PARALLEL MAP ANALYSIS ON THE CM-5 FOR LANDSCAPE ECOLOGY MODELS

Karen Stoner Minser

Computer Science Department

CS-93-197 August 1993

Parallel Map Analysis on the CM-5 for Landscape Ecology Models

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

Karen Stoner Minser

August 1993

Acknowledgements

I would like to thank my advisor, Dr. Michael Berry, for his constant enthusiasm, patience, and support. His abilities as a computer scientist, mathematician, and writer made this project a successful learning experience. I would also like to thank Drs. Jack Dongarra and Bob Gardner for serving on my committee. Dr. Gardner, an expert in landscape ecology, not only provided us with this thesis problem, but furnished ecological wisdom along the way. Finally, I would like to thank my family for their support and understanding, tolerating many disrupted schedules and fast-food dinners.

This research was supported by the National Science Foundation under grant number NSF CDA-9115428.

Abstract

In landscape ecology, computer modeling is used to assess habitat fragmentation and its ecological implications. Specifically, maps (2-D grids) of habitat clusters are analyzed to determine numbers, sizes, and geometry of clusters. Previous ecological models have relied upon sequential Fortran-77 programs which have limited the size and density of maps that can be analyzed. To efficiently analyze relatively large maps, we present parallel map analysis software implemented on the CM-5. For algorithm development, random maps of different sizes and densities were generated and analyzed. Initially, the Fortran-77 program was rewritten in C, and the sequential cluster identification algorithm was improved and implemented as a recursive or nonrecursive algorithm. The major focus of parallelization was on cluster geometry using C with CMMD message passing routines. Several different parallel models were implemented: host/node, hostless, and host/node with vector units (VUs). All models obtained some speed improvements when compared against several RISC-based workstations. The host/node model with VUs proved to be the most efficient and flexible with speed improvements for a 512×512 map of 187, 95, and 20 over the Sun Sparc 2, HP 9000-750, and IBM RS/6000-350, respectively. When tested on an actual map produced through remote imagery and used in ecological studies this same model obtained a speed improvement of 119 over the Sun Sparc 2.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this thesis, we present parallel map analysis software used to define spatial relationships of ecological resources. Initially, the original Fortran-77 program which was developed by Dr. Robert H. Gardner at the Environmental Sciences Division of the Oak Ridge National Laboratories, was rewritten in C and then parallelized on the CM-5. The definition of the problem and our goal is discussed in the next section followed by the definition of map analysis. In Chapter 2, the methods used are explained, and Chapter 3 contains the results. Our conclusions are given in Chapter 4 along with suggestions for future work.

1.1 Problem Definition and Goal

Landscape ecology is the study of habitat fragmentation and its ecological implications. Computer modeling is typically used to assess landscape patterns and how particular organism(s) might react to changing patterns or conditions. For example, researchers at the Oak Ridge National Laboratory are studying the effects of the 1988 fires in Yellowstone National Park on the distribution of forage and its effect on the ungulate[1] movement patterns.

Within the computer models a series of map analyses are performed that involve traversal of a 2-D grid to identify habitat clusters (cluster identification) and to measure the clusters' sizes and shapes (cluster geometry). Depending on the size and density of the grid, the analysis can be extremely time-consuming. The original code used in these models consisted of a sequential Fortran-77 program, which imposed a practical limitation on the size of the map and density of clusters to be analyzed. Therefore, if map analysis is to be completed in reasonable lengths of time, sequential methods are restricted to resolving grids whose dimensions are less that 500 rows and columns.

[1] Ungulates are hoofed animals, and these studies focus on the bison and elk.

Our goal is to develop scalable map analysis software on a parallel machine, the CM-5, that can efficiently resolve cluster identification and cluster geometry on large maps arising from landscape models or remote imagery applications.

1.2 Definition of Map Analysis

The program accepts an input file representing a map or 2-D grid which contains nonnegative integers with 0's representing unsuitable habitat and positive integers representing different habitats or mapclasses. The input file is loaded into an (`nrows`+ 2) $\times$ (`ncols` + 2) array[2], `in_map`, in which the two extra rows and columns define a border of -9's around the (`nrows`) $\times$ (`ncols`) grid. All mapclasses are analyzed separately so that the numbering of clusters is not confused between classes. An (`nrows` + 2) $\times$ (`ncols` + 2) array, `wk_map`, is also created to hold the grid values of the current mapclass being analyzed. Wherever current mapclass values are located in `in_map`, corresponding -1's are placed in `wk_map`; the remaining positions within `wk_map` are filled with 0's. After `wk_map` has been filled, it is also surrounded with a border of -9's.

1.2.1 Cluster Identification

The first step in analyzing a mapclass, or particular habitat type, involves locating and labeling clusters of habitat represented by -1's in `wk_map`. A cluster can be defined by different neighbor rules. The original Fortran-77 code predefines three different neighbor rules and also allows the user to define a neighbor rule at runtime. Rule 1 defines a cluster as a set of adjacent pixels[3] that border each other in one or more of the four cardinal directions, North, East, West, and South (NEWS). A working grid containing 5 clusters as determined by Rule 1 is illustrated in Figure 1.1. Although we only apply Rule 1 for cluster identification in all our computer implementations, we illustrate the two other neighbor rules to be complete. The choice of neighbor rule is certainly application-dependent.

Rule 2 defines clusters as pixels that border each other in the NEWS directions as well as on the diagonals (see Figure 1.2), and Rule 3 is essentially Rule 2 plus pixels that border each other in the NEWS directions two positions away (see Figure 1.3).

The grid is typically traversed sequentially while searching for clusters of -1's. Upon locating a cluster, the pixels are re-labeled with the current cluster label (labeling is in numerical order), and the size of the cluster is determined. After the cluster has been identified, the current cluster label is incremented by one for the next cluster, and traversal of the grid continues. Cluster identification terminates when all clusters

[2] where `nrows` and `ncols` reflect the number of rows and columns, respectively, in the 2-D grid.

[3] Pixels and sites refer to the individual grid elements of the map and are used interchangeably throughout this thesis.

0	-1	0	-1	0	0	0
-1	-1	-1	0	-1	0	-1
0	-1	0	-1	0	0	0

(a)

0	1	0	2	0	0	0
1	1	1	0	3	0	4
0	1	0	5	0	0	0

(b)

Figure 1.1: Rule 1: before (a) and after (b) labeling.

0	-1	0	-1	0	0	0
-1	-1	-1	0	-1	0	-1
0	-1	0	-1	0	0	0

(a)

0	1	0	1	0	0	0
1	1	1	0	1	0	2
0	1	0	1	0	0	0

(b)

Figure 1.2: Rule 2: before (a) and after (b) labeling.

have been located and uniquely labeled, and the total number of clusters, the largest cluster size, and the average cluster size has been determined.

1.2.2 Cluster Geometry

Computing the geometry of clusters involves calculating the mean squared radius for each cluster. The mean squared radius (R_s^2) of a cluster ([StAh91]) is given by

$$2R_s^2 = \sum_{i,j} \frac{|r_i - r_j|^2}{s^2} \ , \tag{1.1}$$

where r_i is the position of the i-th element in the cluster, and s is the number of elements in the cluster. To determine distances between pixel positions, the absolute differences of the x-coordinates and y-coordinates between pixels must be calculated. This distance between pixels must be determined for each pixel of a cluster, so that the computational complexity of (1.1) is essentially $O(n^2)$, where n is the number of pixels in a cluster. In the sequential algorithm, the grid must be traversed for every cluster, and arrays must be used to store the x- and y-coordinates for each pixel

0	-1	0	-1	0	0	0
-1	-1	-1	0	-1	0	-1
0	-1	0	-1	0	0	0

(a)

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

(b)

Figure 1.3: Rule 3: Before (a) and after (b) labeling.

belonging to the current cluster. After all coordinates are recorded, the absolute value of the differences in coordinate distances are calculated, incremented by one, squared, and added to an accumulating total, τ, defined by

$$\tau = \sum_{i,j} |r_i - r_j|^2 \ . \tag{1.2}$$

A sample C program to calculate τ is shown in Figure 1.4. Using (1.2), R_s^2 is then given by

$$R_s^2 = \frac{\tau}{s^2} \ . \tag{1.3}$$

In landscape ecology, the mean squared radius of each cluster is needed to calculate the correlation length ([StAh91])

$$\xi^2 = \frac{2 \sum_s R_s^2 s^2 N_s}{\sum s^2 N_s} \ , \tag{1.4}$$

where N_s is the number of clusters of size s. The correlation length, ξ, is the average distance of two pixels belonging to the same cluster ([GaOT93]). Since the correlation length, ξ^2, is easily derived using the mean squared radius, R_s^2, this thesis will focus on the more time-consuming computation of R_s^2.

1.3 Random Maps

In order to simulate cluster analysis on general maps, we generate random maps to facilitate algorithm development and to test our models. Analysis of random maps allows the accuracy and performance of the program to be tested on various sizes and densities of prospective maps of interest to landscape ecologists. Typically, $m \times m$ grids are randomly generated by setting the grid elements or pixels to 1 with a probability of p (or equivalently to 0 with a probability of $1-p$). The value 1 usually

```
for(i=0;i<arraysize:i++) {
  for(j=i+1;j<arraysize;j++){
     rid = fabs(xcoords[i] - xcoords[j]) + 1;
     rjd = fabs(ycoords[i] - ycoords[j]) + 1;
     r2 += rid*rid + rjd*rjd;
    }
  }
```

Figure 1.4: C code for determining τ.

represents suitable habitat while 0 represents unsuitable habitat. Thus, pm^2 is the number of pixels of suitable habitat (or 1's) within a map ([GaOT93]). For testing purposes, we generate $m \times m$ random maps, where $m = 64k$, $k = 1$, 2, 4, 8, 16, and $p = 0.1$, 0.3, and 0.62 for each value of m.

As shown in Tables 1.1 and A.1, all cluster sizes for maps with $p = 0.1$ and $p = 0.3$ are smaller than 100. Cluster size increases slowly until p reaches a critical threshold when the cluster size of a single cluster changes dramatically and is capable of extending from one edge of the map to the other. This sudden change in cluster size is explained by percolation theory ([StAh91]), where random maps with p-values greater than a critical threshold[4] (0.5928) yield a large dominating cluster that extends or *percolates* across the map. Thus, all maps with $p = 0.62$ contain a large cluster that extends from one boundary to the other.

As p (the fraction of sites occupied by the habitat type of interest) increases, the amount of work in cluster identification and analysis changes. In developing kernels to be used by applications requiring cluster identification and analysis, the programmer should design methods that can adapt to the specific characteristics of any map.

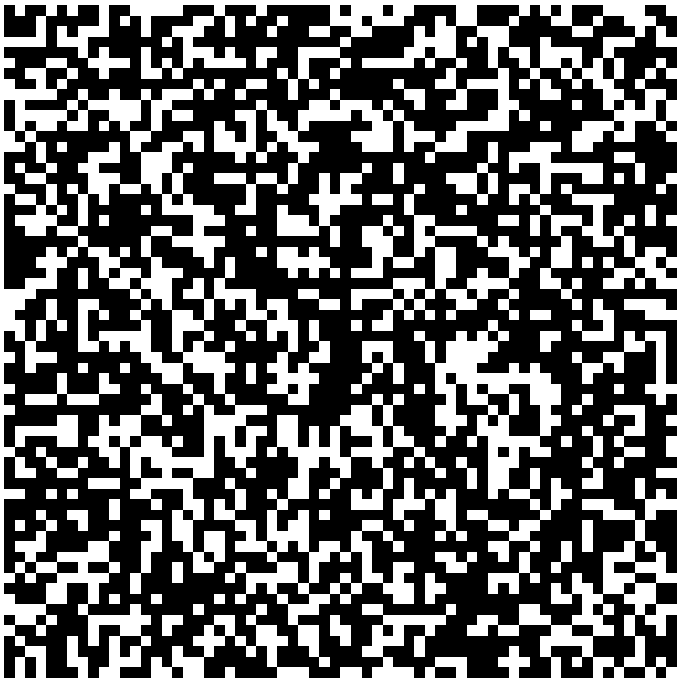
Figure 1.5 illustrates the differences in cluster density associated with $p = 0.3$ and $p = 0.62$ for 64×64 maps. The map with $p = 0.3$ (Figure 1.5(a)) has many small fragmented clusters, while the map with $p = 0.62$ (Figure 1.5(b)) has one large cluster that spreads across the map along with many smaller clusters.

[4]The critical threshold value varies with the neighbor rule employed for cluster definition. 0.5928 is the critical threshold when the 4-neighbor rule is used as described in Rule 1.

Table 1.1: Distribution of cluster sizes for three p-values.

		Size of		Cluster Size						
	Map	Largest	No. of		101-	501-	1001-	5001-	10001-	100001-
p	Size	Cluster	Clusters	<100	500	1000	5000	10000	100000	500000
0.1	64	8	325	325						
	128	6	1305	1305						
	256	7	5227	5227						
	512	12	20917	20917						
0.3	64	25	534	534						
	128	29	2157	2157						
	256	33	8484	8484						
	512	42	33891	33891						
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

(a)



(b)

Figure 1.5: Sample 64×64 random maps with $p = 0.30$ (a) and $p = 0.62$ (b).

Chapter 2

Methods

Initially, the original Fortran-77 code for map analysis was rewritten in C to take advantage of dynamic memory allocation and recursion. Next, cluster identification was studied and improved. Most efforts, however, focused on the parallelization of cluster geometry as it is the most time-consuming part of map analysis. Several different parallel models, including versions that additionally exploit vectorization were implemented and analyzed on the CM-5.

2.1 Improvements to the Sequential Code

As the Fortran-77 code was rewritten, several changes were made. Statically declared arrays were changed to dynamically allocated arrays which allowed for more efficient use of memory. In Fortran-77, the input data is normally read into `in_map` one element at a time. However, in C, the entire datafile is processed with one read function, `read`, into an (`nrows`) $\times$ (`ncols`) buffer. The data is then moved into the (`nrows` $+ 2$) $\times$ (`ncols` $+ 2$) array, `in_map`, and the border of -9's is added. Unlike the original Fortran-77 version, our C implementation can analyze rectangular (`nrows` $\neq$ `ncols`) as well as square maps.

Whereas the original Fortran version of cluster identification emulated recursion, two different versions of the functions, `clusterID` and `Site_Label`, were written in C: a true recursive version and an improved nonrecursive version. For all versions, traversal of the 2-D grid begins in `clusterID`. When a -1 is located (beginning of cluster), `Site_Label`, which is called by `clusterID`, is used to label the cluster and determine its size. After `Site_Label` completes cluster labeling, `clusterID` continues its traversal of the grid until the beginning of another cluster is located, at which time `Site_Label` is called again. This process continues until all pixels have been traversed and all clusters have been identified.

The original Fortran-77 and recursive/nonrecursive C versions of cluster identi-

fication all differ in the function `Site_Label`. In the original Fortran-77 version, `Site_Label` performed cluster labeling by determining if the NEWS neighbors of a pixel were also members of the cluster. If so, the coordinates of the pixel(s) were added to arrays of coordinates. The coordinate arrays are treated as queues in that all array members will have *their* neighbors checked in turn. If these neighbors are cluster members, their coordinates are also added to the queue. The coordinate arrays (queues) are processed in order so that the current pixel coordinates are always in the first position of the array, `array[0]`. After `array[0]` is processed, `array[1]` must move into `array[0]`. As a result, the whole array was moved up one position. For example, after each pixel is processed in `array[0]` the array was maneuvered as follows

```
for(i=0;i<arraysize;i++)
  { array[i] = array[i+1]; }
```

The shifting of elements in the coordinate arrays continues until the arrays are empty. Also, the original implementation of `Site_Label`, although quite modular, requires several function calls within its loops. The nonrecursive C version, on the other hand, in-lines[1] these smaller functions, and instead of using two coordinate arrays (queues), only one array of pointers representing the addresses of pixels is needed. In order to avoid direct manipulation of the array of pixel addresses, a pointer is used to move down through the queue of addresses to mark the current pixel to be processed. In the recursive C version, `Site_Label` is truly recursive. Starting with the south neighbor, if this neighbor is a member of the cluster, `Site_Label` increments the cluster size count by 1, labels the pixel, and calls itself, passing the address of the south neighbor to be the current pixel to be processed. The south neighbor of this pixel is now checked, and if the pixel does not have any neighboring member pixels, `Site_Label` returns, popping the stack so that the previous pixel is ready to have the east neighbor checked. This continues until all possible neighbors have been checked.

Table 2.1 lists six different cluster identification algorithms that were compared. With regard to performance (see Section 3.2), there is no significant difference between the best Fortran-77 and nonrecursive C algorithm, `FORT_1&2` and `C_NON_1&2`, and the recursive C algorithm, `C_REC`. Since the recursive C version is more compact, we select this version for our map analysis program. Prologues to selected functions are provided in Appendix D.

Table 2.1: Six cluster identification algorithms implemented.

Program Name	Function
`FORT_ORIG`	The original Fortran-77 program
`FORT_1`	Fortran-77 program with Modification 1*
`FORT_1&2`	Fortran-77 program with Modification 1 & Modification 2**
`C_NON_1`	C version of FORT_1
`C_NON_1&2`	C version of FORT_1&2
`C_REC`	Recursive C program

*Modification 1 : in-lining small, frequently called functions

**Modification 2 : elimination of unnecessary array manipulation

[1] The code of a function is included directly in the current program stream.

