ([Lomn92]).

The influences of the serial constraint on searching and foraging imposed by use of a movement series, and the consequent need for intra-move biomass updating, varies over the 180 day NOYELP cycle. Biomass updating is not particularly important in early time steps (i.e., during the autumn), when animals are widely distributed over the landscape and infrequently choose the same destination pixel (Figure 7.7). However, during the winter months when snow cover is deep and available resources become scarce, animals tend to congregate in the low-elevation Mammoth-Gardiner (snow-shadow) area and on south slopes throughout the study area where there is less snow cover ([TWWR+93]). During these periods, population densities in the more favorable habitat pixels can build to high levels, as each group chooses a shared resource pixel solely because that pixel has the highest available resource level in its search area, regardless of the number of other groups which may inhabit the same (1 hectare) pixel. Results of the NOYELP model run (Figure 7.7) indicate that from late December through mid-February up to 80 groups (with 2 to 9 members per group) share an individual pixel on a given day. Depletion of available biomass by groups that rank high in the movement series can be a significant factor in the selection of destination pixels by lower ranked groups during this period. Category-specific resource updating can be a critical factor in determining animal movement during the winter. As such, the serial constraints associated with the use of the movement series for biomass updating becomes important during this period.

Once snow melt begins and these ungulate groups disperse, the importance of biomass updating decreases, and the movement rule again becomes somewhat less constraining to ungulate movement patterns.

Allowing multiple groups to share destination pixels constitutes an important and ecologically consistent movement condition which should be retained in the parallel model. However, biomass updating, which is based on the movement series, becomes problematic when animal groups are required to make their destination choices in parallel. An exact parallel implementation of NOYELP serial biomass updating would still require the maintenance of a hierarchical ranking of animal groups (i.e., movement series) identical to that used in the serial model. The series would be used to

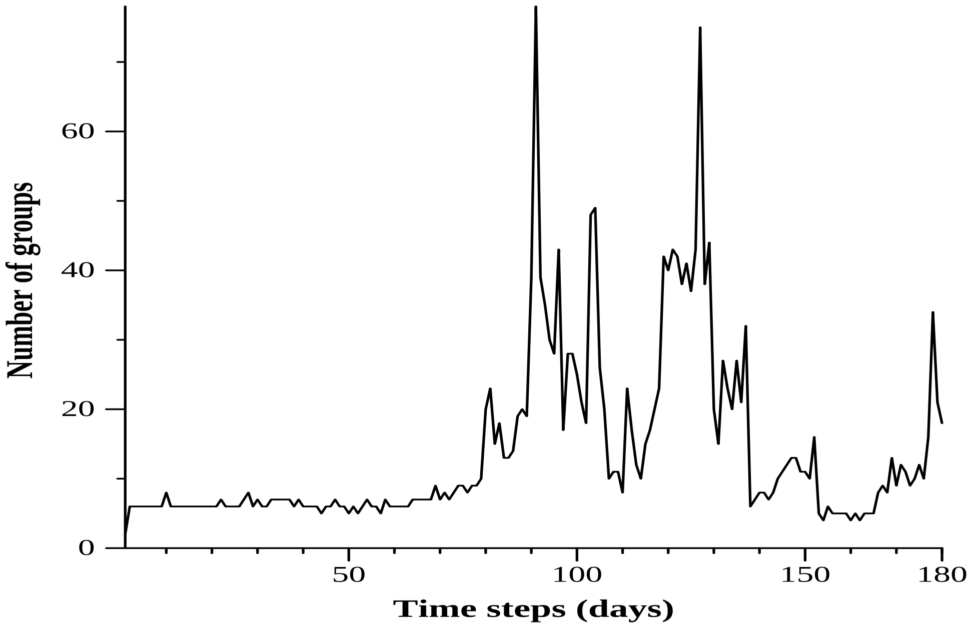


Figure 7.7: Maximum number of ungulate groups which share a pixel over the 180-day cycle of the NOYELP model.

implement a parallel-based movement algorithm that can accommodate the absence of the serial biomass updating scheme in the initial selection of destination pixels.

In this parallel movement algorithm, animal groups on all processors would make a given move as if no conflict existed. If more than one group chose the same destination pixel on the same move, a *rollback* ([BeTs89]) technique based on the movement series could be used to duplicate the serial behavior by allowing the effects of these interactions to propagate backward to previous model decisions which were based on biomass values that had not been updated to reflect the foraging activities of animal groups higher in the movement series ([Palm92]). Figure 7.8 illustrates the chain of events following the choice of the same destination pixel (pixel 9) by two animal groups (1 and 2) on the third move of the day. The highest ranking group (group 1) would be allowed to graze and biomass would be updated for its selected destination pixel. Then, the next lower ranked group which selected the same destination pixel (group 2) would *roll back* its pixel selection process by one move, reevaluating its search area to choose its new maximum resource pixel (considering grazing effects for the day by higher ranked groups, but not by lower ranked groups). If this new pixel had been previously grazed by another (lower ranked) animal group (group 3) on the same move, or on a previous move on the same day, group 2 would graze the pixel and group 3's pixel selections would be rolled back to the previous move. Group 3 would reevaluate its search area and reselect its maximum resource pixel. The rollback