2.2 Parallel Models On The CM-5

All parallel models in this study were implemented on a Thinking Machine Corporation CM-5 having 32 processing nodes (PNs) interconnected via a *fat-tree* configuration ([Hwan93]). For space-sharing purposes, the PNs can be grouped into partitions with each partition under the control of a control processor (CP). On the CM-5 used in this study, the 32 PNs make up one partition under the control of one CP or partition manager ([TMC92c]). Each node is a 32 MHz Sun Sparc 2 processor with the addition of 4 vector units (VUs) positioned between the memory bank and the system bus (see Figure 2.1) ([TMC92d]). Each VU acts as the memory controller for 8 megabytes of memory which yields 32 megabytes of total memory per node, and each VU contains 128 single-precision (4-byte) registers that can be aligned to form vector registers of varying lengths and precision (e.g. eight 64-bit registers or sixteen 32-bit registers).

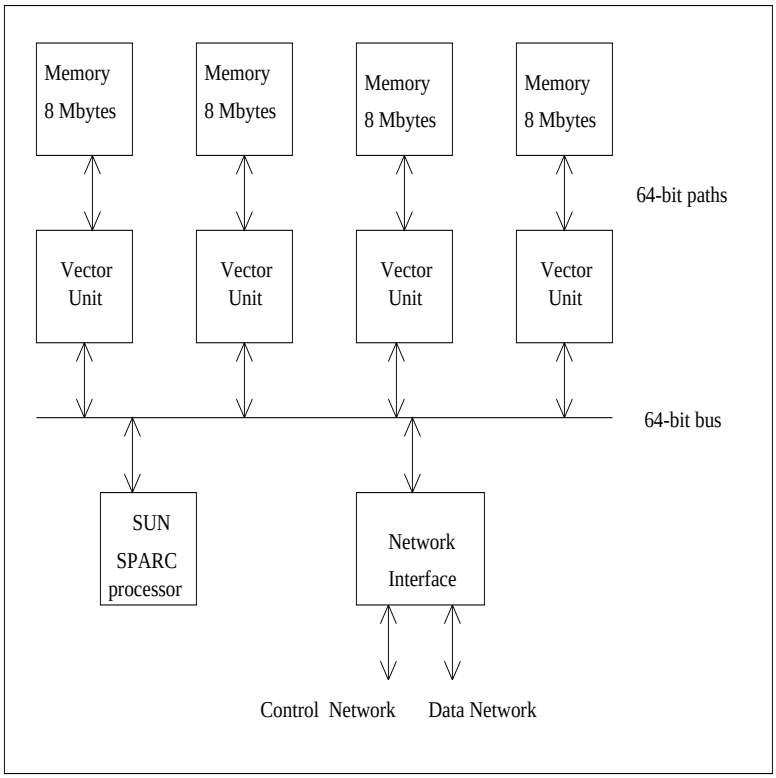


Figure 2.1: CM-5 processor node with vector units.

The CM-5 offers both *SIMD* and *MIMD* parallel programming models. All parallel programs developed in this study are *MIMD* and use message passing functions defined in the message passing library, *CMMD*. Two different models for message-

passing programs are available: host/node and hostless. The host/node model involves a host program that is separate from the node program. The host program, whose main role is normally I/O based, runs on the host node or control processor for that partition. Although the hostless model requires only node programs from the programmer, a host program (provided by the CMMD library) is still executed on the CP in order to act as an I/O interface for the nodes.

In this study, all programs are written in C with calls to CMMD message-passing routines. Both programming models, host/node and hostless, are implemented with the vector units exploited in the host/node model.

2.2.1 Host/Node Models

Several approaches to map analysis have been employed using the host/node model. In all cases, the same basic algorithm is used: the two programs continuously interact so that the node processors (PN's) always report to the host processor when they become idle. If there is still work to be done, the host processor will assign all reporting node processors more work. The nodes send their results back to the host processor, and receive their next assignment. This continues (without any global synchronization) until all work has been assigned. There is no interprocessor communication. Since sequential cluster identification (on a single node) has been observed to be very fast (requiring a very small percentage of the total map analysis time), no parallelization of cluster identification was implemented in these models. Hence, the focus of parallelization was computation of the geometrically weighted radius, with I/O and cluster identification done sequentially on the host processor.

Two different methods of parallelization have been implemented within a single *hybrid* strategy that can employ either method.

(i.) The first parallelized version is *fine-grained* in that all 32 processors of the CM-5 work together to resolve R_s^2 for a single cluster. The host processor must first build the x- and y-coordinate arrays of the relevant cluster and then broadcasts these arrays to all nodes. The host processor then assigns to a node processor one pixel of the cluster, and the node must determine the x- and y-coordinate differences of this pixel compared against all other pixels positioned below it in the array. The node processor squares all differences, adds them together, and sends the results back to the host processor. The host then assigns this node the next pixel to be resolved. The host continuously accumulates and sums partial τ values (1.2) for a cluster during this process. After all pixels have been assigned, the host calculates the final R_s^2 for that cluster. This process is then repeated for all clusters containing more than one pixel.

(ii.) The second parallelized version is *coarse-grained* in that each node processor resolves R_s^2 of an entire cluster by itself. The host processor broadcasts the map to all nodes. The nodes report to the host to get their cluster assignment, build the x- and y-

coordinate arrays for that cluster, and calculate its R_s^2. The node processors will then send the radius calculations back to the host which stores all radius calculations in an array. If there are more clusters to be resolved, the host sends the work to reporting node processors. This process is then repeated until all cluster radius computations are completed.

While both versions show good speed improvements for random maps, the coarse-grained method performs better for maps with many small clusters (i.e., $p = 0.1$ and $p = 0.3$). The fine-grained method, on the other hand, performs poorly on these particular maps. However, once there is a large dominating cluster, as is the case with maps having $p = 0.62$, the performance trends of the two methods reverse. The coarse-grained method in this case is comparable in execution time to the sequential C implementation on the Sun Sparc 2 workstation (see Section 3.3.1).

In order to take advantage of the optimal performance for each method of parallelization which will adequately handle different map and cluster characteristics, we have developed a hybrid method which can invoke either strategy depending on available cluster information. After cluster identification, the array containing the sizes of all clusters is partitioned according to a specified threshold value δ. All clusters with size s, where $s \leq \delta$, are resolved by the coarse-grain method, and all clusters with size s, where $s > \delta$ are resolved by the fine-grain method. For $200 \leq s \leq 1000$, either method works well, so δ is set at the average cluster size within this range.

2.2.2 Hostless Models

The hostless methods of parallelization were influenced by the cluster identification algorithms of Tamayo [FlTa92] for quantum physics applications on the CM-5. The first part of our parallel map analysis, cluster identification, is similar to Tamayo's algorithm with some major differences.

For cluster identification, Tamayo divides the 2-D grid into equal sub-grids across a 2-D grid of processors. For example, a CM-5 with 32 processor nodes (PN's) would be used as a 4×8 processor grid. Therefore, if a 64×64 grid is distributed across the 32 PN's, each PN would get a sub-grid of size 16×8. Any local cluster labeling method can be used within the sub-grids prior to global cluster labeling across nodes through a series of *relaxation cycles* [FlTa92].

In this study, the 32 PN's are utilized as a linear array rather than a grid (see Figure 2.2). Hence, the map is divided into 32 equal cross-sections such that PN 0 gets the top sub-grid whose dimensions are (`nrows`/32) $\times$ `ncols`. PN 1 would get the next subsection of the map, and so on until the bottom section of the grid is assigned to PN 31.

The data structure used to represent the sub-grids on each node is an array of *structs*, where each struct represents a pixel in the grid. The structs contain information as shown in Figure 2.3.

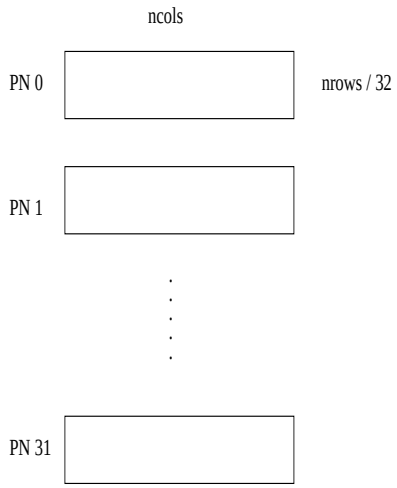


Figure 2.2: Division of data across the CM-5 processors.

Tamayo, on the other hand, uses arrays of integers to represent and store values of the sub-grid as well as some of the information listed in the struct above. The nodes obtain their data from the datafile (in order of their PN address). Using the I/O function,

`CMMD_set_open_mode`(CMMD_sync_seq),

which enables the nodes to get the same amount of data sequentially, PN 0 would get the first set of data, PN 1 the next, and so on.

In the Tamayo algorithm, any local cluster labeling method may be used within the sub-grids. The recursive cluster identification algorithm described in Section 2.1 can be used here with slight modification. Whenever a pixel is labeled as part of a cluster, its pointer, `head`, is set to point to the *head pixel* (the first occurrence of a pixel containing that label as the grid is traversed in row-major order). All label values for each node start at an offset related to the node address so there are no duplications of local cluster labels (see Figure 2.4(a)).

Next, the clusters must be re-labeled across the PN's to account for clusters that cross borders of sub-grids. Tamayo performs this re-labeling in *relaxation cycles* in which one cycle involves the following tasks:

1. Every node, except the last, loads an array of length `ncols` with the head pixel labels pointed to by the pixels in the bottom row of its sub-grid.

2. Loaded arrays are sent to the neighboring southern PN.

3. The receiving PN's (1-31) compare values of the array to the top row of their sub-grid and identify clusters traversing sub-grid boundaries. A nonzero value in

```
struct MAP {
        int label               /* cluster label */
        int local_size          /* local cluster size within PN */
        int cl_size             /* global cluster size across PN's */
        double radius           /* partial sum of τ */
        struct Map *head        /* pointer to head pixel */
        }
```

Figure 2.3: Data structure used to represent each site or pixel in the 2-D grid.

corresponding positions in the array and the top row signifies a cluster crossing PN boundaries. If the value in the array is smaller than its bordering pixel, then the head pixel's label pointed to by the bordering pixel is given that smaller value.

4. PN's (1-31) allocate an array of values containing the head pixel labels pointed to by the pixels in their top row. These arrays are then sent to their northern neighbor.

5. The northern neighbor PN's (0-30) receive the arrays and compare the values to the head pixel values pointed to by the pixels in their bottom row. If the incoming array values are smaller than the head pixels' values of relevant pixels, the values of the head pixels are changed to the smaller values.

We note that the Tamayo algorithm [FlTa92] allows for toroidal wrap, so all PN's participate in every step of the relaxation cycle. In our implementation, PN 0 does not communicate with PN 31, as the top and bottom of the map do not form shared borders. Thus, any communication sent to the southern PN cannot be initiated by PN 31, and any communication sent to the northern PN cannot be initiated by PN 0.

This cycle continues until no more changes are recorded. At this point, all head pixels contain the appropriate cluster labels for their clusters. The other pixels of the clusters must be re-labeled to reflect their appropriate labels. To accomplish this, each PN traverses its grid sequentially and changes all pixel labels to match those of the head pixel to which they point (see Figure 2.4(b)).

With all pixels properly labeled, global cluster sizes must be determined. This is done both locally and globally. Locally, there can be more than one cluster in a sub-grid with the same label but different head pixels and cluster sizes (see Figure

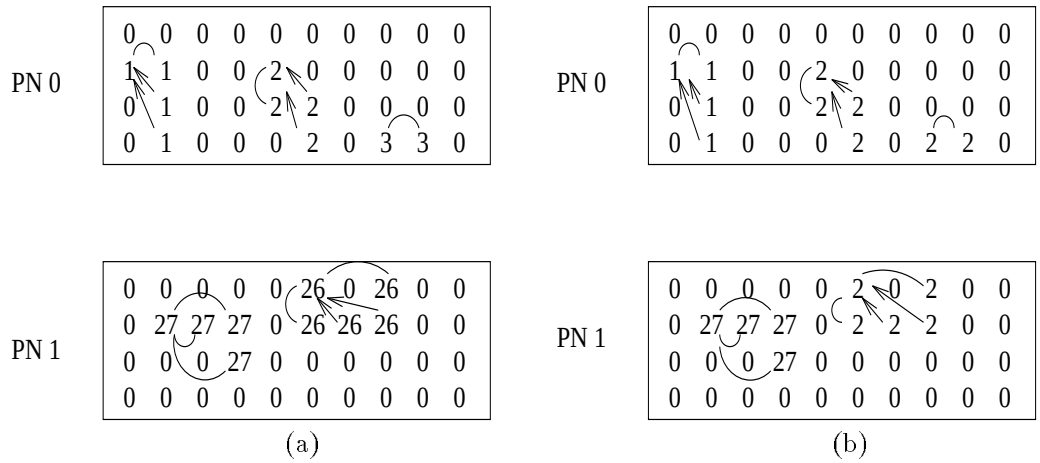


Figure 2.4: Local cluster identification (a) and global cluster identification with re-labeling(b).

2.4(b)). To gather similar clusters within a sub-grid, the PN's must traverse their sub-grids sequentially and locate all clusters with matching labels. When found, the cluster sizes are added to the total size for that cluster, and the head pixel pointers are reset to point to the unique head-pixel (see Figure 2.5).

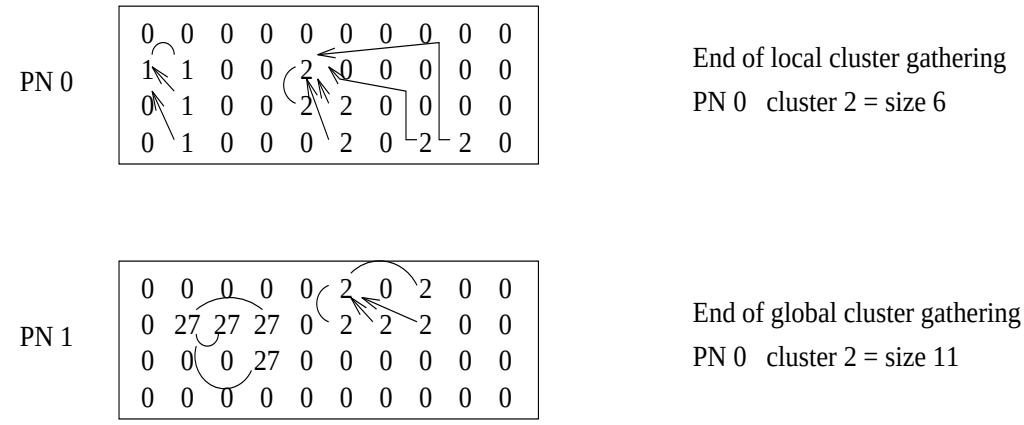


Figure 2.5: Cluster gathering.

Globally, clusters with similar labels across PNs must be located in order to determine cluster sizes. A PN is considered to *own* a cluster if the value of the cluster's label originated on that PN (again label values are offsets related to the PN addresses). Since all PN's can easily determine if a cluster belongs to them, each PN sequentially traverses its grid checking cluster ownership. If a cluster is found that is not owned by that PN, the PN loads a buffer containing the cluster label and local cluster size. This buffer is sent to the owner PN for that cluster using the asynchronous message

passing function,

CMMD_send_noblock(destination, tag, *buffer, buffer_desc).

Since each PN can possibly *own* clusters they must also be available to receive messages from other nodes. Upon receipt of a message, the PN adds the incoming cluster size to the relevant accumulating total cluster size and records it in an array of cluster sizes. After global gathering, all PN's will have an array of values that represent the sizes of clusters that originate in their sub-grid (see Figure 2.5).

At this point, cluster identification is complete and cluster geometry, R_s^2, must be calculated for each cluster. This is performed in two steps: (*i.*) local calculation of τ defined in (1.2), and (*ii.*) calculation and storage of the final mean squared radius, R_s^2, in an array by the PN that owns the cluster.

Two different methods of calculating τ are implemented according to the type of message passing used: synchronous and asynchronous. The first version which uses synchronous message passing allows for each PN, one at a time, to determine which clusters it owns and to have all other nodes look to see if they share these clusters. If shared, they determine their local τ for that cluster. An illustrative pseudo-code for computing τ in this manner is shown in Figure 2.6. The second

```
for i = 1,Number_of_PNs {
   if(my_address = i) {
      sequentially search through sub-grid
        for clusters originating here
      if(found) {
         broadcast cluster label to all other nodes
         determine local coordinate comparisons
       }
    }
   else {
      receive broadcast of cluster label from a node
      if(my_address > i) {
        look for that cluster in sub-grid
        if(found) {
          determine local coordinate comparisons
         }
       }
    }
  }
```

Figure 2.6: Pseudo-code for τ calculations using synchronous message passing.

method for calculating τ utilizes asynchronous message passing. In this case, each

PN independently traverses its sub-grid looking for head-pixels of clusters. When found, the PN determines if the cluster is non-local (it does not originate on that node). If it is non-local, the PN loads the x- and y-coordinate arrays, sends them asynchronously to the PNs that contain these clusters, and calculates and stores the local τ value. If the clusters are local (originate on that node and do not cross borders) to the node, no communication with other PNs is necessary, and τ is calculated and stored. Upon receipt of the x- and y-coordinates, a PN must determine the local τ comparing those coordinates to its own. A sample algorithm for computing τ with asynchronous message passing is provided in Figure 2.7.

```
for i = 1, gridsize {
  look for head-pixel of a cluster
  if (found) {
    load x,y-coordinates of cluster in arrays
    if (local_cluster) {
       determine local coordinate comparisons and store
     }
    else {
       send x,y-coordinate arrays to PN's between self
          and owner PN, inclusive.
       determine local coordinate comparisons
     }
   }
  if (incoming_message) {
     receive incoming x,y-coordinates
     determine local coordinate comparisons
   }
 }
while(!messages_done) {
  if (incoming_message) {
     receive incoming x,y-coordinates
     determine local coordinate comparisons
    }
 }
```

Figure 2.7: Pseudo-code for τ calculations using asynchronous message passing.