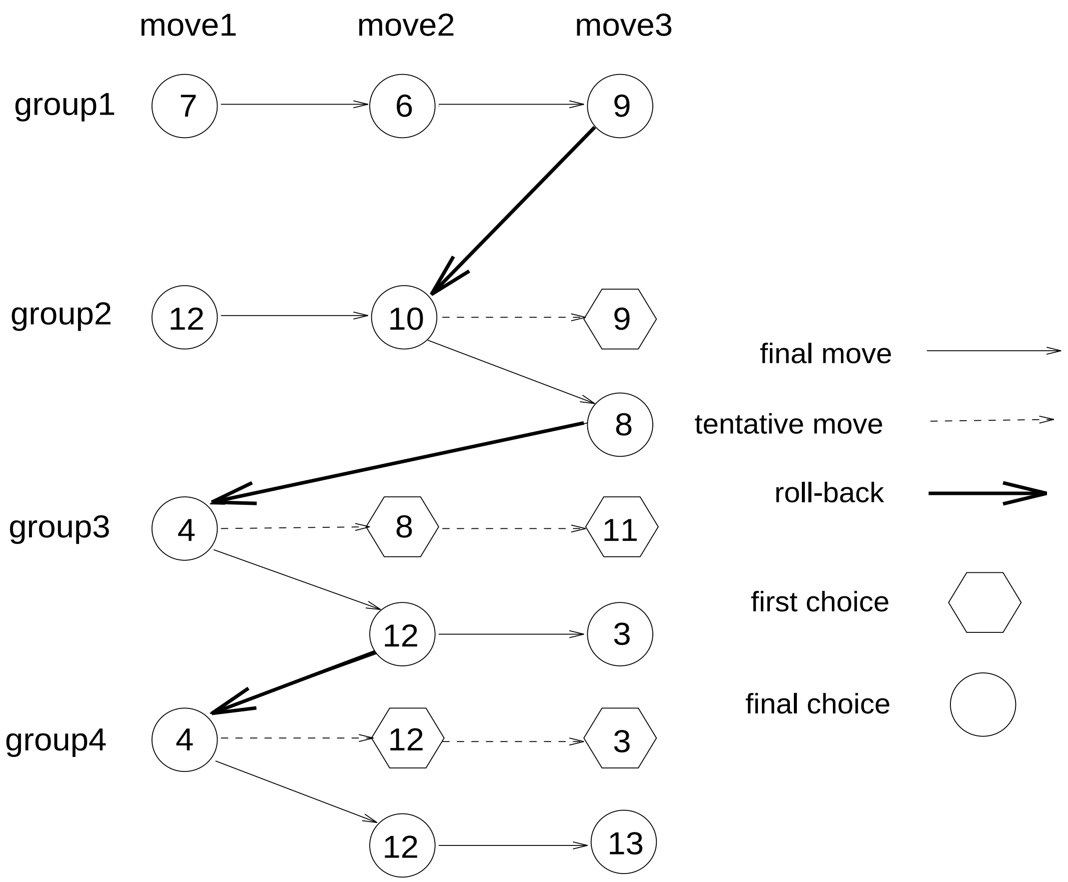


Figure 7.8: Rollback mechanism for resolving resource depletion updates in exact parallel implementation of NOYELP movement rule.

sequence would be repeated after each move of a given day until all conflicts were resolved. As is evident in comparing the first and final choices of group 4 on move 2 in Figure 7.8, a group may re-select the same pixel based on updated biomass. All biomass updates for the day would be held in stacked buffers (along with information about which group was responsible for the update) until all moves for a given day were resolved. Note in Figure 7.8 that the rollback for group 3 was two moves, even though biomass was updated after every move. Such multiple-move rollbacks are required to effectively duplicate the outputs of the serial algorithm, wherein a particular group makes all of its moves for a given day before groups lower in the movement series make any moves.

Based on this reasoning, it appears that when pixel sharing is important in the NOYELP model, as it is during winter, any parallel performance advantages would quickly be lost in the recursive roll-backs required to emulate serial updating of biomass values for making movement decisions. When the average number of groups per destination pixel is large and the number of moves made per day is high, computing time required to implement this rollback procedure could quickly become prohibitive.

If the goals of the parallelization effort were not driven by the need to duplicate serial behavior and output but were rather to develop and implement an equally meaningful parallel movement rule, these multiple-move rollbacks would not be necessary. The parallel movement rule would resolve the issue of allocation of destination pixel resources when multiple groups choose the same destination pixel in a way that avoids this time-consuming serial component. One option would be to allow the several groups on the pixel to share available biomass equally. Thus, if n groups choose a pixel with resources of R kilograms, each could calculate their biomass consumption from a base of R/n kg. An alternative option would be to allow the groups to graze the destination pixel serially, with animal groups ranked low in the series grazing from pixels whose biomass has been diminished by groups ranked higher in the series. If resources were very low, a low-ranked group might actually be shut out from grazing (as resources are depleted to the refuge level by previously grazing groups) even though alternate resource patches might be available in the search area.

A third parallelization option, one which incorporates knowledge updating, would give groups sharing resource-limited pixels another opportunity to choose a destination (from among updated pixels). However, this capability would introduce another time-consuming serial component into the move-graze function. Many processors would be idle waiting for a few groups to move and model performance would be degraded.

All of these alternatives would differ from the serial movement rule in that some ungulate groups might have initially chosen other pixels in their search area based on their knowledge of the effects of grazing by higher ranked groups (i.e., if resource updating had occurred following the movement of each group).

In a parallel version of the NOYELP model, serial processing will be necessary at several points in the simulation, regardless of the movement rule implemented, for the following reasons:

1. The number of map pixels is greater than the number of MP-2 processors, requiring each processor to handle more than one habitat pixel. As discussed in Chapter 2, with 4096 processors and 320×640 habitat pixels, 50 pixels are mapped to each processor. These pixels are stored in data arrays within each processor. Each PE can process only one pixel at a time, so the pixels are processed serially.

2. A habitat pixel may be host to more than one animal group. Each group residing on a given pixel must be processed serially in its selection of a destination pixel.

3. When more than one animal group chooses a particular destination pixel, a serial component of some sort is unavoidably introduced into the `move` and `graze` functions.

These unavoidable serial components should be recognized and exploited in parallel approaches to modeling ecologically significant animal behavior. Forcing serial behavior at other points in the parallel model would cause performance degradation and should be avoided. If, as appears to be the case with the serial NOYELP movement rule, parallel constraints cannot be compensated for by adjusting flexible model parameters, then modification of the basic move-graze structure will be required for development of an efficient parallel model.

7.6 Conclusions

Improvements to the animal movement component of the serial NOYELP model made over the course of this study removed much of the computational complexity from the program loops and reduced execution time by an order of magnitude. Serial improvements and the fundamental incompatibility of the serial movement rule with parallel processing capabilities combined to make parallelization of animal movements on the MasPar MP-2 infeasible for this study. A revised movement rule is proposed which would exploit the advantages of parallel processing by incorporating the following features:

1. Animal groups should be restricted to moving within the local processor subgrid for any given move to accomplish resource pixel comparisons and destination selection with a minimum of inter-processor communication. Hierarchical rather than cut-and-stack mapping would be used to exploit localized communication. This movement rule modification would have great speed advantages, but would necessitate rules governing when a group is moved from one processor to another.

2. All animal groups should move and graze in parallel (within the constraints of virtualization). A parallel move and graze algorithm requires different approaches to biomass updating and allocation of resources when multiple groups are present on a pixel. Available biomass values should be updated at the end of each parallel move/graze action, so that on move m all groups are aware of any resource depletion (which has occurred due to foraging) on move $m - 1$ before choosing its next destination pixel. An advantage of this feature would be the possible elimination of the requirement that animal groups not revisit the same pixel in the same day (which was implemented by setting the movement path to 0 on the available biomass map). This constraint was necessary in the serial version in part to compensate for the fact that an ungulate group could not see the effects of grazing by subsequent groups on the same day. These other groups (lower in the movement series) might choose a previously visited high-resource pixel and deplete its resources before the next graz-

ing choice is made by the higher ranked group. In the parallel algorithm, all grazing effects would be recorded at the end of each parallel grazing event, so that groups could make a more informed choice at each move of the day.

The allocation of resources when multiple groups choose the same destination pixel is a more complex issue presented by a parallel move/graze algorithm. In the serial model, groups choose and graze in sequence, so a group chooses with knowledge of the grazing effects of previous groups, and only chooses a given pixel if it is the maximum resource pixel in its search area. In the parallel model, multiple ungulate groups will choose a pixel at the same time, before any grazing is recorded, unaware of how many other animals are choosing the same pixel. If they could see grazing effects of other groups sharing the destination pixel, they might choose a different pixel as their destination pixel, perhaps one with a smaller current resource level, but with fewer grazers for the current move. To address this resource sharing issue, groups could share resources on the destination pixel evenly, or graze in sequence, based on their position in a movement series or randomly.

Another potential problem with a parallel move/graze algorithm that involves no intra-move updating is that once animal groups share a pixel, they might tend to move together to the same destination pixel. If selection of destination pixels were based solely on available biomass at the start of the move, all groups on the pixel would have the same available resource values to choose from and would choose the same pixel. This phenomenon has been referred to as *artificial synchronization* ([TWWR+93]). This situation is readily addressed by recognizing that animals located on the same pixel are processed on the same PE, and hence are processed in series, thereby providing the opportunity to update biomass according to a movement series within a particular move for groups on the same pixel. In other words, grazing effects for each group could be recorded in sequence, giving subsequent groups on that pixel knowledge of these effects, as in the sequential algorithm. Groups making knowledge-based decisions would be less likely to choose the same destination pixel.

3. One available biomass map should be employed for all animal categories if possible. This would eliminate category-specific variables in the calculation of available biomass, the most time-consuming component of the serial animal movement algorithm. Serial values for the category-specific component of the feedback calculation (`swhi`) are currently 14 for elk calves, 15 for elk cows, 16 for elk bulls and bison calves, and 18 for bison cows and bulls. A single value could be chosen for all categories, and an attempt to compensate for this simplification could be made by substituting complexity in another model component (i.e., adjusting the foraging rate or maintenance energy values for each category). Category-specific information could be retained in the pre-grazing calculation of feedback for foraging. This feature could also be incorporated into the serial model.

Chapter 8

Conclusions

For this thesis, parallel kernels for performing cluster identification and mean squared radius computation were implemented on the MasPar MP-2 using both hierarchical and cut-and-stack data mapping strategies. These kernels were tested on random maps as well as on resource maps extracted from runs of the NOYELP model. Speed improvements for parallel implementations over serial algorithms for communication-intensive cluster identification were modest (<12) for random maps of most sizes and densities tested. The hierarchical algorithm consistently outperformed the cut-and-stack algorithm, which was slower than the serial program for densities near the percolation threshold for all map sizes tested. Larger speed improvements were measured for dense maps with large, dominating clusters. For random maps with $p = 0.85$, the hierarchically-mapped version was over 15 times faster than the serial version on a Sun SPARCstation 2. When cluster identification is performed repeatedly over time steps, as in the NOYELP model, even small speed improvements can be significant.

Speed improvements for parallel implementations of the more computationally-intensive mean squared radius computation, which increased with map size and density, were more substantial. Speed improvements of over 150 were realized for both parallel mapping versions over the serial program on a Sun SPARCstation IPX on 512×512 random maps with $p \geq 0.85$. These results make radius measurements a viable tool for many landscape ecology applications. Speed improvements for NOYELP-derived maps were lower, due largely to the inclusion of many additional non-habitat pixels in the NOYELP grid, which contributed to low densities for these maps. An area for future research is the examination of strategies to improve SIMD performance for applications with irregularly-shaped study areas, perhaps by blocking relevant data (including locality-preserving references) out to the front-end machine and redistributing data equitably among processors. Future work in this area should also involve a closer examination of the the spectrum of natural spatial patterns and cluster configurations encountered in landscape maps, the relative performance of parallel kernels on these configurations, and methods of optimizing parallel performance over the range of patterns.

Parallel kernels for cluster identification or mean squared radius computation could be integrated into the NOYELP model or other landscape ecology models for efficient cluster analysis. Parallel kernels (running on the MasPar MP-2 DPU) could be called by serial programs running on the DECstation 5000-200 front-end machine. Data to be analyzed would be *blocked out* to the DPU, and results could either be returned to the calling program or be written to an output file.

Serial modifications to the cluster identification and animal movement components of the original Fortran-77 NOYELP program resulted in a revised serial version which executes 11 times faster than the original (CPU time on a Sun SPARCstation 2), with no change in functionality. Modifications included replacing the original *local diffusion*-based cluster identification algorithm with a more efficient pseudo-recursive algorithm and revising the calculation of available biomass in the animal movement component. Non-category-specific components in available biomass calculations were removed from nested animal movement loops, so that these calculations are performed once per time step rather than once per animal group per time step.

The results of this thesis effort indicate that, if benefits are to be realized from parallelization of animal movement, the number and complexity of movement operations performed inside parallelizable program loops must be high enough to offset the amount of interprocessor communication required to implement the movement rule chosen. The original serial NOYELP model appeared to fit this criterion, spending 95% of program execution time in the animal movement component of the model. Complex multi-variable calculations of available biomass feedback values are performed repetitively within a daily sequential loop through animal groups. However, performance improvements made to the movement component of the serial NOYELP program (discussed above) resulted in a 24-fold speed improvement over the original movement component (on a Sun SPARCstation 2). After these modifications, insufficient computational complexity remained within the animal movement loop to support the communication requirements associated with parallel implementation of the current movement rule with any significant speed improvement.

The results of this research highlight the fundamental incompatibility of the NOYELP serial movement rule, with its intra-move biomass updating schedule, with parallel processing constraints. Integral to this movement rule is a movement series that determines the order in which ungulate groups move and graze on the NYNP landscape. The model updates biomass prior to each day's set of moves for each ungulate group according to the movement series. It is concluded that efficient parallelization would involve re-conceptualizing the movement rule. Recalibration of the parallel model with suitable ecological data would also be required.

Components of a new parallel movement rule which exploit the advantages of hierarchical mapping are proposed for NOYELP and similar individual-based landscape ecology models. It is suggested that the search area for selection of resource pixels be limited for a given move to the pixels assigned to one processor, thereby limiting the amount of inter-processor communication required. To expand the search area, a group selecting a PE border pixel could be relocated to to the adjoining processor for the next move. A composite suitability index could be computed for each processor's resource subgrid and communicated to adjacent processors to influence the direction of movement. Group-specific information (i.e., location, body weight, category) should be distributed across processors independently of group location on the landscape, and maintained in one place rather than being moved when the group relocates. Available biomass values should be updated at the end of each parallel move/graze, to provide an acceptable level of knowledge in the foraging activities. Given parallel grazing and updating, groups probably do not need to be constrained from revisiting pixels on a given day. When resource pixels are shared by animal groups, biomass could be shared equally or be allocated according to a ranking scheme consistent with the autecology of the particular species. In addition, the new rule should minimize the use of category-specific variables in the calculation of available biomass.

Further conceptualization and development of parallel algorithms to implement the new movement rule are subjects for future research.