At this point, all pixels of similar clusters have been compared whether through synchronous or asynchronous message passing. However, partial τ sums are scattered across the PN's and must be gathered to calculate the final R_s^2 value for each cluster. This comprises the second step of cluster geometry, and is accomplished by a synchronous method similar to that of calculating partial τ values. However, reduction

functions are needed in order to sum across nodes, such as

`CMMD_reduce_double`(value,CMMD_combiner_dadd).

For example, at some point each PN will broadcast the labels of clusters it owns to all other PNs. All other nodes look for that cluster and set the value to be reduced equal to their local τ value, or to 0 if it is not located on that sub-grid. The reduction function is used to add these values globally, and the broadcasting PN uses the sum to calculate the final R_s^2 for that cluster which it subsequently stores.

Finally, each node determines its largest cluster and largest cluster R_s^2. The largest cluster and R_s^2 of the whole map are then determined using the global reduction functions,

`CMMD_reduce_int`(value, CMMD_combiner_max)

and

`CMMD_reduce_double`(value, CMMD_combiner_dmax),

respectively, to return the maximum values across PN's. The average cluster size is also calculated by dividing the total number of pixels labeled by the total number of clusters.

2.3 Addition Of Vector Units

The CM-5 architecture embodies another level of parallelization which can further enhance the machine's performance. Each PN contains four vector units (VUs) which collectively form a small SIMD machine with four nodes (see Section 2.2). Accessing the vector units using the current CMMD message-passing routines[2] is difficult. Our approach has been to write routines using both *Sparc* assembler and *DPEAC*, a pseudo-assembly language for generating vector instructions. All *DPEAC* code is compiled with the `dpas` assembler which preprocesses the *DPEAC* code into *Sparc* assembly code which is then run through the `as` (*Sparc*) assembler [TMC92d]. A sample DPEAC kernel is provided in Appendix E.

To exploit this lower level of parallelization, the VUs are used to compare the x- and y-coordinate arrays for each cluster (see Section 1.2.2). Before employing the VUs, we list three constraints that should be satisfied:

(*i.*) The x- and y-coordinates of each pixel must be compared to those of every other pixel in the same cluster,

(*ii.*) pixel coordinates should not be compared to themselves, and

(*iii.*) once the coordinates of a pixel have been compared, it is not necessary to compare them again. For example, if pixel 1 coordinates have been compared to those

[2]The current version is CMMD 2.0. The next release, CMMD 3.0, will allow the user to access the vector units through the use of CMFortran on the individual nodes.

of pixel 2, it is not necessary for pixel 2 coordinates to be compared to those of pixel 1.

This results in two basic scenarios for comparing vectors of array elements. The first scenario involves comparisons of similar subsets of pixel coordinates. Figure 2.8

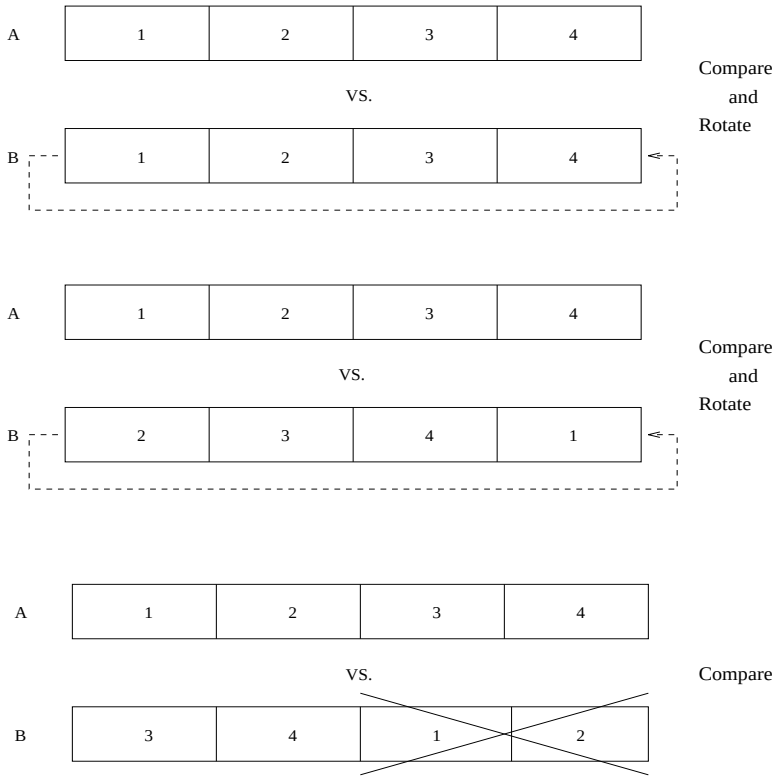


Figure 2.8: Comparing similar arrays of pixel coordinates. Each of the 4 subsets can represent one or more array elements.

illustrates the rotations and comparisons necessary to compare identical arrays `A` and `B`. The 4 subsets (1–4) of array element(s) can represent one or many elements. If an individual element is represented by each subset, then the first comparison shown is unnecessary. However, if the subsets represent more than one element, the initial comparison is needed so that the different elements of each subset are compared. In either case, three comparisons are made for the 4 subsets. In general, $n_r = n_s/2$, where n_r and n_s are the number of rotations and subsets, respectively. For the last rotation, we set $\tau = \tau/2$ to compensate for duplicate comparisons, as subset 1 is being compared to subset 3 twice and the same is true for subsets 2 and 4.

The second scenario involves comparisons of different vector subsets of pixel coordinates. Again, the subsets (1–4) shown in Figure 2.9 can represent one or more elements. In either case, 4 comparisons and 3 rotations are necessary for 4 differ-

ent subsets in order for all elements to be compared. Here, we have $n_c = n_s$ and $n_r = n_c - 1$, where n_c is the number of comparisons.

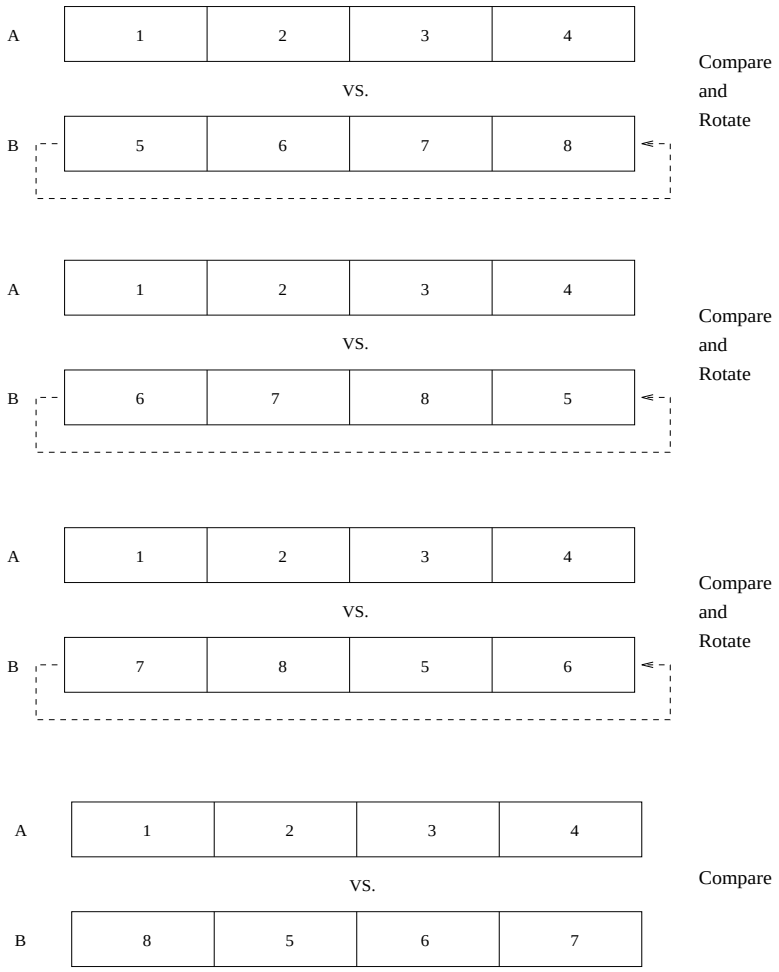


Figure 2.9: Comparing dissimilar arrays of pixel coordinates. Each of the 4 subsets can represent one or more array elements.

There are several different levels at which these arrays may be partitioned and compared: from the original vector of pixel coordinates down to the sets of the individual coordinates in the vector registers. At each level, whether comparing subsets or the individual pixel coordinates, scenarios 1 and 2 are relevant. Occassionally, in order to prevent extra array manipulation outside of the `DPEAC` kernels, some comparisons between similar pixels and duplicate comparisons between different pixels are made. However, in most of these cases, extra instruction cycles are not required, and the extraneous results are predictable and can be removed with one or two instructions.

2.3.1 Implementation of the VUs

The original x-coordinates array, `istack`, is created and loaded in the C code[3]. The `istack` array is copied twice into temporary arrays, `tmpx1` and `tmpx2`. When the `DPEAC` kernels are called, addresses of parameters, such as `tmpx1` and `tmpx2`, are stored in *SPARC* registers, and subsections of these arrays must be moved into each VU's memory space, referred to as the *parallel stack*. Using this stack, array elements are loaded into vector registers. Vector instructions can then be executed simultaneously on all or any combination of the four VUs.

Since there are four VUs or datapaths (dps), both arrays, `tmpx1` and `tmpx2`, are subdivided into 4 equal parts for each dp (see Figure 2.10). Specifically, dp0 will

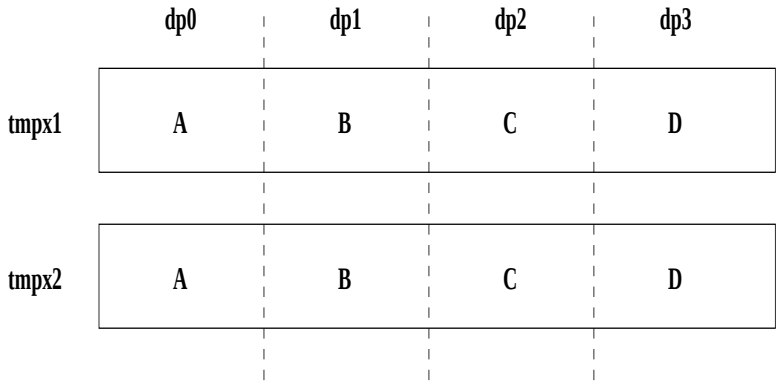


Figure 2.10: Division of vectors across datapaths.

handle all comparisons for section A, dp1 for section B, and so on. When all comparisons have been made using the `DPEAC` kernels, sections of `tmpx2` are rotated one partition in the C code. Each dp will then have a new subset of pixel coordinates on the subsequent call to the `DPEAC` kernel (see Figure 2.11). This process continues until all coordinate subsets, A thru D, have been compared against each other.

After comparing similar subsets, the array, `tmpx2`, must be rotated and compared only twice, and only half of the last rotation is included as described in scenario 1 in the previous section.

The next layer of "dividing/comparing" subsets of pixel coordinates is performed on each dp. Each dp is assigned a subset of `tmpx1` and `tmpx2` where all elements of these subsets must be compared. Three different `DPEAC` kernels have been written to make these comparisons: `RAD1, RAD2`, and `RAD3` (see Figures 2.12 – 2.14). The `DPEAC` routine `RAD1` (see Appendix E) determines τ when comparing similar coordinate subsets (scenario 1, see Figure 2.12), and `RAD2` determines τ for different vector segments (scenario 2, see Figure 2.13). The `RAD3` routine used to compare two coordinate

[3]All methods described for the x-coordinates array, `istack`, also apply for the y-coordinates array, `jstack`.

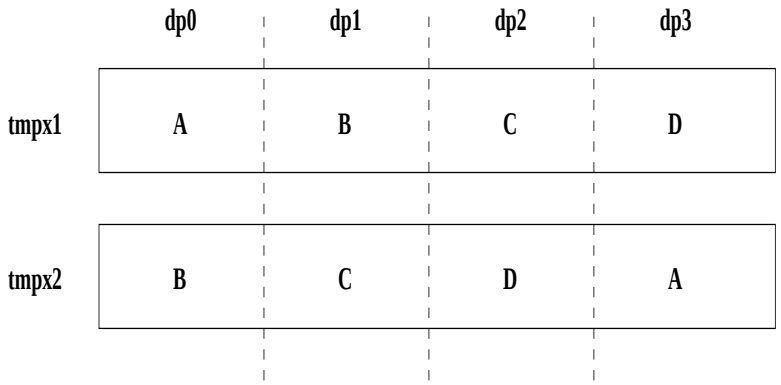


Figure 2.11: Division of vectors across datapaths after one rotation.

subsets of different sizes and elements is shown in Figure 2.14.

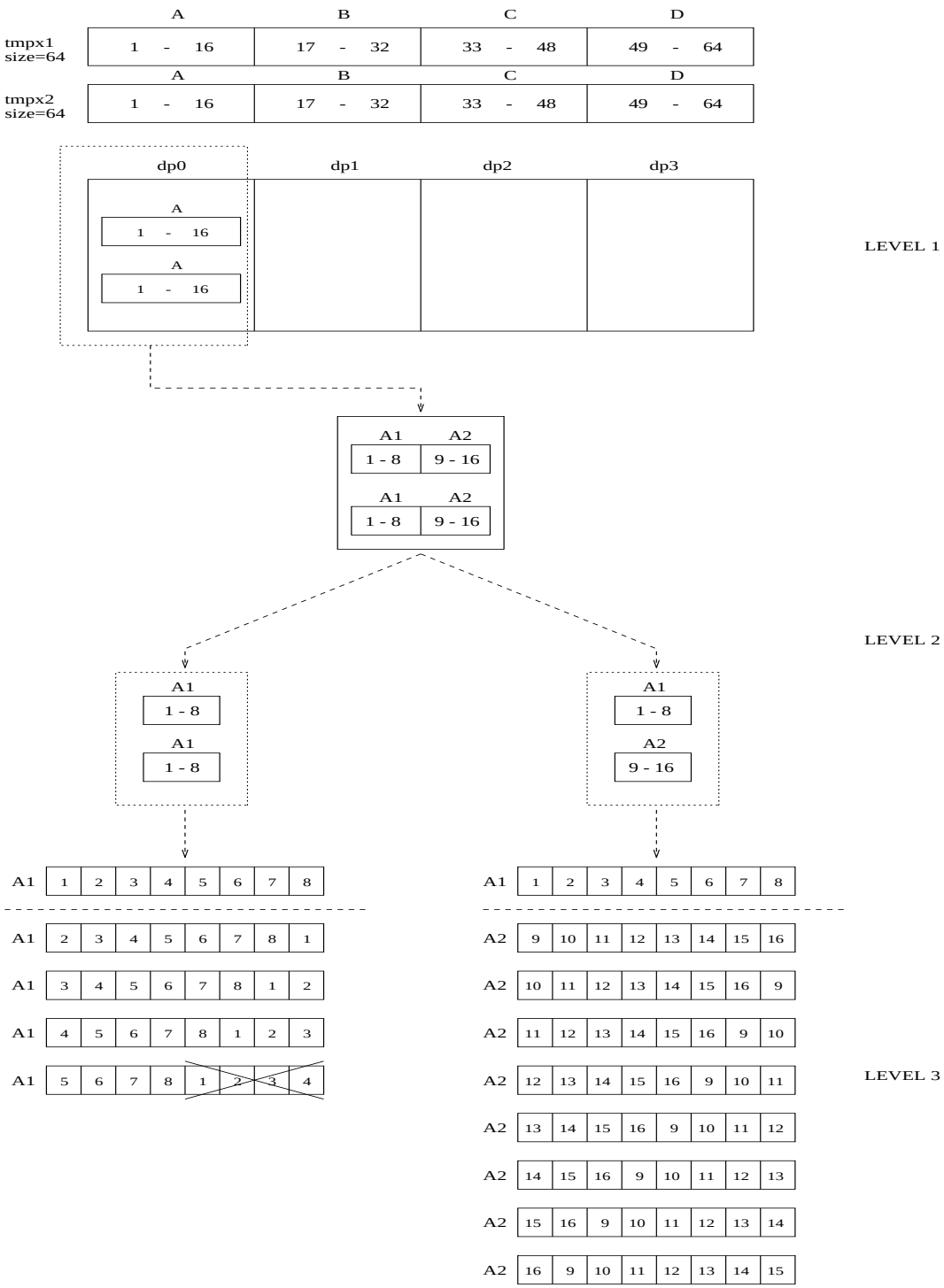


Figure 2.12: Comparing similar subsets of pixel coordinates in `DPEAC` kernel `RAD1`. The comparison of A2 vs. A2 is not shown, but is similar to A1 vs. A1.