Bibliography

Bibliography

[ApCM92] J. Apostolakis, P. Coddington and E. Marinari. *A Multi-Grid Cluster Labeling Scheme.* Europhys. Lett. 17(3), 1992.

[BrTY91] R. C. Brower, P. Tamayo and B. York. *A Parallel Multigrid Algorithm for Percolation Clusters.* Jour. of Stat. Phys. 63, 1991.

[BeCM93a] M. Berry, J. Comiskey, and K. Minser. Parallel Map Analysis on 2-D Grids. *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, Norfolk, VA, 312–319, 1993.

[BeCM93b] M. Berry, J. Comiskey, and K. Minser. Parallel map analysis on the MasPar MP-2. *Computer Science Department Technical Report CS-93-190*, University of Tennessee, March 1993.

[BeTs89] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Communication–Numerical Methods.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[FlTa92] M. Flanigan and P. Tamayo. A Parallel Cluster Labeling Method For Monte Carlo Dynamics. *Int. J. Mod. Phys. C*, 3(1):1235–1249, 1992.

[GaOT93] R. Gardner, R. V. O'Neill, and M. G. Turner. Ecological Implications of Landscape Fragmentation. In: *Humans as Components of Ecosystems: Subtle Human Effects and the Ecology of Populated Areas.* S. T. A. Pickett and M. J. McDonnell, Springer-Verlag, NY, 1992.

[Haef92] J. W. Haefner *Parallel Computers and Individual-Based Models: An Overview.* In: *Individual Based Models and Approaches in Ecology.* D. L. DeAngelis and L. J. Gross, Chapman & Hall, NY, 1992.

[HGTR+92] W. W. Hargrove, R. H. Gardner, M. G. Turner, W. W. Romme, and D. G. Despain. *Simulating Fire Patterns in Heterogeneous Landscapes.* Preprint, 1992.

[HoSa83] E. Horowitz and S. Sahni. *Fundamentals of Data Structures.* Computer Science Press, 1983.

[HoKo76] J. Hoshen and R. Kopelman. *Phys. Rev. B.* 14, 1976.

[HwBr93] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, New York, 1984.

[Lomn92] A. Lomnicki. *Population Ecology from the Individual Perspective.* In: *Individual Based Models and Approaches in Ecology.* D. L. DeAngelis and L. J. Gross, Chapman & Hall, NY, 1992.

[MasP92a] *Data-Parallel Programming Guide*, MasPar Computer Corporation, Sunnyvale California, June (1992).

[MasP92b] *MasPar Parallel Application Language (MPL) Reference Manual*, MasPar Computer Corporation, Sunnyvale California, June (1992).

[MasP92c] *MasPar Parallel Application Language (MPL) User Guide*, MasPar Computer Corporation, Sunnyvale California, June (1992).

[Mins93] K. Minser. *Technical Report* CS-93-197, August 1993.

[Palm92] J. Palmer. *Hierarchical and Concurrent Individual Based Modeling.* In: *Individual Based Models and Approaches in Ecology.* D. L. DeAngelis and L. J. Gross, Chapman & Hall, NY, 1992.

[StAh91] D. Stauffer and A. Aharony. *Introduction to Percolation Theory*, Second edition. Taylor & Francis Ltd, London, 1991.

[TWWR+93] M. G. Turner, Y. Wu, L. L. Wallace, W. H. Romme and A. Brenkert. *Simulating Interactions Among Ungulates, Vegetation and Fire in Northern Yellowstone National Park During Winter.* Submitted to *Ecological Applications*

Appendices

Appendix A

MasPar Specifics

A.1 Family of MPIPL Functions for Virtual Array Conversion

mpi1dcsto2dh - MPIPL function to convert a virtualized array from 1D cut-and-stack to 2D hierarchical.

mpi1dhto2dh - MPIPL function to convert a virtualized array from 1D hierarchical to 2D hierarchical.

mpi1dhto2dh - MPIPL function to convert a virtualized array from 1D hierarchical to 2D hierarchical.

mpi2dcsto2dh - MPIPL function to convert a virtualized array from 2D cut-and-stack to 2D hierarchical.

mpi2dhto1dcs - MPIPL function to convert a virtualized array from 2D hierarchical to 1D cut-and-stack.

mpi2dhto2dcs - MPIPL function to convert a virtualized array from 2D hierarchical to 2D cut-and-stack.

Each of these functions takes the form:

`mpi2sourceXtargetYN(imgSrc, rows, cols, imgDst)`,

taking a virtual array of N-bit objects located at imgSrc using X virtualization, and returning in imgDst the same array revirtualized to use a Y virtualization, where $N = 8, 16,$ or 32.

A.2 Using the MasPar

Much of the code development for this project was done on the MasPar MP-1, maintained by the Joint Institute for Computational Science at the University of Tennessee. This particular machine was upgraded in January (1993) to an MP-2, and all execution times included in this thesis reflect MP-2 timings. The effective speed of the MP-2 ranges between 2 and 4 times faster than that of the MP-1. Although the number of processors remained at 4,096, the MP-2 upgrade uses 32-bit PEs with a 16-bit datapath from PE memory to PE registers. The original MP-1 machine, on the other hand, employed 4-bit PEs with a 4-bit datapath from PE memory to PE registers.

The progressive evolution of programming efforts for this project has been driven by bottleneck resolution. When an algorithm is modified, performance and accuracy over all mapsizes and p-values must be re-evaluated. Some approaches work well for some size/density combinations, but perform poorly for others. The MPPE profiling capability has been very useful for pinpointing where execution time is spent, and for comparing the efficiency of alternative MPL functions.

The (ANSI C compatible) MasPar Programming Language is fairly well documented (except for the omission of specific data mapping examples) and it is relatively easy for a C programmer to write MPL code that works. However, initial efforts are often slow and inefficient. Programming for performance is especially important for parallel applications, particularly in the following areas:

1. **Registers.** - Each PE has 32 32-bit registers available for user-declared register variables. Use of these registers gave a 25-35% performance improvement for cluster identification and radius calculation codes.

2. **Communications** - The availability of several communication constructs in MPL: (**proc**, three types of **xnet**, and global **routing**) provide flexibility, allowing the programmer to minimize communication overhead within the constraints of the application. In some situations (i.e., when a large percentage of PEs need to view the same value) it is more efficient to store values in singular variables visible to all PEs and control access by limiting the active set rather than use router constructs to broadcast these values selectively.

3. **Pipelining.** - Stores to memory are always time-consuming, but the programmer can take advantage of pipelining on the MasPar by intelligent use of registers. For example:

```
registertemp=sum[n];
registertemp+=value_to_store;
sum[n]=registertemp;
```

is significantly faster than

```
sum[n]+=value_to_store;
```

4. **Variable size.** - Operations with *shorts* are twice as fast on MasPar PEs as operations with *ints*, and operations with *ints* are twice as fast as operations with *long longs*, so the programmer should carefully calculate the maximum variable size needed. Significant performance improvements were seen in this project by using *ints* for low-level calculations, and casting to larger sizes as values are accumulated.

5. **Storage of results.** - Address space limitations on the ACU (where singular variables are handled) are fairly restrictive. Rethinking of traditional programming approaches can often yield alternatives to array storage for collecting results which are more efficient for parallel applications and allow larger problems to be solved within the memory constraints.

Appendix B

NOYELP Specifics

B.1 Outline of NOYELP Subroutines

The serial NOYELP model is implemented as a 4736 line Fortran-77 program, divided into 13 subroutines.

SUBROUTINE INPUT()
lines: 304
Subroutine `input()` inputs the parameter values, snow index matrix, habitat type matrix, and interface choices, and sets initial rows and columns for bison distribution.
parameters: none
called by: `main()`
calls to to: none

SUBROUTINE INITIAL()
lines: 701
Subroutine `initial()` initializes biomass and ungulate location matrices and establishes the chosen pattern of burned areas on the landscape.
parameters: none
called by: `main()`
calls to to: `unevenb(), ranmap()`

SUBROUTINE UNEVENB()
lines: 40
Subroutine `unevenb()` creates an uneven biomass map for the landscape.
parameters: none
called by: `initial()`
calls to: none

SUBROUTINE RANMAP()
lines: 63
Subroutine `ranmap()` creates a random habitat map for the landscape, if random landscape is chosen.
parameters: none
called by: `initial()`
calls to: none

SUBROUTINE SSNOW()
lines: 174
Subroutine `ssnow()` distributes snow depth in each pixel and updates it at the start of each snow period (3 days).
parameters: none
called by: `main()`
calls to: none

SUBROUTINE UNGFIND()
lines: 153
Subroutine `ungfind()` locates each ungulate group on the landscape map, then initiates moving and grazing for the day.
parameters: none
called by: `main()`
calls to: `graze()`, `move()`

SUBROUTINE GRAZE(I,J,UNGRANK)
lines: 128
Subroutine `graze()` allows an ungulate in a resource pixel to forage. Total biomass on the grazed pixel is reduced by the amount consumed by the grazing ungulate group. Foraging is precluded on a given site when biomass declines below a *refuge* level.

parameters:	integer `i`	row (x-coord.) of current group
	integer `j`	column (y-coord.) of current group
	integer `ungrank`	rank of current group in movement series

called by: `ungfind()`
calls to: `move()`, `energet()`

SUBROUTINE MOVE(I,J,UNGRANK)
lines: 277
Subroutine `move()` implements the destination resource pixel selection phase of the movement rule.

parameters:	integer `i`	row (x-coord.) of current group
	integer `j`	column (y-coord.) of current group
	integer `ungrank`	rank of current group in movement series

called by: `graze()`, `ungfind()`
calls to: none

SUBROUTINE ENERGET(I,J,UNGRANK)
lines: 169
Subroutine `energet()` calculates energy cost and gain each day for each ungulate. Body weight lost is tracked and mortality is simulated.

parameters:	integer `i`	row (x-coord.) of current group
	integer `j`	column (y-coord.) of current group
	integer `ungrank`	rank of current group in movement series

called by: `graze()`
calls to: none

SUBROUTINE PATCH()
lines: 350
Subroutine `patch()` identifies clusters of available biomass pixels and reports the number of clusters, average cluster size, and largest patch size for available resources. Cluster analysis is performed twice each day: once for pixels with available biomass above 0, and again for pixels with available biomass above a preset alpha level.
parameters: none
called by: `main()`
calls to: none

SUBROUTINE STATIST()
lines: 1018
Subroutine `statist()` provides a statistical summary over all replicate runs and calculates mean, minimum, maximum values and standard deviation for each replicate at day 180.
parameters: none
called by: `main()`
calls to: none

SUBROUTINE OUTFILE()
lines: 109
Subroutine `outfile()` writes summary results to an outfile.
parameters: none
called by: `main()`
calls to: none

B.2 Selected Formulas used in NOYELP model computations

Feedback due to foraging: represents the effect of a reduction in the amount of available forage on the rate of forage intake.

$$FBforage = 1 - \frac{REF}{BIO}$$

where:
FB_{forage} = value ranging from 0–1 (unitless),
REF = refuge value of biomass not available to ungulates (kg/ha), and
BIO = actual biomass in that pixel (kg/ha).

Feedback due to snow: represents the effect of snow on the ability of an animal to forage.

$$FB_{snow} = \left[1 - \frac{(SWE - SWLO)_{+}}{SWHI - SWLO}\right]_{+}$$

where:
FB_{snow} = value ranging from 0–1 (unitless),
SWE = actual snow water equivalent in that grid cell,
$SWLO$ = the SWE value at which limitation to foraging begins,
$SWHI$ = the SWE value at which feeding goes to zero, and
$+$ indicates that the term must remain positive, i.e., it is set to zero if negative.

DIST: maximum daily moving distance as modified by snow conditions; used to constrain animal movement such that elk or bison move a shorter distance when snow conditions are severe.

$$DIST = \frac{MDISTM}{1 + Y/100}$$

where:
$DIST$ = the modified maximum daily moving distance in snow,
$MDISTM$ = the initial maximum daily moving distance (category-specific) and,
Y = the relative increase in travel energetics in snow.

Energy balance equations:

$$E_{balance} = E_{gain} - E_{cost}$$
$$E_{gain} = I * ENPK$$
$$E_{cost} = E_{maint} - E_{travel}$$

where:
I = total intake of forage(kg), $ENPK$ = forage energy content (kcal/kg),
E_{maint} = metabolizable energy needed for zero energy balance, excluding travel (kcal), and
E_{travel} = energy cost of travel (kcal).

Energy content of forage:

$$ENPK = GE * IVDMD * MC$$

where:
$ENPK$ = forage energy content (kcal/kg),
GE = gross energy (4400 kcal/kg),
$IVDMD$ = in vitro dry matter digestibility (0.374 from field data), and
MC = metabolizable energy coefficient (0.82).

E$_{maint}$**:** Maintenance energy

$$E_{maint} = ME * BW^{0.75}$$

where:
ME = metabolizable energy needed per kg body weight (kcal/kg), and
BW = present body weight (kg) of the ungulate.

E$_{travel}$**(no snow):** Energy cost of traveling in the absence of snow.

$$E_{travel}(nosnow) = [2.97kcal/kg * BW^{-0.34}] * BW * S$$

where:
S = distance traveled (km), and
BW = present body weight of the animal (kg).

Y: relative increase in energy costs for travel in snow (%)

$$Y = [0.71 + 2.6(\rho - 0.2)] * RSD * e^{0.019+0.016(\rho-0.2)]} * RSD$$

where:
ρ = snow density (g/cm^3), and RSD = relative sinking depth [(sinking depth/brisket height) $\times$ 100].