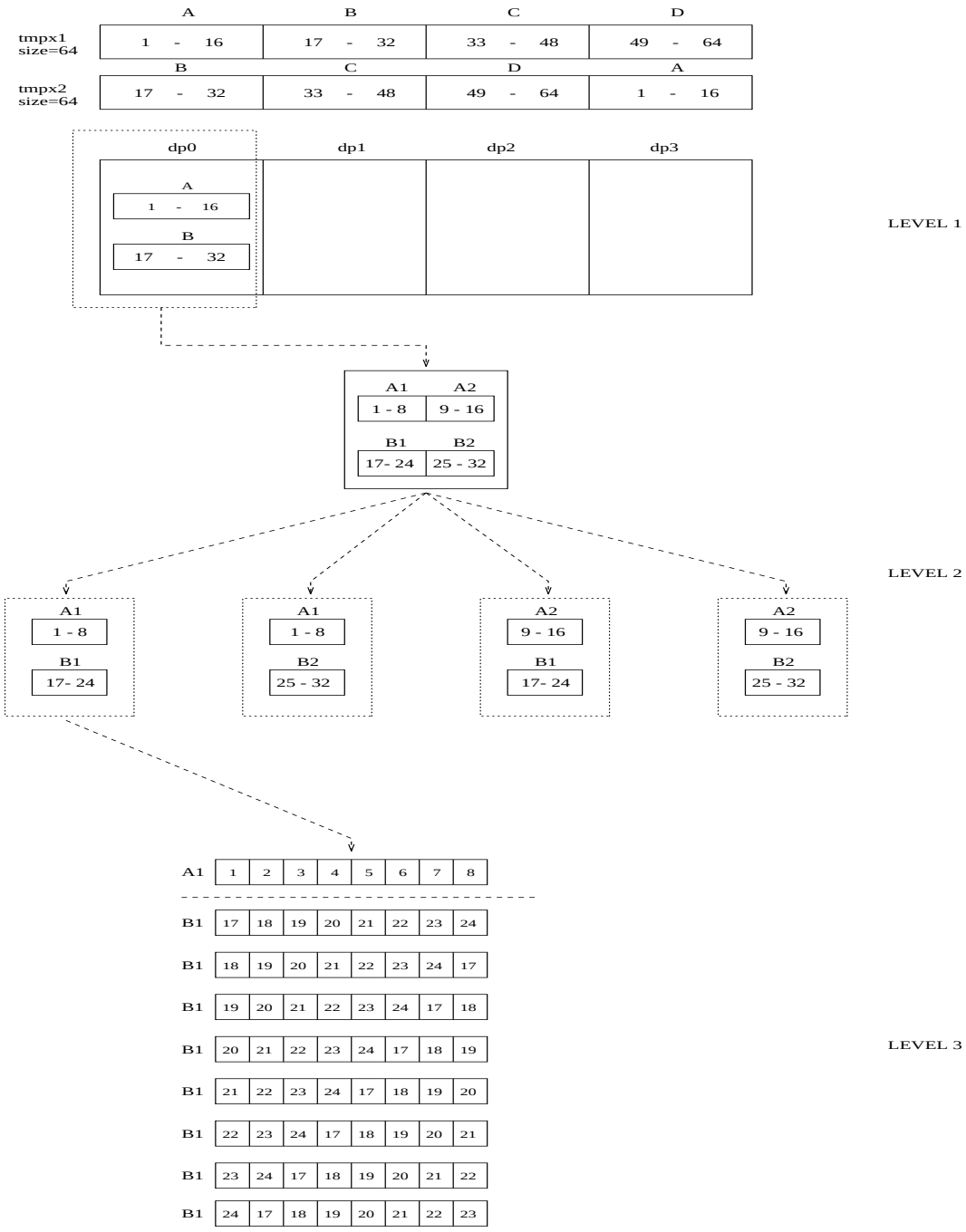


Figure 2.13: Comparing different subsets of pixel coordinates in `DPEAC` kernel `RAD2`.

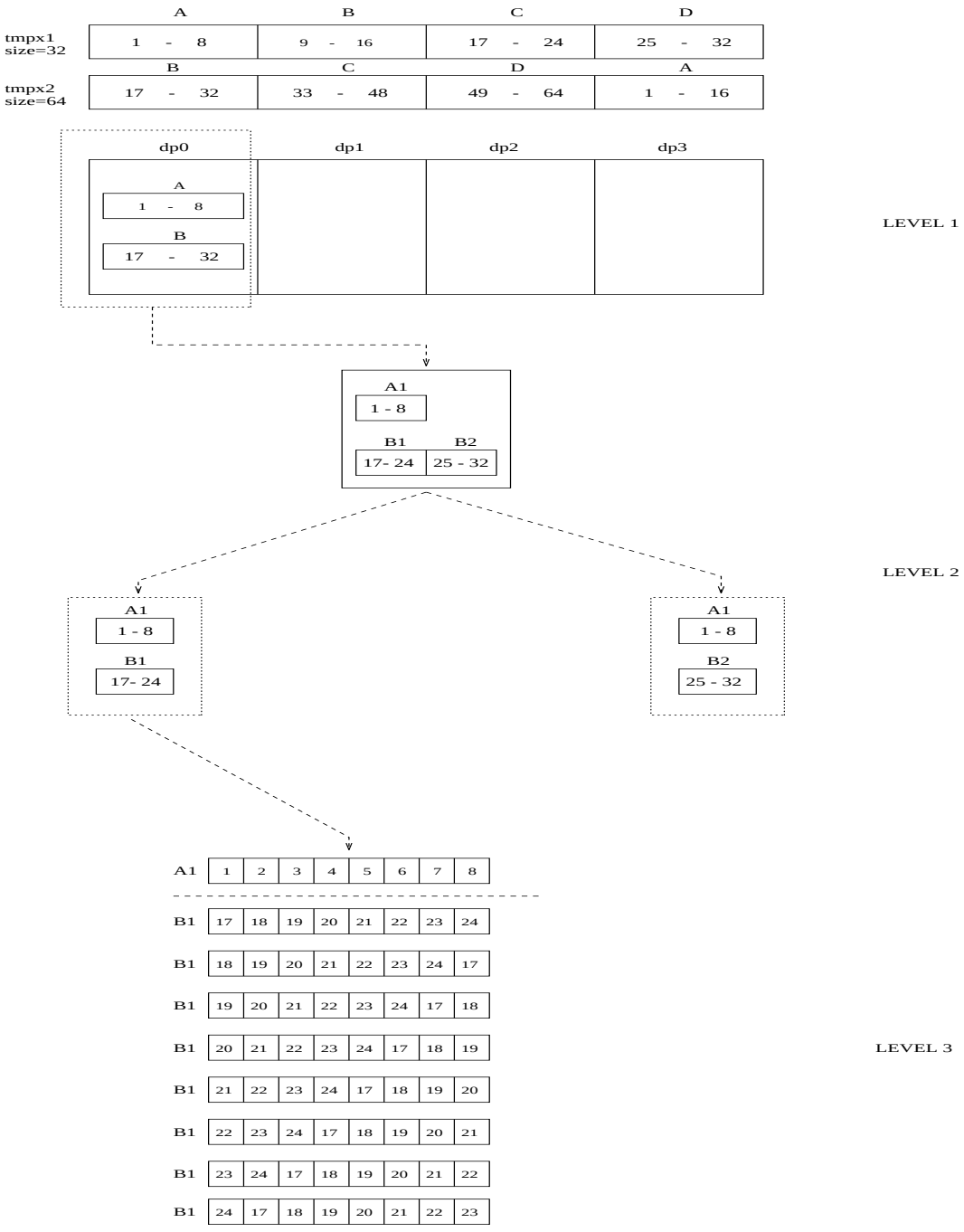


Figure 2.14: Comparing different subsets of pixel coordinates of different sizes in `DPEAC` kernel `RAD3`.

In all cases, each dp gets one-fourth of the original vector passed to the DPEAC kernel (see LEVEL 1 in Figures 2.12 – 2.14). Here, the vector registers contain 8 double precision elements. Therefore, the number of segments of 8, SEGS, contained in each subarray must be calculated to determine how many times the vector registers must be loaded and compared (see LEVEL 2 in Figures 2.12 – 2.14). After the registers have been filled, each element of the registers must be compared to the others. These comparisons are made by rotating the elements of the vector registers following the defined rules and scenarios discussed earlier (see Level 3 in Figures 2.12 – 2.14). When comparing similar vectors in RAD1, there is one comparison of similar pixels plus one duplicate comparison. The extraneous results are predictable and are easily removed with one subtract and one divide instruction. These two DPEAC instructions are less costly than the extra array manipulations needed in the C code to avoid the extra comparisons. If these extra comparisons were eliminated from the DPEAC kernels, the program would have to return from the DPEAC functions, manipulate the arrays in the C code (which can get time-consuming for large arrays), and call the DPEAC routines again and accrue the overhead of reloading the *parallel stack*.

2.3.2 Implementation of the DPEAC Kernels

The DPEAC kernels, RAD1, RAD2, and RAD3, have been implemented with the hybrid host/node model. Wherever RAD1 and RAD2 are both used in the coarse-grained portion, all three kernels are used in the fine-grained portion. The coarse-grained portion allows each PN to resolve R_s^2 for a cluster (see Section 2.2.1). Thus, after finding its assigned cluster in the grid, the node fills the x- and y-coordinate arrays prior to using the vector units to resolve τ.

The fine-grained portion, on the other hand, only assigns one pixel to each PN to resolve part of τ (see Section 2.2.1). Using the VUs is this case is not efficient, because there is not enough work performed on a node to warrant the overhead of the VUs. As a result, the fine-grained method has been modified to allow more efficient vectorization. To allow more work per node without the need of communication with the host, each PN is assigned a subset of the original x- and y-coordinate arrays that is relative to its PN address. Each PN is responsible for comparing the elements in that array subset to themselves and to all remaining elements in the original array. For instance, say the coordinate array has n elements, and PN 0 gets a subset that includes elements 1 thru 10. PN 0 must compare elements 1 thru 10 to elements 1 thru n. If PN 1 has a subset which includes elements 11 thru 20, it must compare 11 thru 20 to elements 11 thru n. In this case, DPEAC kernels RAD1 and RAD2, are used to compare the PN subset to itself (section A, in Figure 2.15), and DPEAC kernel, RAD3, is used to compare the PN subset to the remaining elements (section B, in Figure 2.15).

With this new division of labor, the only communications necessary are the original

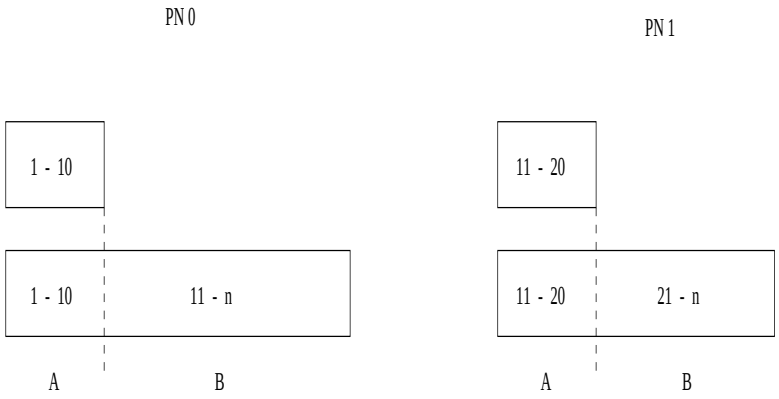


Figure 2.15: Division of segments in PN 0 and PN 1.

broadcast from the host to the nodes of the x- and y-coordinate arrays and a reduction of R_s^2 back to the host when all nodes are finished with their segments. The reduce function,

`CMMD_reduce_to_host_double`(r2,CMMD_combiner_dadd)

adds all results to get τ, and the host determines the final R_s^2.

In both methods, coarse- and fine-grained, the `tmpx` arrays sent to the `DPEAC` kernels are filled with the number of elements (equal to a multiple of 32) closest to the original coordinate array length, n. Hence, the length of `tmpx`, vl, is given by,

$$vl = n/32 \times 32 \ .$$

This allows the 8-element registers on the 4 dps to remain full. Any *strip-mining* of the leftover elements from the coordinate arrays (ranging from 0 thru 31 elements) are resolved sequentially in the C code.

2.4 Load Balancing In The Fine-Grained Model

Since the coordinate arrays are divided among the PNs in the fine-grained method that exploits the VUs, it is important to divide the arrays so that each PN gets an equal amount of work. As described in the previous section, the position of the subset of the coordinate array assigned to a PN is relative to the PNs address, and that subset will be compared only to the array elements whose array positions are equal to or greater than those in the subset (see Figure 2.15). As a result, the subset of elements that PN 31 must compare its subset against is much smaller than the other PNs, while PN 0 will compare against the largest subset. It is important, therefore, to divide the subsets across the PNs so that the lower numbered PNs get smaller subsets than the higher numbered PNs, allowing each PN to make equal numbers of

comparisons. In order to ensure such a division of work, a gradient-like segmentation is required (see Figure 2.16).

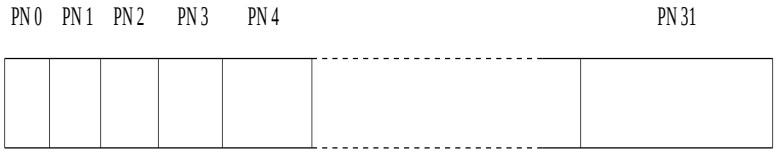


Figure 2.16: Segmentation of work for load-balancing.

Three different methods for load-balancing were implemented. The first attempt, `UNBAL`, was a quick and easy method which did not balance the work very accurately, but was still better than merely dividing the coordinate arrays equally across the PNs. In the `UNBAL` algorithm, the coordinate array is halved, and the first half is divided equally across the first 20 PNs (0 – 19). The second half is divided equally across the last 12 PNs (20 – 31). For example, the subset lengths, k, for PNs 0 – 19 given a vector size of n are defined by

$$k = \frac{n/2}{20} = \frac{n}{40}, \tag{2.1}$$

and subset lengths, l, for PNs 20 – 31 are given by

$$l = \frac{n/2}{12} = \frac{n}{24}. \tag{2.2}$$

All leftover elements are assigned to PN 31. Therefore, 20 nodes have subset sizes = k, and 11 nodes have subset sizes = l where $l > k$. One node, PN 31, gets a subset size, m, where $m \geq l$, depending on the number of leftover elements it receives.

All parallel models with the implementation of the VUs in the fine-grained portion use `UNBAL`, and the initial performance results are excellent (see Results, Section 3.4), but to ensure a more balanced work load, two additional load balancing algorithms (`BAL` and `BAL32`) were also developed.

In balancing the work across the PNs, we refer to work as total number of comparisons necessary. The actual subset size of the arrays that each PN obtains is based on the number of comparisons per node. The number of total comparisons, C_A, needed for vector A of length n (see Figure 2.17) is

$$C_A = \frac{n(n-1)}{2} \ ,$$

and the number of comparisons per node, C_{pn}, is defined by

$$C_{pn} = \frac{C_A}{P} \ , \tag{2.3}$$

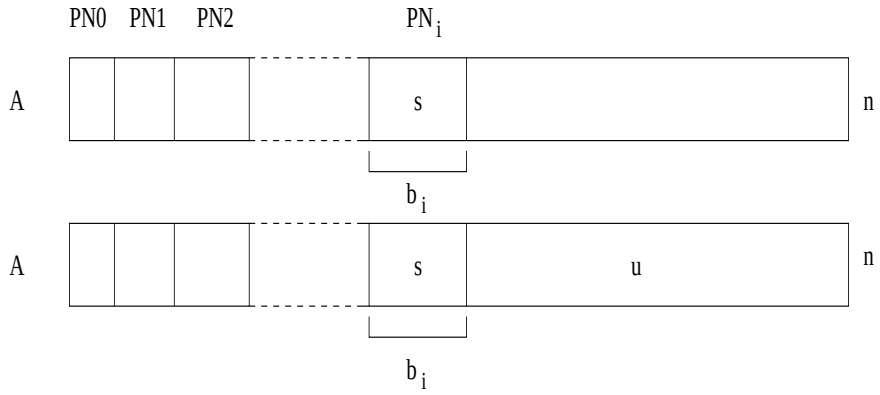


Figure 2.17: Array subsets and comparisons per PN.

where P is the number of PNs.

Referrng to Figure 2.17, C_{pn} can also be defined in terms of the subset size b_i for the i-th PN, so that C_{pn} is equal to the number of comparisons between sections s and itself, $\bar{c}_1$, plus the number of comparisons between sections s and u, $\bar{c}_2$. Hence, we have

$$C_{pn} = \bar{c}_1 + \bar{c}_2 \ , \tag{2.4}$$

where

$$\bar{c}_1 = \frac{b_i(b_i - 1)}{2} \ , \quad \text{and} \quad \bar{c}_2 = b_i(n - \sum_{j=0}^{i} b_j) \ .$$

From (2.3) and (2.4), it follows that

$$\frac{b_i(b_i - 1)}{2} + b_i(n - \sum_{j=0}^{i} b_j) = \frac{C_A}{P}.$$

It is easy to show that the subset size b_i for the i-th PN satisfies

$$b_i^2 - \beta b_i + \gamma = 0, \quad \text{where} \quad \beta = \begin{cases} 2(n - \sum_{j=0}^{i-1} b_j) - 1, & for \ \ 0 < i \ < P - 1, \\ 2n - 1, & for \ \ i = 0, \end{cases} \tag{2.5}$$

$$\gamma = \ 2C_{pn}, \qquad for \ \ 0 \leq i \ < P - 1$$

$$\text{and} \qquad b_i = n - \sum_{j=0}^{i-1} b_j, \quad for \ \ i = P - 1.$$

For $0 \leq i < P - 1$, we solve (2.5) and choose b_i to be the smaller of the two solutions (if they are real). For complex solutions to (2.5), we set $b_i = \beta/2$.

For BAL, the first 28 PNs use $\lceil b_i \rceil$ while the last 4 PNs round their b_i to the nearest integer resulting in a balanced work load across the processors. However, the performance results for BAL are not competitive with those obtained with UNBAL. These results are due to the amount of strip-mining that takes place per node (i.e., only subset sizes that are multiples of 32 are processed via the VUs). Since the leftover elements are resolved sequentially, the cost of strip-mining increases, especially for the lower numbered PNs which will have more comparisons to process sequentially.

In an attempt to eliminate the strip-mining problem in BAL, with BAL32 we allowed each PN (except PN 31) to have a subset of pixel coordinates that is a multiple of 32. In other words, the first 12 PNs are given $32 \times \lfloor b_i/32 \rfloor$ coordinate pairs while the last 20 PNs are assigned $32 \times \lceil b_i/32 \rceil$ coordinate pairs with any remainder pairs of coordinates given to PN 31. When compared to BAL, the performance of BAL32 is much better, but it was still not comparable to UNBAL. Even though strip-mining was eliminated for the subset of coordinates assigned to each PN (section A, Figure 2.15), strip-mining is still necessary for section B. Thus, the relationship of subset sizes and strip-mining affects the performance of the load balancing algorithms. There are vector lengths for which UNBAL performs worse than the other two algorithms, however, UNBAL achieves the best performance overall (see Table B.1).

2.5 Thresholds

After adding the VUs to the host/node hybrid program, HN_HY_VU, R_s^2 can now be determined by a coarse-grained and/or fine-grained method, and either method can be used with or without the VUs (see flowchart in Appendix C). Therefore, there are four different contexts in which HN_HY_VU can be implemented: coarse-grained (with or without VUs) or fine-grained (with or without VUs). We consider 3 thresholds (see Figure 2.18) to determine which context is appropriate for a particular vector of pixel coordinates.

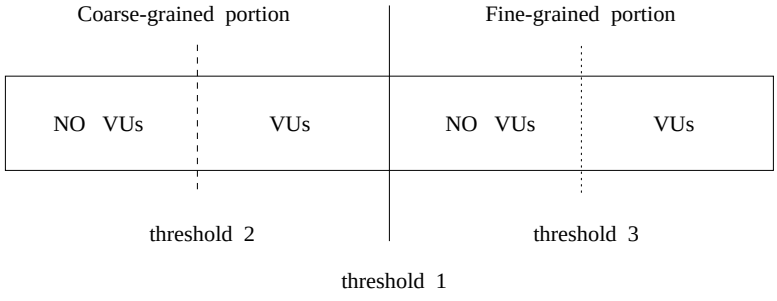


Figure 2.18: Three thresholds in HN_HY_VU program.

The setting for threshold 1 (coarse-grained or fine-grained) is determined dynam-

ically at runtime as described in Section 2.2.1. Threshold 2 (with or without VUs for the coarse-grained portion) was determined by comparing the performance of the coarse-grained method with and without the VUs in order to find the minimal vector length which warrants use of the VUs. Figure 2.19 shows that even with the effects of strip-mining (sequentially resolving leftover elements) the cross-over point is at vector length 64. Therefore, threshold 2 is set at 64 which corresponds to 2016 pixel coorinate comparisons. Threshold 3 (with or without VUs for the fine-grained portion) is set at 32, which is the minimum vector size that can be processed with the `DPEAC` kernels. Though this vector length would seem to allow for too few comparisons ($C_{pn} < 2016$), all PNs, except PN 31, will be comparing their subset of pixel coordinates to a larger subset, surpassing the minimum number of comparisons needed.

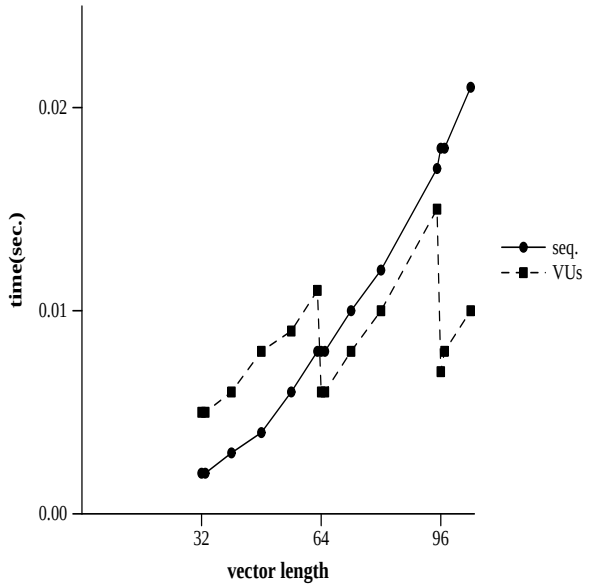


Figure 2.19: VUs vs. sequential time showing effects of strip-mining.

2.6 Model Comparisons

For the sequential cluster identification algorithms, the differences in performance are negligible. The recursive code is more compact than the nonrecursive and for that reason is more desirable. However, when working with extremely large maps, the largest cluster can be large enough so that the recursive algorithm may exhaust all available stack space. To conserve memory when resolving large maps, modifying the original `Site_Label` algorithm which manipulates the coordinate arrays, to always

process the pixel in position `array[0]` (see Section 2.1) may be more suitable (but much slower).

The host/node parallel programs are perhaps the most versatile of the parallel models. They are the easiest to implement, and are portable to other MIMD machines (e.g., Intel Hypercube). These models can resolve maps of various sizes and sustain a consistent performance rate across all map types. The host/node hybrid model employs dynamic load-balancing and exploits vectorization on the PNs. A disadvantage of these models is the amount of memory required for very large maps. The host node must keep a copy of the entire map in its memory space, and when resolving clusters in a coarse-grained fashion, the map is broadcasted to all nodes.

The hostless models, on the other hand, are more suitable when processing larger maps under memory limitations. Here, the map is sub-divided across all PNs, and hence, no PN contains the complete grid in its memory space. However, for the maps used in this study, these particular models were not competitive (less portable and inconsistent performance across map types). Also, load balancing the hostless models is a difficult problem in that the work-load is data-dependent and not dynamic.

Exploiting the vector units can be difficult when using CMMD message-passing. However, an investment in the design and implementation of `DPEAC` kernels have yielded excellent speed improvements over sequential implementations.

Chapter 3

Results

In this chapter, we discuss our methodology for program implementations and performance evaluation. This methodology includes representations of programs with and without VUs, data representation and I/O, program execution, program timing, and speed improvement calculations. The results were collected for sequential algorithms, parallel host/node programs, parallel hostless programs, parallel host/node programs with VUs, load-balancing algorithms, sequential programs on various workstations, and the parallel host/node hybrid program with VUs (on a real map). Table 3.1 summarizes the various programs implemented in this study and their functions.

3.1 Methodology

Before discussing specific results, we review some details of our program implementations and how timing information was collected.

All performance measures for parallel models without VUs (`HN_CG, HN_FG`, and `HN_HY`) were gathered using integers for most of the R_s^2 computations. However, since the memory alignment in the VUs are 64-bits, all R_s^2 variables were promoted to doubles to use the VUs more efficiently. Also, all performance measures on models with VUs (`HN_CG_VU, HN_FG_VU`, and `HN_HY_VU`) were collected from runs of `HN_HY_VU`. The three thresholds discussed in Section 2.5 for context switching in `HN_HY_VU` between `HN_CG_VU` and `HN_FG_VU` are shown in Table 3.2.

All data files containing the 2-D grids to be analyzed are converted from *ASCII* format to binary before implementation. Sequential and host/node parallel models read the data file in using the low level function `read`. In hostless models, the PNs retrieve their own sub-grid of data from the data file via

`CMMD_set_open_mode`(CMMD_sync_seq)

followed by a call to the intrinsic C functions `open` and `read`. In all programs, cluster information and timings are written to `stdout`.

Table 3.1: Map analysis software.

Sequential Programs

Program Name	Function
FORT_ORIG	Original cluster identification
FORT_1	Cluster identification with Modification 1*
FORT_1&2	Cluster identification with Modification 1 & Modification 2**
C_NON_1	Cluster identification with Modification 1
C_NON_1&2	Cluster identification with Modification 1 & Modification 2
C_REC	Recursive cluster identification
C_MAP	Complete map analysis

*Modification 1 : in-lining small, frequently called functions

**Modification 2 : elimination of unnecessary array manipulations

Parallel Programs

Program Name	Function
HN_CG	Host/node coarse-grained model without VUs
HN_FG	Host/node fine-grained model without VUs
HN_HY	Host/node hybrid model without VUs
HL_SYNC	Hostless model with synchronous message passing
HL_ASYNC	Hostless model with limited asynchronous message passing
HN_CG_VU	Host/node coarse-grained model with VUs
HN_FG_VU	Host/node fine-grained model with VUs
HN_HY_VU	Host/node hybrid model with VUs

DPEAC kernels

Program Name	Function
RAD1	Comparisons of similar coordinate subsets
RAD2	Comparisons of different coordinate subsets
RAD3	Comparisons of different coordinate subsets of unequal lengths

Load Balancing Algorithms

Program Name	Function
UNBAL	Unbalanced algorithm
BAL	Balanced algorithm
BAL32	Balanced algorithm with partitions in multiples of 32

Table 3.2: Threshold settings in HN_HY_VU

Threshold	HN_CG_VU	HN_FG_VU	HN_HY_VU
1	> largest cluster	40	determined dynamically
2	64	64	64
3	not applicable	32	32

Elapsed CPU time for sequential cluster identification algorithms was obtained using the function `getrusage`. Elapsed wall-clock time for host/node models and the sequential map analysis program, `C_MAP`, was obtained using the function `gettimeofday`. For the hostless models, elapsed time was represented by the node busy time obtained via the node timers. Although such times are not exactly wall-clock time, they are comparable to the times gathered by the system command `time`. In fact, we observed discrepancies in these timers of only 4-6% for the larger maps.

Speed improvements, S_i, were determined by

$$S_i = \frac{T_{seq}}{T_{par}}$$

where T_{seq} = sequential time and T_{par} = parallel time. All sequential timings were obtained on the front-end of the CM-5, a Sun Sparc 2, unless stated otherwise.

3.2 Sequential Algorithms

The elapsed CPU times of six different cluster identification algorithms are included in Table 3.3: 3 Fortran-77 and 3 C programs. `FORT_ORIG` represents the original fortran code, and `FORT_1` and `FORT_1&2` are optimized versions of `FORT_ORIG` using Modification 1 and Modifications 1 and 2, respectively, discussed in Section 2.1. We recall that Modification 1 involves function in-lining, while Modification 2 involves the elimination of unnecessary array manipulations. When comparing times[1] between `FORT_ORIG` and `FORT_1&2` on the 768×768 map, the modifications improve the performance by a factor of 739. `C_NON_1` and `C_NON_1&2` are the C analogs of the `FORT_1` and `FORT_1&2` programs. Although the two C programs are somewhat slower but comparable to their respective Fortran-77 programs for maps having p-values of 0.1 and 0.3, the C programs can be as much as 2 times faster for maps having p-values of 0.62. Finally, `C_REC`, the true recursive version of cluster identification, is the fastest version for the lower p-values, but is slower than `FORT_1&2` and `C_NON_1&2` for $p = 0.62$ due to a rapidly increasing stack size when resolving the large clusters. However, the time difference is negligible when the radius computations are introduced.

[1] Resolving the 1024×1024 map with $p = 0.62$ using `FORT_ORIG` exhausted a few hours of elapsed wall-clock time before it was manually terminated.

Table 3.3: Total CPU times for cluster identification algorithms.

	FORT_ORIG Original			FORT_1 Modification 1*			FORT_1&2 Modification 1 & 2**		
	p-value			p-value			p-value		
Map Size	0.1	0.3	0.62	0.1	0.3	0.62	0.1	0.3	0.62
64	0.01	0.02	0.13	0.01	0.01	0.04	0.01	0.01	0.02
128	0.02	0.05	1.22	0.02	0.05	0.19	0.02	0.05	0.07
256	0.11	0.24	22.82	0.09	0.17	1.63	0.07	0.16	0.28
512	0.40	0.88	281.19	0.33	0.67	11.41	0.32	0.62	1.16
768	0.90	2.12	1878.32	0.76	1.51	40.65	0.76	1.46	2.54
1024	1.63	3.76	——	1.35	2.71	95.76	1.33	2.50	4.69

	C_REC Recursive			C_NON_1 Modification 1*			C_NON_1&2 Modification 1 & 2**		
	p-value			p-value			p-value		
Map Size	0.1	0.3	0.62	0.1	0.3	0.62	0.1	0.3	0.62
64	0.00	0.10	0.10	0.01	0.01	0.02	0.01	0.01	0.01
128	0.01	0.02	0.08	0.03	0.05	0.08	0.03	0.05	0.03
256	0.05	0.09	0.32	0.10	0.17	0.73	0.13	0.20	0.16
512	0.19	0.34	1.37	0.46	0.70	5.53	0.49	0.75	0.58
768	0.43	0.77	3.08	1.02	1.69	19.00	1.09	1.75	1.53
1024	0.79	1.35	5.37	1.80	2.95	45.35	1.91	3.44	2.72

*Modification 1 : in-lining small, frequently called functions

**Modification 2 : elimination of unnecessary array manipulation

Table 3.4 shows the results for the sequential map analysis program (C_MAP) on the front-end of the CM-5 (Sun Sparc 2). The C_MAP program includes both cluster identification and cluster geometry, and timings were only collected for maps of size 512×512 or smaller. We use these particular results for the speed improvement rates associated with the parallel models.

Table 3.4: Total wall-clock times (sec.) for sequential map analysis, C_MAP, on a Sun Sparc 2.

	Sequential Map Analysis - C_MAP		
	p-value		
Map Size	0.1	0.3	0.62
64	0.12	0.38	4.73
128	1.48	5.75	75.29
256	25.65	102.64	2036.45
512	408.81	1617.47	32830.08

3.3 Parallel Models

In this section, the performance results for the parallel host/node programs (HN_CG, HN_FG, and HN_HY) and the hostless programs (HL_SYNC and HL_ASYNC) are compared. The performance of the host/node hybrid program (HN_HY) is also compared to that of the hostless programs. A complete set of performance results are included in Appendix A.

3.3.1 Host/Node Methods

In Table 3.5, we list the total wall-clock times for the host/node coarse-grained, fine-grained, and hybrid programs, HN_CG, HN_FG, and HN_HY, respectively.

Table 3.5: Total wall-clock times (sec.) for parallel host/node models without VUs.

	HN_CG			HN_FG			HN_HY		
	p-value			p-value			p-value		
Map Size	0.1	0.3	0.62	0.1	0.3	0.62	0.1	0.3	0.62
64	1.47	1.68	6.69	1.33	2.47	3.44	1.45	1.54	2.96
128	2.94	3.10	57.80	3.74	11.57	9.69	2.54	3.04	7.24
256	5.42	8.43	1747.04	32.22	127.73	82.00	5.28	8.01	61.78
512	18.32	52.36	27916.48	440.61	1784.48	1158.78	19.19	52.25	925.47

For the less dense maps ($p = 0.1,\ 0.3$) HN_CG clearly out performs HN_FG and

obtains speed improvements[2] of 31, for the 512×512 map with $p = 0.3$. HN_FG is in fact slower than the sequential program, C_MAP, for these lower p-values. On the other hand, once the percolation threshold (see Section 1.3) is crossed, as is the case with the $p = 0.62$ maps, HN_FG becomes the better method of parallelization, resulting in speed improvements as high as 28 for the 512×512 map with $p = 0.62$. For this same map, HN_CG is now as slow as C_MAP. The reason for the opposing behavior between HN_CG and HN_FG is primarily due to the cluster sizes above and below the percolation threshold. HN_CG works well if the cluster sizes, s, are relatively small ($s \leq 200$) which is the case for $p = 0.1$ or 0.3. On the other hand, HN_FG is more suitable for the large dominating clusters ($s \geq 1000$) found in maps with $p = 0.62$. Either method works well for $200 \leq s \leq 1000$.

Finally, the hybrid program (HN_HY) which utilizes both the coarse-grained and fine-grained methods, can dynamically switch between the two methods so that the best method is matched with the appropriate cluster sizes. As a result, good speed improvements have been obtained for all p-values considered. The graphs in Figure 3.1 illustrate the speed improvements[3] for HN_CG, HN_FG, and HN_HY when processing maps with $p = 0.3$ and 0.62. In Figure 3.1(a), HN_CG and HN_HY show practically the same speed improvement since they are both resolving cluster geometries using only the coarse-grained method. HN_FG, on the other hand, virtually shows no speed improvements. Figure 3.1(b) shows contrasting results for HN_FG and HN_CG with HN_HY out performing HN_FG due to HN_HY's capability of resolving the large cluster(s) in a fine-grained manner and the small accompanying clusters in the coarse-grained manner. Among host/node models, we recommend the hybrid model for analyzing maps with any particular p-value.