Appendix C

Timing Tables

Table C.1: Wall-clock times for cluster identification with cut-and-stack data mapping on the MasPar MP-2 (all times are in seconds).

		Map Size						
Operation	p-value	64	128	256	512	768	1024	2048
Read	0.10	0.020	0.020	0.047	0.129	0.297	0.523	10.406
Label	0.10	0.008	0.012	0.016	0.062	0.148	0.230	1.137
Collect	0.10	0.004	0.004	0.004	0.066	0.289	0.813	11.797
ClusterID	0.10	0.012	0.016	0.020	0.129	0.437	1.043	12.934
Total	0.10	0.031	0.035	0.066	0.258	0.734	1.566	23.340
Read	0.30	0.012	0.023	0.047	0.133	0.297	0.520	9.980
Label	0.30	0.008	0.016	0.031	0.129	0.277	0.570	2.328
Collect	0.30	0.004	0.000	0.020	0.086	0.328	0.895	12.152
ClusterID	0.30	0.012	0.016	0.051	0.215	0.605	1.465	14.480
Total	0.30	0.023	0.039	0.098	0.348	0.902	1.984	24.461
Read	0.62	0.020	0.027	0.043	0.133	0.297	0.523	10.285
Label	0.62	0.020	0.070	0.297	1.094	2.188	4.480	19.746
Collect	0.62	0.023	0.043	0.066	0.164	0.437	1.055	12.648
ClusterID	0.62	0.043	0.113	0.363	1.258	2.625	5.535	32.395
Total	0.62	0.062	0.141	0.406	1.391	2.922	6.059	42.680
Read	0.85	0.020	0.023	0.043	0.133	0.301	0.520	10.000
Label	0.85	0.020	0.039	0.148	0.582	1.270	2.293	8.977
Collect	0.85	0.035	0.039	0.047	0.082	0.191	0.445	5.168
ClusterID	0.85	0.055	0.078	0.195	0.664	1.461	2.738	12.145
Total	0.85	0.074	0.102	0.238	0.797	1.762	3.258	24.145
Read	1.00	0.016	0.023	0.039	0.133	0.293	0.519	10.215
Label	1.00	0.012	0.039	0.145	0.543	1.207	2.125	8.508
Collect	1.00	0.039	0.043	0.039	0.043	0.059	0.070	0.176
ClusterID	1.00	0.051	0.082	0.184	0.586	1.266	2.195	8.684
Total	1.00	0.066	0.106	0.223	0.719	1.559	2.715	18.898

Note: ClusterID is the sum of Label and Collect times.

Table C.2: Wall-clock times for cluster identification with hierarchical data mapping on the MasPar MP-2 (all times are in seconds).

		Map Size						
Operation	p-value	64	128	256	512	768	1024	2048
Read	0.10	0.020	0.031	0.039	0.105	0.250	0.496	9.539
Label	0.10	0.008	0.004	0.012	0.023	0.043	0.074	0.305
Collect	0.10	0.000	0.000	0.004	0.016	0.047	0.113	0.801
ClusterID	0.10	0.008	0.004	0.016	0.039	0.090	0.187	1.105
Total	0.10	0.027	0.035	0.055	0.145	0.340	0.684	10.750
Read	0.30	0.027	0.023	0.043	0.559	0.250	0.477	9.668
Label	0.30	0.012	0.008	0.012	0.027	0.047	0.082	0.344
Collect	0.30	0.000	0.000	0.004	0.039	0.117	0.289	2.098
ClusterID	0.30	0.012	0.008	0.016	0.066	0.164	0.371	2.441
Total	0.30	0.039	0.031	0.059	0.625	0.414	0.848	12.109
Read	0.62	0.031	0.027	0.047	0.113	0.250	0.469	10.000
Label	0.62	0.023	0.027	0.039	0.074	0.113	0.160	0.488
Collect	0.62	0.023	0.027	0.062	0.141	0.273	0.519	4.723
ClusterID	0.62	0.047	0.055	0.102	0.215	0.387	0.680	5.211
Total	0.62	0.078	0.082	0.148	0.328	0.637	1.148	15.211
Read	0.85	0.023	0.027	0.043	0.109	0.250	0.469	9.953
Label	0.85	0.016	0.020	0.023	0.031	0.059	0.094	0.363
Collect	0.85	0.035	0.043	0.047	0.066	0.105	0.152	0.855
ClusterID	0.85	0.051	0.062	0.070	0.098	0.164	0.246	1.219
Total	0.85	0.074	0.090	0.113	0.207	0.414	0.715	11.172
Read	1.00	0.023	0.023	0.039	0.105	0.250	0.496	9.754
Label	1.00	0.012	0.020	0.020	0.031	0.047	0.074	0.305
Collect	1.00	0.043	0.039	0.039	0.043	0.047	0.047	0.666
ClusterID	1.00	0.055	0.059	0.059	0.074	0.094	0.121	0.371
Total	1.00	0.078	0.082	0.098	0.180	0.344	0.617	10.125

Note: ClusterID is the sum of Label and Collect times.

Table C.3: Wall-clock times for pseudo-recursive sequential Fortran cluster identification program on Sun SPARCstation 2 (all times are in seconds).

		Map Size						
Operation	p-value	64	128	256	512	768	1024	2048
Read	0.10	0.04	0.19	0.66	2.83	52.75	28.12	46.75
ClusterID	0.10	0.00	0.01	0.08	0.32	0.76	1.32	5.31
Total	0.10	0.06	0.23	0.93	3.78	55.04	31.84	62.11
Read	0.30	0.33	1.34	0.68	23.69	6.34	11.45	46.53
ClusterID	0.30	0.01	0.02	0.12	0.47	1.13	1.97	7.92
Total	0.30	0.35	1.42	0.95	24.99	8.92	16.04	64.35
Read	0.62	0.33	0.19	0.67	2.83	6.25	11.41	46.62
ClusterID	0.62	0.01	0.04	0.17	0.68	1.58	2.75	11.16
Total	0.62	0.35	0.26	0.99	4.12	9.34	16.74	66.28
Read	0.85	0.05	0.15	0.68	23.90	6.57	11.67	46.52
ClusterID	0.85	0.01	0.04	0.22	0.88	2.08	3.71	15.11
Total	0.85	0.07	0.25	1.04	25.39	9.91	17.62	71.65
Read	1.00	0.35	1.33	5.51	23.82	6.41	11.26	46.89
ClusterID	1.00	0.02	0.06	0.27	1.14	2.71	4.70	19.40
Total	1.00	0.38	1.43	6.00	25.62	10.52	18.65	75.93

Table C.4: Wall-clock times for total map analysis (including radius computation) with cut-and-stack data mapping on the MasPar MP-2 (all times are in seconds).