3.3.2 Hostless Methods

Although the synchronous and asynchronous hostless programs (HL_SYNC and HL_ASYNC) showed significant speed improvements over C_MAP, they are not consistently better across all p-values. Table 3.6 shows the times for HL_SYNC and HL_ASYNC and Figure 3.2 compares their speed improvements to that of the host/node hybrid program (HN_HY) for all p-values of the 512×512 map[4].

For the small p-values (0.1 and 0.3) HL_ASYNC obtains the best performance among all parallel models. Again, all clusters in these maps are small, and as a result, will usually be *local* to the PNs sub-grid. Since most clusters do not cross boundaries, there is very little communication needed between PNs. However, once a large cluster percolates across the whole map, as in the $p = 0.62$ map, all PNs must communicate with each other to resolve the R_s^2. Since PN 0 will be the *owner* of this large cluster,

[2] All speed improvements are in reference to the sequential program, C_MAP, unless otherwise stated.
[3] Results for maps with $p = 0.1$ are similar to those with $p = 0.3$.
[4] A similar trend is true for the other map sizes.

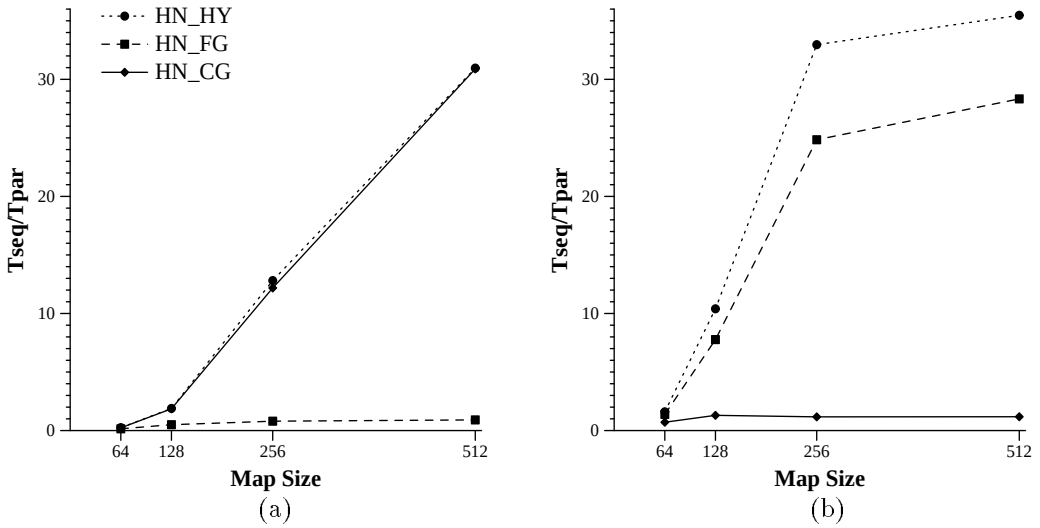


Figure 3.1: Speed improvements of CM-5 implementations over the sequential C version, `C_MAP`, on a Sparc 2 for a 512×512 map with $p = 0.3$ (a) and $p = 0.62$ (b).

Table 3.6: Total wall-clock times (sec.) for parallel hostless models.

	HL_SYNC			HL_ASYNC		
	p-value			p-value		
Map Size	0.1	0.3	0.62	0.1	0.3	0.62
64	0.37	0.72	1.14	0.26	0.29	0.96
128	1.10	4.19	6.67	0.48	2.00	8.25
256	6.79	33.96	118.68	1.00	6.72	88.62
512	118.04	392.11	1787.24	9.93	29.39	1512.42

all other PNs must send PN 0 their coordinates and partial τ values for that cluster. With asynchronous message-passing, PN 0 gets too many messages at one time to process efficiently. As a result, the performance of `HL_ASYNC` for the $p = 0.62$ maps drops dramatically.

The synchronous version (`HL_SYNC`) shows some speed improvements for low p-values, but the performance improves for $p = 0.62$. When the clusters are small, the cost of synchronization and idle-time tend to dominate some of the PNs as they wait for the other processors to finish. For maps with large (e.g., $p = 0.62$) p-values, large clusters tend to dominate the map, and such synchronization becomes an asset by keeping any one node from getting inundated with too many messages at one time. When comparing the host/node hybrid program (`HN_HY`) to the hostless programs (see Figure 3.2) `HL_ASYNC` shows the best results for $p = 0.1$ and 0.3. However, `HN_HY` is more suitable for the denser map, and is generally more consistent across all p-values.

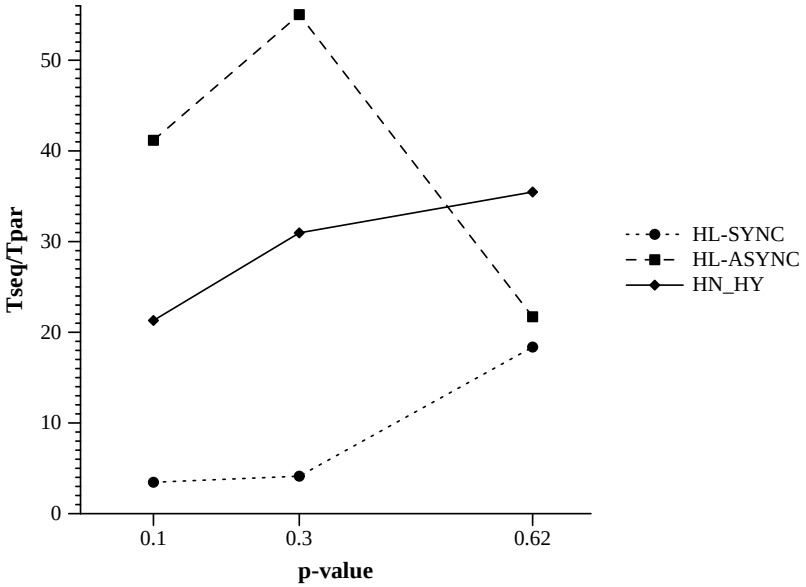


Figure 3.2: Speed improvements of CM-5 parallel implementations for total map analysis on a 512×512 map, all p-values.

3.4 Parallel Models With Vector Units

The host/node hybrid program (`HN_HY`) is perhaps the most flexible of all the parallel models. Since it is very adaptable to different map sizes and map characteristics, we selected this particular program for incorporating the use of vector units (VUs) on the CM-5 nodes.

The VU kernels were first implemented in the coarse-grained method (`HN_CG`), and the effects are illustrated in Table 3.7. Using the vector units, `HN_CG_VU` improves the performance by a factor of 9 for the 512×512 map with $p = 0.62$, but no appreciable gain is obtained for the lower p-value maps. In order for the VUs to be effective, the cluster size must be at least 64 (see Section 2.5). For maps with $p = 0.1$ or 0.3, all cluster sizes are smaller than 64, and as a result, all R_s^2 values are resolved without using the VUs. Although the coarse-grained method with VUs (`HN_CG_VU`) shows good speed improvements for maps with $p = 0.62$, our fine-grained program, `HN_FG`, is still preferable for the dense map. Specifically, `HN_FG` achieves speed improvements on the order of 28 for the 512×512 map with $p = 0.62$ (see Figure 3.3).

Table 3.8 illustrates the effects of using vector units within the fine-grained method (`HN_FG_VU`). As discussed in Section 2.3.2, the fine-grained algorithm is modified to accommodate the VUs whereby each PN is allocated a *block* of pixels instead of one pixel. Consequently, the total cluster size, s, must be at least 40 before the fine-grained method can be implemented. Otherwise, R_s^2 should be resolved in a coarse-

Table 3.7: Total wall-clock times (sec) for coarse-grained method with and without VUs.

	HN_CG			HN_CG_VU		
	p-value			p-value		
Map Size	0.1	0.3	0.62	0.1	0.3	0.62
64	1.47	1.68	6.69	0.57	0.59	1.88
128	2.94	3.10	57.80	1.45	2.08	9.63
256	5.42	8.43	1747.04	4.70	7.60	224.70
512	18.32	52.36	27916.48	19.24	53.26	3713.80

grained fashion. This change in the fine-grained algorithm yields results for maps with $p = 0.1$ and 0.3 that can be misleading. The VUs appear to significantly enhance the performance, however, the cluster sizes are too small ($s < 40$) for implementation of the fine-grained method, and are therefore, resolved by the coarse-grained method which we recall is the preferable method for resolving these smaller clusters. Although the results appear to be good, a comparison with the coarse-grained method without VUs (`HN_CG`) reveals nearly identical timings.

Table 3.8: Total wall-clock times (sec.) for fine-grained method with and without VUs.

	HN_FG			HN_FG_VU		
	p-value			p-value		
Map Size	0.1	0.3	0.62	0.1	0.3	0.62
64	1.33	2.47	3.44	0.41	0.84	0.93
128	3.74	11.57	9.69	1.37	2.07	2.14
256	32.22	127.73	82.00	4.60	7.45	16.82
512	440.61	1784.48	1158.78	19.00	54.42	180.99

It is not until cluster sizes become substantially large that a comparison can be made between the fine-grained methods with and without VUs, `HN_FG_VU` and `HN_FG`, respectively. Figure 3.3 shows speed improvements for all coarse-grained (`CG`) fine-grained (`FG`) and hybrid (`HY`) programs with and without VUs for the 512×512 map with $p = 0.62$. The fine-grained model (`HN_FG_VU`) demonstrates speed improvements of 181 while the coarse-grained program (`HN_CG_VU`) yields a speed improvement of only 9. Obviously, the use of VUs to resolve R_s^2 for the large clusters is where the best performance gains can be made.

Finally, a comparison between implementations of the hybrid program[5] with and without VUs, `HN_HY_VU` and `HN_HY`, respectively, is provided in Table 3.9. As mentioned earlier, the VUs are not effective for maps with $p = 0.1$ and 0.3. However,

[5]Both coarse-grained and fine-grained portions are included in this model.

for maps with $p = 0.62$, the cluster sizes are large enough to exploit the VUs in the fine-grained portion. The performance results in this case are similar to that of the fine-grained model with VUs (`HN_FG_VU`). However, the availability of the coarse-grained method in the hybrid code to resolve small clusters in the dense map allows for slightly better speed improvements ($S_i = 187$) than the fine-grained code ($S_i = 181$) for the 512×512 map with $p = 0.62$.

Table 3.9: Total wall-clock times (sec.) for hybrid method with and without VUs.

	HN_HY			HN_HY_VU		
	p-value			p-value		
Map Size	0.1	0.3	0.62	0.1	0.3	0.62
64	1.45	1.54	2.96	0.40	0.90	0.58
128	2.54	3.04	7.24	1.40	2.85	2.28
256	5.28	8.01	61.78	4.68	10.64	16.16
512	19.19	52.25	925.47	18.76	52.33	176.07
768	72.03	248.01	4796.34	69.61	250.18	884.03
1024	206.97	767.08	14990.30	206.17	853.76	2580.48

3.5 Load Balancing

As described in Section 2.4, three different load balancing algorithms were developed, `UNBAL`, `BAL`, and `BAL32` which attempt to distribute the amount of work evenly across all PNs. We note that all programs in this study using the new fine-grained method with VUs employed `UNBAL`. However, the performance could be enhanced if a better load balancing algorithm was used. Although `BAL` and `BAL32` provide a more equitable distribution of comparisons across the PNs (see Figure 3.4), the resulting performances are, in fact, worse than that of `UNBAL` (see Table B.1).

This phenomenon is best explained by the strip-mining process that must take place when subsets of pixels assigned to each PN are not multiples of 32 in size. The direct relationship between the time spent strip-mining and the Mflops[6]/sec rate for each node for a vector of length 8192 is illustrated in Figures 3.5 and 3.6 for `UNBAL` and `BAL`. Figures B.1 and B.2 illustrate the same relationship for `BAL32`. For each algorithm considered, whenever strip-mining time increases, the subsequent Mflops/sec rate decreases and vice versa. In order to obtain a more optimal load balancing algorithm, the strip-mining effect would have to be diminished.

[6]Millions of floating point operations.

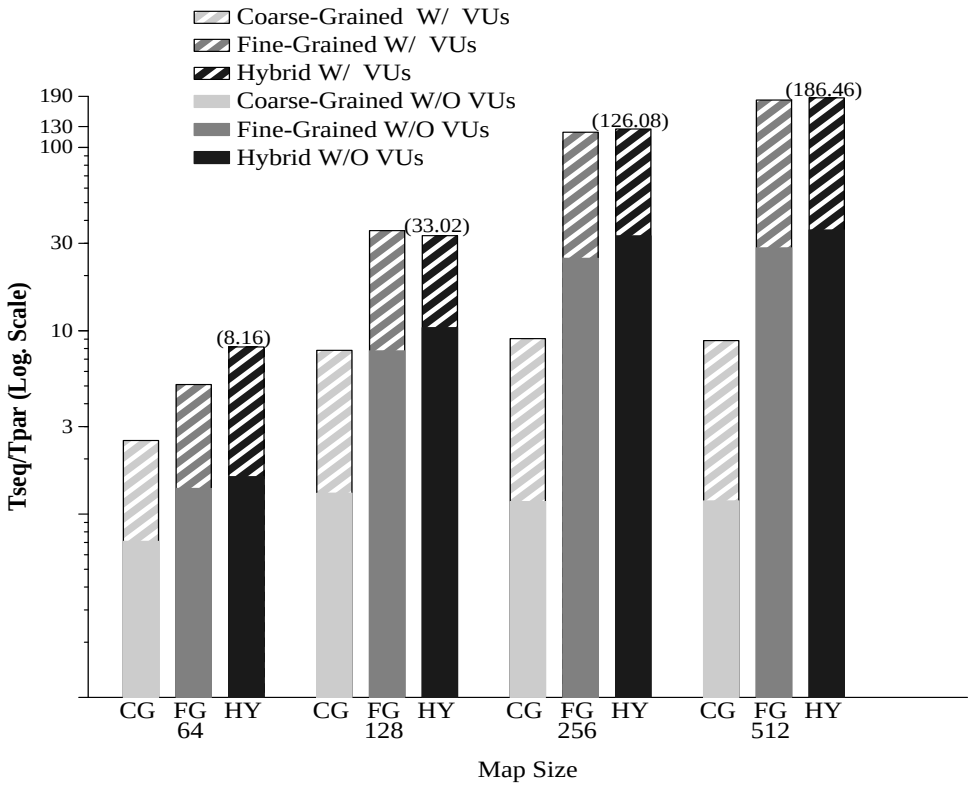


Figure 3.3: Speed improvements for CM-5 parallel implementations of map analysis over the sequential C version on a Sun Sparc 2 for maps with $p = 0.62$.

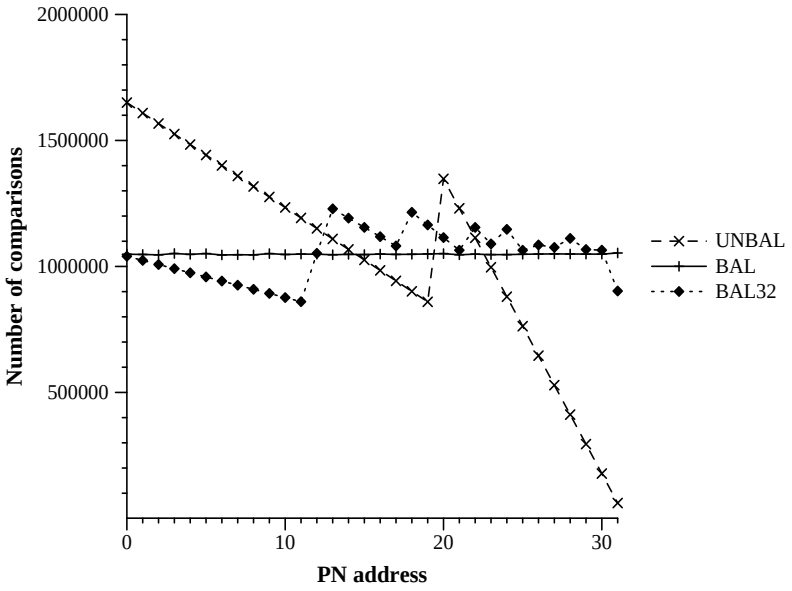


Figure 3.4: Number of comparisons per PN for vector length 8192.

3.6 Comparisons with RISC-based workstations

With the development and availability of faster RISC-based workstations, one may question the need for parallelization. Specifically, how much better are the parallel results compared against that of the fast RISC-base workstations? Is the time and effort put into parallelization really necessary? To address such questions, the sequential map analysis program, `C_MAP`, was also executed on an IBM RS/6000-350 and an Hewlett Packard 9000-750. Table 3.10 lists the speed improvements of the host/node hybrid code with and without VUs (`HN_HY_VU` and `HN_HY`) over `C_MAP` on the three RISC-based workstations. Speed improvements range from 3.71 to 35.47 without VUs, and from 19.50 to 186.46 with VUs. Hence, the parallel models (`HN_HY` and `HN_HY_VU`) used on the CM-5 significantly out-perform the serial implementation on RISC-base workstations and thereby suggest that parallelization is certainly worthwhile.

3.7 Real Maps

The use of random maps was helpful for designing flexible parallel models that could effectively process maps of varying size and densities. The hybrid program with VUs (`HN_HY_VU`) is one such flexible program that is capable of efficiently resolving a variety of maps. To test this claim, we then analyzed a *real* map produced through the process of remote imagery and used in landscape ecology studies. The map shown

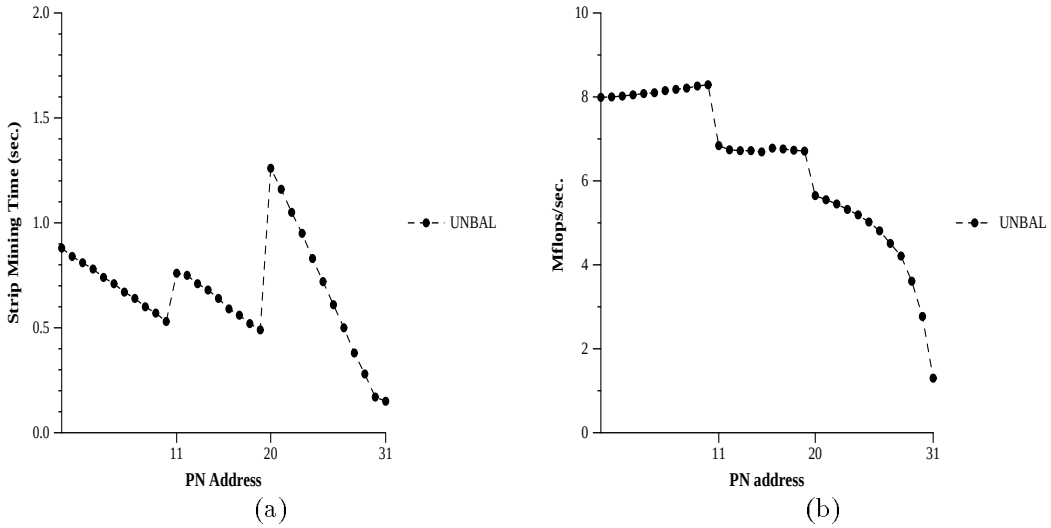


Figure 3.5: Comparison of strip-mining time (a) and Mflops/sec (b) for each PN employing `UNBAL` for vector length 8192.

Table 3.10: Speed improvements of `HN_HY_VU` over `C_MAP` on RISC-based workstations for a 512×512 map, $p = 0.62$.

Machine	without VUs	with VUs
Sun/Sparc2	35.47	186.46
HP 9000-750	18.14	95.36
IBM RS/6000-350	3.71	19.50

in Figure 3.7 reflects a portion of Yellowstone National Park used within a fire model, *EMBYR*, developed at Oak Ridge National Laboratory [HGTR+92]. This particular map is 454×454 and contains 10 different map classes or habitat types, which are resolved individually. As illustrated in Table 3.11 a cumulative speed improvement on the order of 119 is obtained over the sequential program, `C_MAP`. Again, the flexibility and the efficiency of the parallel program is well-demonstrated as the map classes are made up of different numbers and sizes of clusters.

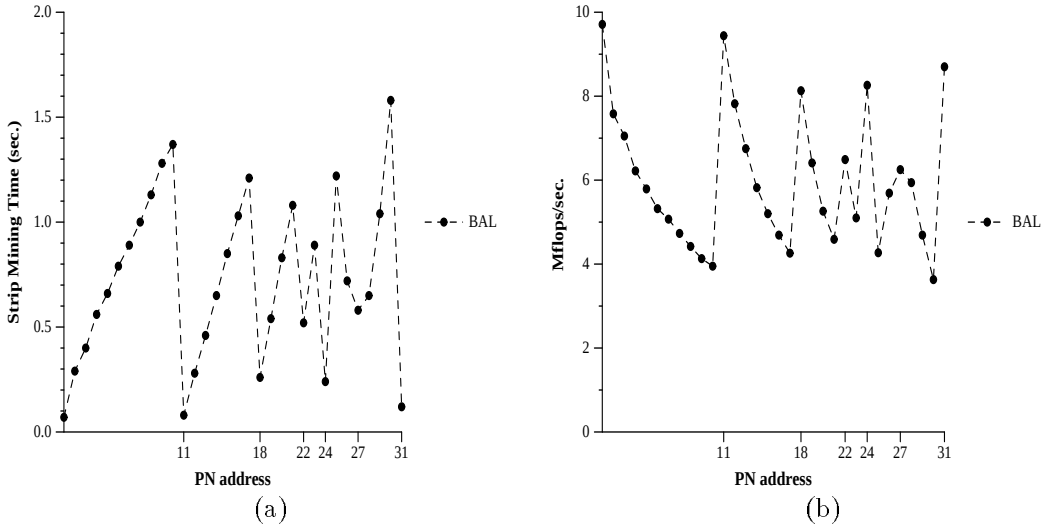


Figure 3.6: Comparison of strip-mining time (a) and Mflops/sec (b) for each PN employing `BAL` for vector length 8192.

Table 3.11: Total times (sec.) and speed improvements over the sequential C version, `C_MAP`, on a Sun Sparc 2 for parallel implementations for the CM-5 when resolving map classes of the fire map in Figure 3.7.

Mapclass #	No. of Clusters	Size of Largest Cluster	Execution Time (sec.) `C_MAP` Sequential	`HN_HY` (w/o VUs)	`HN_HY_VU` (w/ VUs)
1	29	10525	230.94	16.16	5.81
2	55	15403	484.51	22.78	7.04
3	66	10329	312.46	25.08	8.19
4	107	67920	8641.73	239.20	45.54
5	73	1443	14.57	5.62	4.58
6	19	5395	128.19	15.97	4.76
7	19	9	0.93	1.47	1.65
8	1	1	0.50	1.38	1.48
9	0	0	0.50	1.38	1.60
10	5	173	0.63	1.49	1.77
		Total Time	9815.38	331.77	82.22
	Speed	Improvement		39.55	119.38

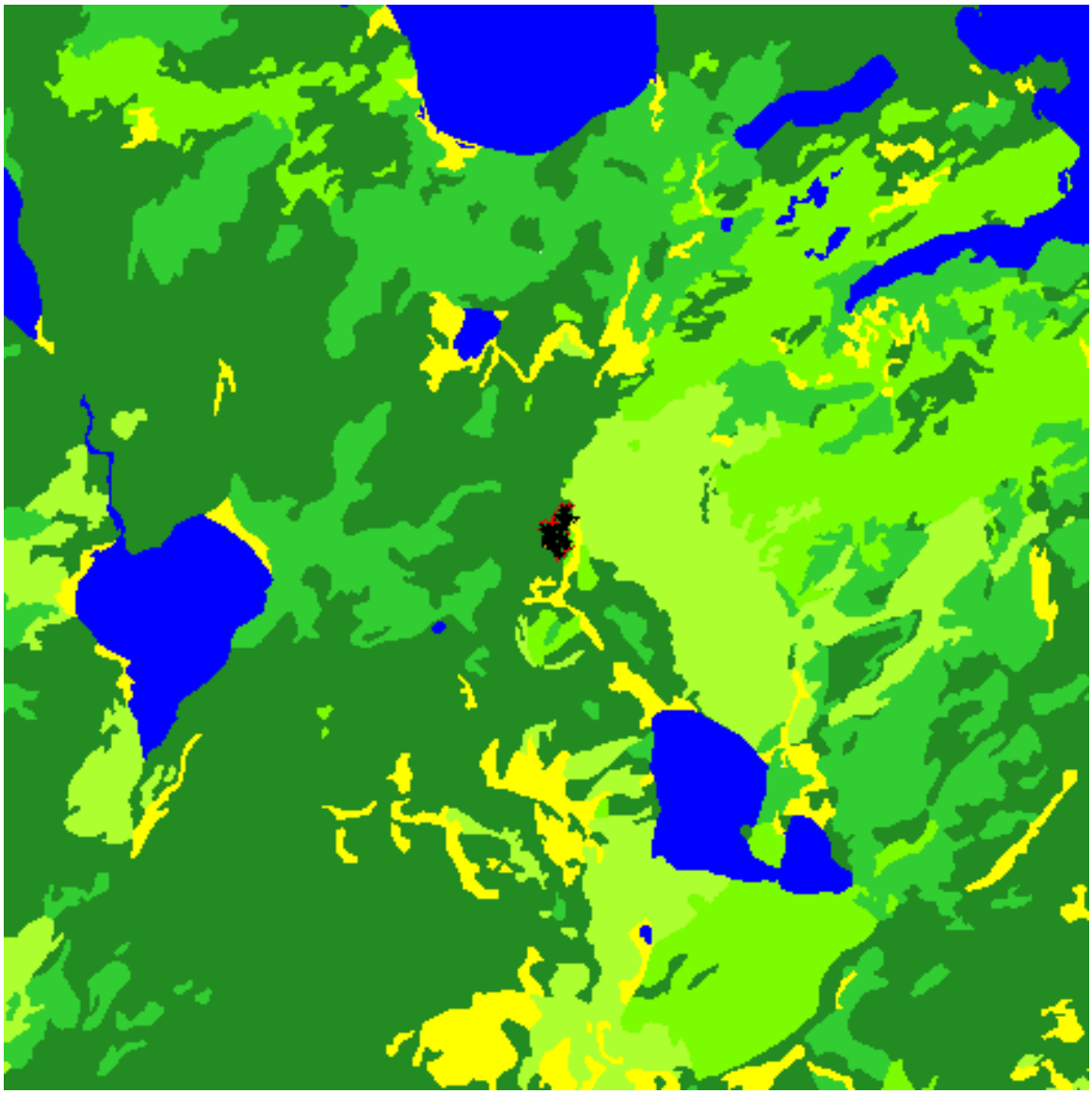


Figure 3.7: A 454×454 map with 10 map classes representing a portion of Yellowstone National Park.

Chapter 4

Conclusions

The goal of this thesis was to provide a scalable map analysis tool for landscape ecology models that could efficiently perform cluster identification and cluster geometry on large maps. To aid in software development, random maps were generated to represent different map sizes and characteristics that are found in real maps. After revising the sequential cluster identification algorithm, cluster geometry became the major focus of parallelization on the CM-5, as it consumes approximately 98% of the sequential map analysis time. Several different parallel methods were implemented successfully, including host/node and hostless models. The host/node models were later modified to exploit the vector units through the use of `DPEAC` kernels.

The hostless program with limited asynchronous message passing (`HL_ASYNC`) showed the best performance for maps with low p-values: speed improvements of 41 and 55 for a 512×512 map with $p = 0.1$ and 0.3, respectively. The denser maps ($p = 0.62$) with large dominating cluster(s), were best resolved by the host/node hybrid programs (`HN_HY` and `HN_HY_VU`). Without utilization of the VUs, the `HN_HY` program obtained a speed improvement of 35 for the 512×512 map with $p = 0.62$. In terms of elapsed wall-clock time, the sequential program, `C_MAP`, required over 9 hours to resolve this same map, and the `HN_HY` program required only 15 minutes. With the addition of the VUs, the `HN_HY_VU` code could complete the analysis for this map in 175 seconds, achieving a speed improvement of 187 over `C_MAP`. When compared to other RISC-based workstations for this same map, the `HN_HY_VU` program could be as much as 95 and 20 times faster than the Hewlett Packard 9000-750 and the IBM RS/6000-350, respectively.

The efficiency and flexiblity of the host/node hybrid codes were further tested with a real map of varying cluster numbers and characteristics used in landscape ecology studies shown in Figure 3.7. Where the sequential map analysis program (`C_MAP`) required 2.75 hours to complete, the `HN_HY_VU` program completed the analysis in 82 seconds and achieved a speed improvement of 119. Also, these particular hybrid programs could be easily ported to other MIMD machines such as the Intel Hypercube.

Future research in map analysis software for landscape ecology involves (*i.*) the inclusion of other neighbor rules for cluster definition, (*ii.*) the development of more effective VUs modules for low density maps, (*iii.*) better load-balancing algorithms to effectively deal with strip-mining, (*iv.*) addressing the effects of strip-mining within the `DPEAC` kernels, and (*v.*) the incorporation of better memory management to enable effective map analysis on larger maps that pose serious memory constraints.

It is our conclusion that our map analysis implementations are quite successful in meeting the goal of this research. The map analysis software is efficient, flexible, scalable, and portable. With the future advancements listed above, these programs should become more scalable and robust, further enhancing the process of analyzing large maps used in landscape models, and more generally, large maps produced through the use of remote imagery.

Bibliography

Bibliography

[BeCM93a] M. Berry, J. Comisky, 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. Comisky, and K. Minser. Parallel map analysis on the MasPar MP-2. *Computer Science Department Technical Report CS-93-190*, University of Tennessee, March 1993.

[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.

[GaON91] R. H. Gardner and R. V. O'Neill. Pattern, process and predictability: The use of neutral models for landscape analysis. In: *Quantitative Methods in Landscape Ecology. The analysis and interpretation of landscape heterogeneity.* M. G. Turner and R. H. Gardner, eds. Ecological Studies Series, Springer-Verlag, NY, 289–307, 1991.

[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.

[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.

[Hwan93] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, New York, 1993.

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

[TMC92a] *CMMD Reference Manual*, Version 2.0. Thinking Machines Corporation, Cambridge, 1992.

[TMC92b] *CMMD User's Guide*, Version 1.1. Thinking Machines Corporation, Cambridge, 1992.

[TMC92c] *CMMD User's Guide*, Version 2.0. Thinking Machines Corporation, Cambridge, 1992.

[TMC92d] *DPEAC Reference Manual*, Version 7.1. Thinking Machines Corporation, Cambridge, 1992.

Appendices

Appendix A

Supplementary Map and Performance Data

	p = 0.1		p = 0.3		p = 0.62	
m	TTL	LC	TTL	LC	TTL	LC
64	325	8	534	25	110	1981
128	1305	6	2157	29	382	6609
256	5227	7	8484	33	1400	34363
512	20917	12	33891	42	5503	141190
768	47281	8	75941	44	11989	323676
1024	84140	15	135122	55	21141	577501

Table A.1: Total number of clusters, TTL, and largest cluster, LC, for $m \times m$ maps, all p-values.

m	C_MAP	HN_CG	HN_FG	HN_HY	HL_SYNC	HL_ASYNC
64	0.12	1.47	1.33	1.45	0.37	0.26
128	1.48	2.94	3.74	2.54	1.10	0.48
256	25.65	5.42	32.22	5.28	6.79	1.00
512	408.81	18.32	440.61	19.19	118.04	9.93
768	—	—	—	72.03	—	46.06
1024	—	—	—	206.97	—	125.22

m	HN_CG_VU*	HN_FG_VU**	HN_HY_VU**
64	0.57	0.41	0.40
128	1.45	1.37	1.40
256	4.70	4.60	4.68
512	19.24	19.00	18.76
768	—	—	69.61
1024	—	—	206.17

*Clusters are too small to use VUs.

**Clusters are too small to use fine-grained method or VUs.

Table A.2: Total times (sec.) for maps with p = 0.1.

m	C_MAP	HN_CG	HN_FG	HN_HY	HL_SYNC	HL_ASYNC
64	0.38	1.68	2.47	1.54	0.72	0.29
128	5.75	3.10	11.57	3.04	4.19	2.00
256	102.64	8.43	127.73	8.01	33.96	6.72
512	1617.47	52.36	1784.48	52.25	392.11	29.39
768	—	—	—	248.01	—	114.76
1024	—	—	—	767.08	—	308.49

m	HN_CG_VU*	HN_FG_VU**	HN_HY_VU**
64	0.59	0.84	0.90
128	2.08	2.07	2.85
256	7.60	7.45	10.64
512	53.26	54.42	52.33
768	—	—	250.18
1024	—	—	853.76

*Clusters are too small to use VUs.

**Clusters are too small to use fine-grained method or VUs.

Table A.3: Total times (sec.) for maps with p = 0.3.

m	C_MAP	HN_CG	HN_FG	HN_HY	HL_SYNC	HL_ASYNC
64	4.73	6.69	3.44	2.96	1.14	0.96
128	75.29	57.80	9.69	7.24	6.67	8.25
256	2036.45	1747.04	82.00	61.78	118.68	88.62
512	32830.08	27916.48	1158.78	925.47	1787.24	1512.42
768	—	—	—	4796.34	—	—
1024	—	—	—	14990.30	—	—

m	HN_CG_VU	HN_FG_VU	HN_HY_VU
64	1.88	0.93	0.58
128	9.63	2.14	2.28
256	224.70	16.82	16.16
512	3713.80	180.99	176.07
768	—	—	884.03
1024	—	—	2580.48

Table A.4: Total times (sec.) for maps with p = 0.62.

Appendix B

Load Balancing

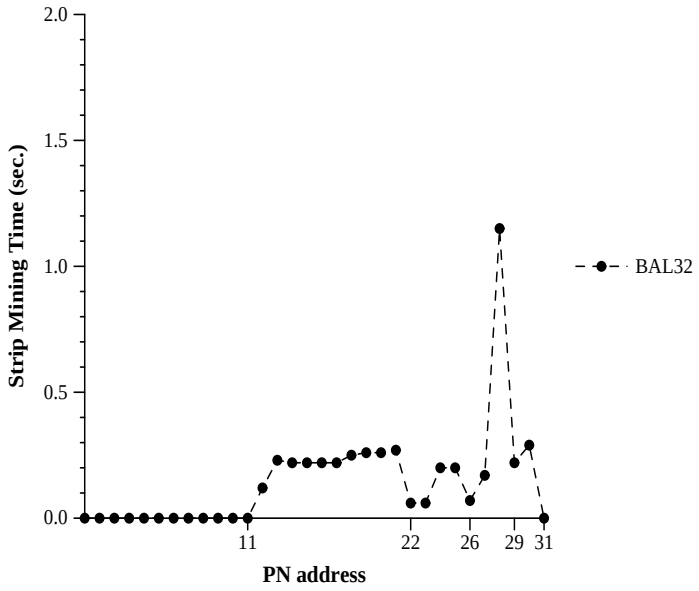


Figure B.1: Strip-mining time (sec.) per PN for `BAL32`.