		Map Size			
Operation	p-value	64	128	256	512
Read	0.10	0.020	0.031	0.047	0.141
ClusterID	0.10	0.016	0.020	0.039	0.145
Radius	0.10	0.090	0.277	1.106	6.027
Total	0.10	0.117	0.316	1.176	6.230
Read	0.30	0.023	0.035	0.047	0.141
ClusterID	0.30	0.016	0.023	0.059	0.246
Radius	0.30	0.168	0.586	2.242	9.949
Total	0.30	0.199	0.633	2.328	10.223
Read	0.62	0.023	0.027	0.051	0.133
ClusterID	0.62	0.074	0.172	0.703	2.684
Radius	0.62	0.305	1.422	13.934	187.158
Total	0.62	0.352	1.520	14.273	188.400
Read	0.85	0.023	0.027	0.047	0.141
ClusterID	0.85	0.102	0.234	0.758	2.863
Radius	0.85	0.418	2.207	21.289	289.184
Total	0.85	0.461	2.281	21.488	289.910
Read	1.00	0.024	0.026	0.046	0.135
ClusterID	1.00	0.109	0.266	0.855	3.215
Radius	1.00	0.387	2.340	24.301	337.777
Total	1.00	0.528	2.621	25.206	341.927

Table C.5: Wall-clock times for total map analysis (including radius computation) with hierarchical data mapping on the MasPar MP-2 (all times are in seconds).

		Map Size			
Operation	p-value	64	128	256	512
Read	0.10	0.027	0.023	0.043	0.105
ClusterID	0.10	0.016	0.016	0.027	0.086
Radius	0.10	0.094	0.285	1.062	4.426
Total	0.10	0.137	0.332	1.133	4.617
Read	0.30	0.027	0.023	0.043	0.109
ClusterID	0.30	0.012	0.016	0.027	0.125
Radius	0.30	0.137	0.605	3.063	16.309
Total	0.30	0.180	0.648	3.137	16.543
Read	0.62	0.023	0.023	0.047	0.109
ClusterID	0.62	0.051	0.059	0.086	0.207
Radius	0.62	0.254	1.461	14.629	196.516
Total	0.62	0.328	1.555	14.770	196.836
Read	0.85	0.027	0.027	0.039	0.109
ClusterID	0.85	0.059	0.062	0.074	0.117
Radius	0.85	0.332	2.031	20.730	286.707
Total	0.85	0.418	2.125	20.848	286.938
Read	1.00	0.030	0.028	0.043	0.111
ClusterID	1.00	0.062	0.062	0.152	0.078
Radius	1.00	0.383	2.363	24.316	337.453
Total	1.00	0.475	2.454	24.520	337.646

Table C.6: Wall-clock times for optimized sequential C program for total map analysis, including read time, cluster identification, and mean squared radius computation on SPARCstation IPX (all times are in seconds).

	Map Size			
p-value	64	128	256	512
0.10	0.12	1.40	23.97	389.66
0.30	0.36	5.43	96.69	1515.39
0.62	3.24	37.27	1052.27	19914.67
0.85	9.57	165.70	2718.12	44156.01
1.00	17.84	236.44	3791.13	59379.85

```
/**************************************************************/
/* Original GMMSR mean squared radius
 *   C program fragment:
 */

Radius(lbl,size,rms)
int lbl;
int size;
double rms;

{
double rid,rjd,r2,s2,*istack,*jstack;
int i,j,nstack;

  /* allocate space for map coordinate vectors */
istack = (double *)malloc(((NROWS*NCOLS)+4)*sizeof(double))
jstack = (double *)malloc(((NROWS*NCOLS)+4)*sizeof(double))

  /* fill istack with x-coordinates of cluster (not shown) */
  /* fill jstack with y-coordinates of cluster (not shown) */

  /* nstack = number of coordinates */
  /* calculate squared coordinate differences */
       for(i=0;i<nstack-1;i++){
          for(j=i+1;j<nstack;j++){
             rid = fabs(istack[j]-istack[i])+1;
             rjd = fabs(jstack[j]-jstack[i])+1;
             r2 += rid*rid + rjd*rjd;
           }
        }

       /* calculate mean radius */
       s2 = (double)size;
       s2 = s2*s2;
       *rms = sqrt(r2/s2);
}
/**************************************************************/
```

```
/***********************************************************/
/* Modifications made to GMMSR for improved performance
 *  (for array storage of x- and y-coordinates)
 */

Radius(lbl,size,rms)
int lbl;
int size;
double rms;

{
int *istack,*jstack,nstack1,nstack,i,j;
double rid,rjd,r2,s2;

  /* allocate space for map coordinate vectors */
istack = (int *)malloc(size*sizeof(int))
jstack = (int *)malloc(size*sizeof(int))

  /* fill istack with x-coordinates of cluster (not shown) */
  /* fill jstack with y-coordinates of cluster (not shown) */

  /* nstack = number of coordinates */
  /* calculate squared coordinate differences */
       nstack1=nstack-1;
       for(i=0;i<nstack1;i++){
          for(j=i+1;j<nstack;j++){
             rid = istack[j]-istack[i]+1;
             if((rjd = (jstack[j]-jstack[i]))<0) rjd=-rjd;
             rjd+=1;
             r2 += rid*rid + rjd*rjd;
          }
       }

       /* calculate mean radius */
       s2 = size;
       *rms = sqrt(r2)/s2;
}
/***********************************************************/
```

Table C.7: Comparison of CPU times on Sun SPARCstation IPX for Gardner/Minser mean squared radius C program with four revision stages.*

		Map Size			
Algorithm	p-value	64	128	256	512
GMMSR:	0.10	0.09	1.41	25.25	406.23
	0.30	0.35	5.48	102.39	1618.90
	0.62	6.33	82.26	2166.23	36343.40
	0.85	19.12	348.33	5595.56	89555.51
STAGE1:	0.10	0.09	1.42	25.21	405.85
	0.30	0.36	5.47	102.07	1607.33
	0.62	3.25	54.75	1439.71	24311.35
	0.85	9.44	232.30	3722.71	59800.57
STAGE2:	0.10	0.10	1.32	23.67	380.22
	0.30	0.33	5.07	95.58	1506.37
	0.62	3.18	36.86	1047.55	17750.05
	0.85	9.44	164.68	2707.21	43564.97
STAGE3:	0.10	0.09	1.31	23.66	380.58
	0.30	0.33	5.16	95.62	1507.75
	0.62	3.21	36.86	1047.27	19701.13
	0.85	9.49	164.60	2705.73	42282.11
LOOKUP:	0.10	0.07	0.94	18.76	303.55
	0.30	0.24	3.67	75.72	1202.34
	0.62	2.57	35.19	837.57	14011.50
	0.85	6.23	99.81	1741.30	28252.92

*See Table C.8 for a description of revision stages.

Table C.8: Optimizing modifications made to serial C program for computing mean squared radius.

GMMSR: Gardner/Minser serial C program for mean squared radius computation. Note: the declaration of istack and jstack as pointers to `doubles` and the subsequent calls to `fabs()` in the original program are more efficient (require less CPU time) than declaration as `ints` with calls to `abs()` on the architectures tested.
Stage1: Eliminate calls to C library function `fabs()` by computing absolute values in place for y-coordinate differences (i.e., if($i < 0$) $i = -i$). Differences in x-coordinates will always be positive as implemented, so no absolute value need be computed for x-coordinate differences.
Stage2: Use dynamically-allocated arrays which are the exact size of the cluster being analyzed to hold x- and y-coordinates; remove arithmetic from *for* loop indices.
Stage3: Do not store x- and y-coordinates in arrays when clustersize is greater than half the map size. Instead, use indices of `for` loops to indicate pixel positions in map and accumulate squared differences as `for` loop is executed. This is somewhat faster for large clusters and reduces memory demands. The optimal cut-off size may vary for maps with different cluster characteristics.
Lookup: Read squared coordinate differences and square roots from lookup tables. Lookup tables are read from binary files, with a separate file for each map size analyzed.

Appendix D

Prologues of Selected Procedures

```
/**********************************************************************
 *                                                                    *
 *                              CSMSR                                 *
 *                                                                    *
 **********************************************************************/
/***********************************************************************

   Description
   -----------
   Cluster identification and mean squared radius computation using
     cut-and-stack data mapping.  Map size is specified by NROWS and
     NCOLS in #define section.

   Input:
   ---------
   Data values for map to be analyzed are read from a binary input
     file of size NROWS x NCOLS specified on the command line.

   Output:
   ---------
   Results are written to screen.
   Cluster statistics: number of clusters, average cluster size,
     maximum cluster size, and mean squared radius.
   Wall-clock execution times: from gettimeofday(), in seconds,
     for read, label, collect, cluster ID and radius separately,
     for total I/O (read plus printout), and for total time.

   Functions called:
   ---------
   CSlabel(): assigns final cluster labels to plural int variable
          "cluster[]".
   CScollect(): collects cluster information at head element of cluster
      and calculates cluster statistics.
   CSradius1(): calculates mean squared radius for small and large
      clusters separately; returns largest mean squared radius as
      a double.
   CSradius2(): calculates mean squared radius for all clusters,
      regardless of size; returns largest mean squared radius as
      a double.

 **********************************************************************/
```

```
/*********************************************************************
 *                                                                   *
 *                            CSlabel()                              *
 *                                                                   *
 *********************************************************************/
/*********************************************************************
  Description:
   ---------
  Function CSlabel() identifies and labels clusters for programs
    using cut-and-stack mapping, using xnet() to make N/S and E/W
    comparisons of cluster ID numbers.  The numerically lowest ID
    number is assigned to all contiguous cluster elements.  This
    algorithm does not require a border of 0s around the map, and
    uses toroidal wrap to access stacked pages.

  Arguments:
  ---------
  plural int cluster[NROWS*NCOLS/nproc];

  Output:
  ---------
  modifies cluster[] array to hold final cluster labels for each
    map pixel.

 *********************************************************************/
```

```
/*******************************************************************
 *                                                                 *
 *                          CScollect()                            *
 *                                                                 *
 *******************************************************************/
/*******************************************************************
  Description:
  -----------
  Function CScollect() collects and sends cluster information from
    cluster members to the head element of each cluster, and
    reports number of clusters, average cluster size, and size of
    the largest cluster.  The address of the head element of
    each cluster can be calculated locally from each member's
    new cluster label.  Results are accumulated locally for
    each cluster represented on a PE before using p_sendwithAdd()
    to transmit local sums to the head element.

  Arguments:
  ---------
    plural int cluster[NROWS*NCOLS/nproc]
    plural int clustersize[NROWS*NCOLS/nproc]
    plural int headcount[NROWS*NCOLS/nproc]
    plural short headptr[NROWS*NCOLS/nproc]
    int largest;
    int numcl;
    float average;

  Output:
  ---------
    o size of each cluster whose head element is located on local PE
      in the clustersize[] array (in the array position indexed by
      the array position of the head cluster element in cluster[];)
    o size of the largest cluster in integer variable largest
    o number of clusters in singular integer variable numcl;
    o average cluster size in singular float variable average.

 *******************************************************************/
```

```
/*******************************************************************
 *                                                                 *
 *                            CSradius1()                          *
 *                                                                 *
 *******************************************************************/
/*******************************************************************

  Description:
  -----------
  Function CSradius1() calculates mean squared radius for clusters
    which do not overlap 64 x 64 cut-and-stack "pages" separately
    from larger, overlapping clusters.  To compute mean squared
    radius, an xnet-shift operation is used to allow each element
    on a page to view all other elements.  Cluster labels are compared,
    and if elements are in the same cluster coordinate differences
    are calculated, squared, and summed for all elements in a cluster;
    the square root is then taken and divided by the number of elements
    in the cluster.  Radius is computed for clusters which overlap
    pages as in function CSradius2().

  Arguments:
  ---------
    plural int cluster[NROWS*NCOLS/nproc]
    plural int clustersize[NROWS*NCOLS/nproc]
    plural int headcount[NROWS*NCOLS/nproc]
    int largest;

  Outputs:
  ---------
     Largest radius is returned in singular double variable "rad".
     Radius values for all clusters are stored in plural double array
       "rsum";

 *******************************************************************/
```

```
/*********************************************************************
 *                                                                   *
 *                          CSradius2()                              *
 *                                                                   *
 *********************************************************************/
/*********************************************************************

 Description:
 -----------
 Function CSradius2() calculates mean squared radius of each cluster
   and reports the largest mean squared radius among clusters having
   the largest number of elements (as determined by cluster
   identification functions).  To compute mean squared radius,
   differences between each element in a cluster and every other
   element are squared and summed. This is accomplished by copying
   the cluster label, x-coordinate and y-coordinate of cluster members
   to singular (shared) variables visible to all PEs.  Squared
   differences are summed for all elements in a cluster; the square
   root is then taken and divided by the number of elements in the
   cluster.

 Arguments:
 ---------
   plural int cluster[NROWS*NCOLS/nproc]
   plural int clustersize[NROWS*NCOLS/nproc]
   int largest;

 Outputs:
 ---------
    Largest radius is returned in singular double variable "rad".
    Radius values for all clusters are stored in plural double array
      "rsum";

 *********************************************************************/
```

VITA

Ethel Jane Goodman Comiskey was born in Knoxville, Tennessee, January 17, 1948. She was Valedictorian of Rule High School in Knoxville and received the Bachelor of Arts degree in Botany from the University of Tennessee in May 1972. She is married to Dr. Charles Comiskey and has two daughters, Jennifer and Andrea. In August 1990, she entered the Computer Science program at the University of Tennessee and was awarded the Master of Science degree in Computer Science in August 1993.