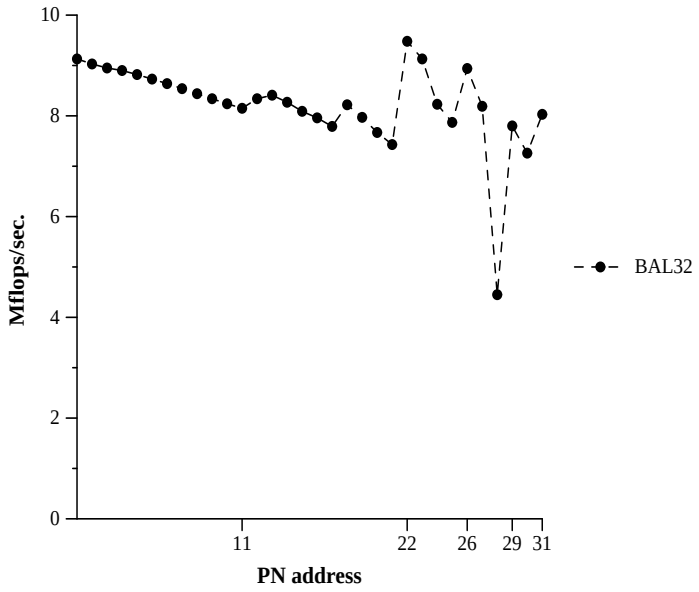


Figure B.2: Mflops/sec. per PN for `BAL32`.

Algorithm	Vector Length					
	8192	14039	16384	28078	32768	68000
`UNBAL`	1.80	3.98	5.77	10.84	16.72	36.66
`BAL`	2.41	3.82	7.37	12.68	26.89	67.36
`BAL32`	1.92	2.89	5.76	13.31	26.79	65.77

Table B.1: Time (sec.) to resolve R_s^2 using different load balancing algorithms.

Appendix C

Flowchart for Host/Node Model with VUs

Figure C.1: Flowchart for the Host/Node Hybrid Model with VUs (`HN_HY_VU`).

* All maps analyzed in this study contained either all small clusters or a mixture of small and very large clusters. Therefore, the maps were resolved with only the coarse-grained method or a combination of the coarse-grained and fine-grained methods. When using the fine-grained method to analyze maps with only large clusters, an option could be added to allow set_up() to call getstats_fine directly.

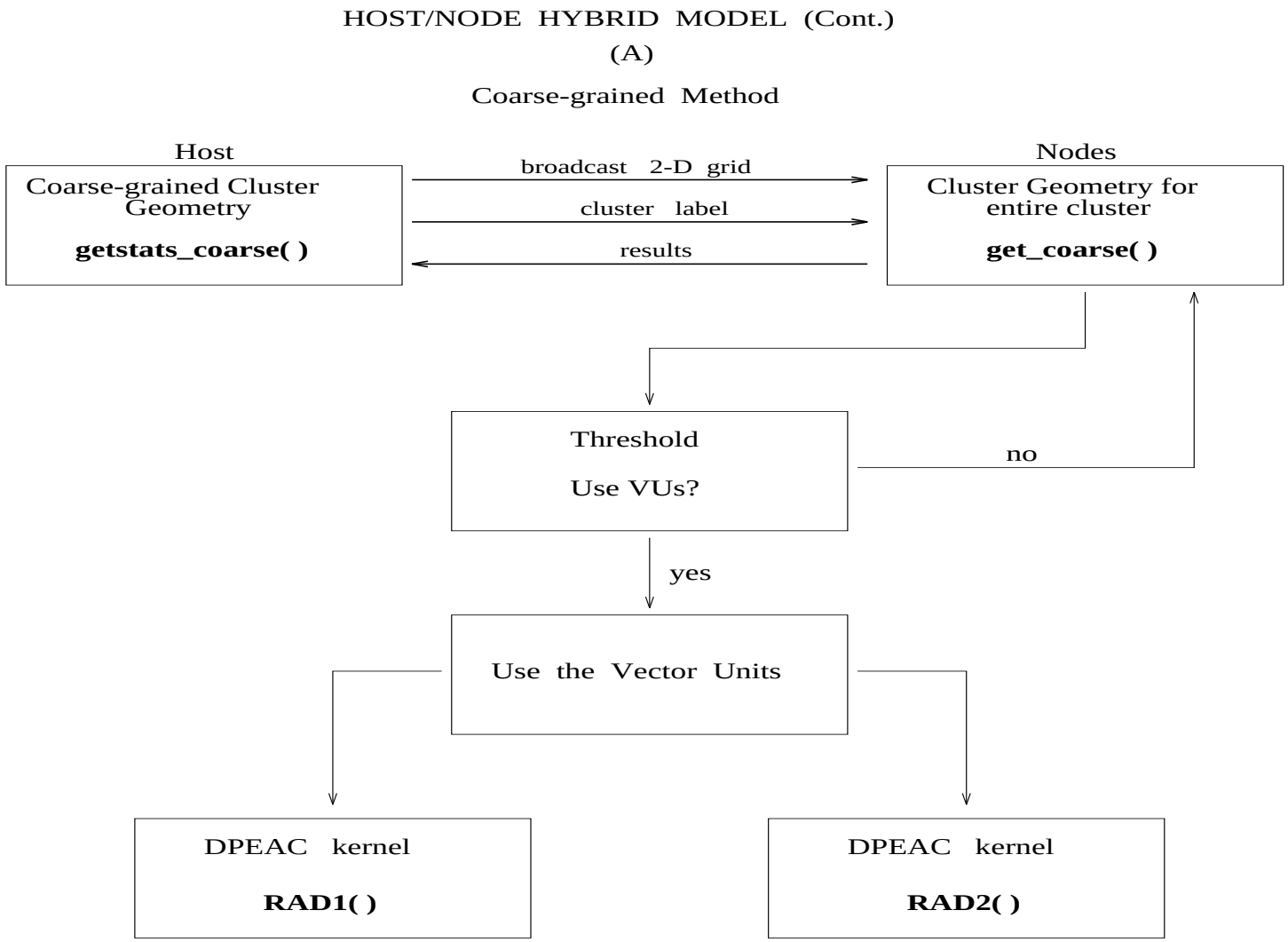


Figure C.2: Flowchart for the coarse-grained portion of `HM_HY_VU`.

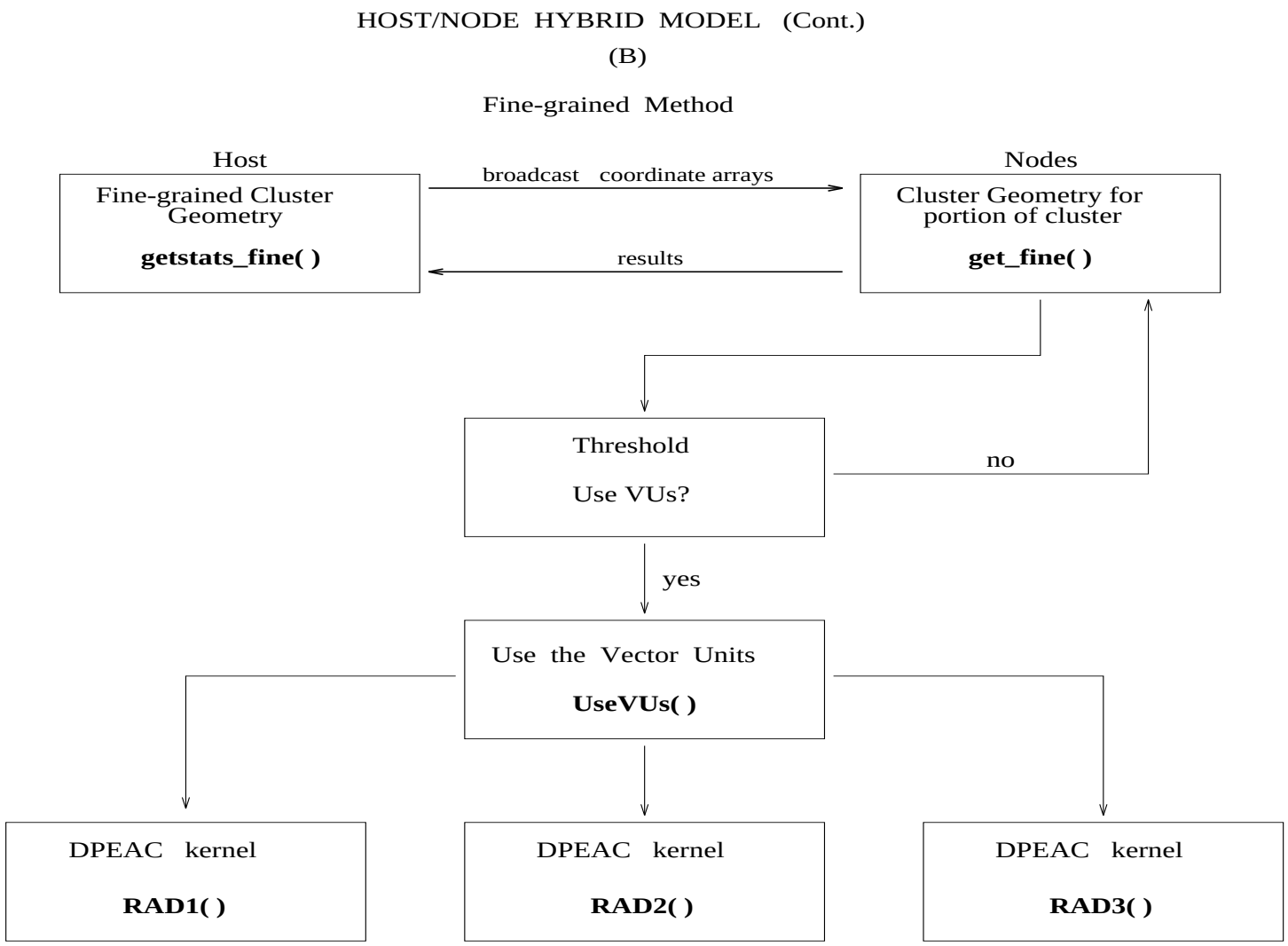


Figure C.3: Flowchart for the fine-grained portion of `HM_HY_VU`.

Appendix D

Prologues of Functions

```
/*************************************************************/
/*                                                           */
/*  Name:  clusterID                                         */
/*                                                           */
/*  Function:                                                */
/*         Traverses 2-D grid in row-major order looking     */
/*         for a value of -1 which designates the            */
/*         beginning of a cluster.  The function             */
/*         Site_Label is then called to locate, label,       */
/*         and count all grid elements or pixels that        */
/*         belong to the current cluster.  The size of       */
/*         the cluster is recorded in the array (csize[]),   */
/*         the cluster label (lbl) is incremented by 1,      */
/*         and traversal of the grid  continues to locate    */
/*         the next cluster.                                 */
/*                                                           */
/*************************************************************/
```

```
/*************************************************************/
/*                                                           */
/*  Name:  Site_Label                                        */
/*                                                           */
/*  Function:                                                */
/*         A recursive function that locates, labels, and    */
/*         counts all elements or pixels of a cluster.       */
/*                                                           */
/*  Parameters:                                              */
/*         tmp1 - current pixel address                      */
/*         coordinates - array containing address offsets    */
/*                       to the NEWS neighbors of a pixel    */
/*         size - address of current cluster size            */
/*         lbl - current cluster label                       */
/*                                                           */
/*************************************************************/
```

```
/*************************************************************/
/*                                                           */
/*  Name:  Radius                                            */
/*                                                           */
/*  Function:                                                */
/*         Locates all labeled pixels of a cluster, loads    */
/*         the x- and y-coordinates into arrays, istack      */
/*         and jstack, respectively, and determines the      */
/*         mean squared radius for that cluster.             */
/*                                                           */
/*  Parameters:                                              */
/*         lbl - current cluster label                       */
/*         size - current cluster size                       */
/*         rms - address of current cluster's mean           */
/*               squared radius.                             */
/*                                                           */
/*************************************************************/


/*************************************************************/
/*                                                           */
/*  Name: RAD1                                               */
/*                                                           */
/*  Function:                                                */
/*         DPEAC kernel that compares elements of two        */
/*         similar vectors.  All elements must be            */
/*         compared to each other requiring rotations        */
/*         of elements in vector registers.  Vector          */
/*         registers are set to hold 8 double-precision      */
/*         elements.                                         */
/*                                                           */
/*  Parameters:                                              */
/*         AIN - address of vector A used for comparisons    */
/*         BIN - address of vector B used for comparisons    */
/*         RES - address of results                          */
/*         DPSP - stack pointer                              */
/*         SIZE - size of vectors A and B                    */
/*         SAVE - number of elements to be processed         */
/*                by each VU                                 */
/*                                                           */
/*************************************************************/
```

```
/*************************************************************/
/*                                                           */
/*  Name: RAD2                                               */
/*                                                           */
/*  Function:                                                */
/*        DPEAC kernel that compares two vectors of the      */
/*        same length that are different subsets of the      */
/*        same vector.  All elements must be compared to     */
/*        each other requiring rotations of elements in      */
/*        vector registers which hold 8 double-precision     */
/*        elements.                                          */
/*                                                           */
/*  Parameters:                                              */
/*        AIN - address of vector A used for comparisons     */
/*        BIN - address of vector B used for comparisons     */
/*        RES - address of results                           */
/*        DPSP - stack pointer                               */
/*        SIZE - size of vectors A and B                     */
/*                                                           */
/*************************************************************/

/*************************************************************/
/*                                                           */
/*  Name: RAD3                                               */
/*                                                           */
/*  Function:                                                */
/*        DPEAC kernel that compares two vectors of          */
/*        dissimilar length and are different subsets        */
/*        of the same vector.  All elements must be          */
/*        compared to each other requiring rotations of      */
/*        elements in vector registers which hold 8          */
/*        double-precision elements.                         */
/*                                                           */
/*  Parameters:                                              */
/*        AIN - address of vector A used for comparisons     */
/*        BIN - address of vector B used for comparisons     */
/*        RES - address of results                           */
/*        DPSP - stack pointer                               */
/*        SIZEA - size of vector A                           */
/*        SIZEB - size of vector B                           */
/*                                                           */
/*************************************************************/
```

Appendix E

Example of DPEAC kernel

```
! DPEAC kernel: RAD1
!
! Function: Compares elements of two similar vectors.  All elements
!    must be compared to each other requiring rotations of elements
!    in vector registers.  Vector registers are set to hold 8
!    double-precision elements.
!
! Parameters:  AIN - address of vector A
!              BIN - address of vector B
!              RES - address of results
!              DPSP - stack pointer
!              SIZE - size of vectors A and B
!              SAVE - number of elements  processed per VU
!

#include <cmsys/dpeac.h>

#define AIN     i0       !  Address of vector A
#define BIN     i1       !  Address of vector B
#define RES     i2       !  Address of results
#define DPSP    i3       !  Stack pointer
#define SIZE    i4       !  Size of vectors A and B
#define SAVE    i5       !  Number of elements processed per VU
#define A       o0       !  Pointer to A in sparc space
#define B       o1       !  Pointer to B in sparc space
#define AP      l0       !  Pointer to A in parallel space
#define BP      l1       !  Pointer to B in parallel space
#define COUNT   l2       !  Counter
#define TMP     o2       !  Temporary storage
#define POINT   l4       !  Pointer
#define COUNTSA l5       !  Saves Count
#define COUNTA  g1       !  Counter for vector A
#define COUNTB  g6       !  Counter for vector B
#define POINTA  o3       !  Pointer for vector A
#define POINTB  o4       !  Pointer for vector B
#define SEGS    o5       !  Number of segments of 8

#define ZERO    0f0.0
#define ONE     0d1.0
#define TWO     0d2.0
#define VLEN    8        !  Vector length
```

```
#define SOD     8        !  Size of data (8 for double)

        dpentry _RAD1, 0, 0

!   Save the count of elements of an array per vu
        srl  %SIZE, 2, %SIZE
        move %SIZE, %COUNTSA

!   Determine number of segments of 8 per vu
        srl %COUNTSA, 3, %SEGS

!   Multiply by 8 ( the size of a double in bytes)
        sll %SIZE, 3, %SIZE

!   Decrement stackpointer to find start address
        sub %DPSP, %SIZE, %DPSP
        sub %DPSP, %SIZE, %DPSP

!   Change from instruction space to data space
        dpchgsp %DPSP, %DPSP

!   Load pointer for A and B in vu space

        add %DPSP, 0, %AP
        add %DPSP, %SIZE, %BP

!   Set counter and offset pointer
        move %COUNTSA, %COUNT
        move 0, %POINT
!   Select dp 0
        dpchgbk %AP, DP_0, %AP
        dpchgbk %BP, DP_0, %BP

        move %AIN, %A
        move %BIN, %B

!   Move array A and B into memory bank of dp 0
loop1:
        ldd [%A + %POINT], %TMP
        dpstd %TMP, [%AP + %POINT]
```

```
        ldd [%B + %POINT], %TMP
        dpstd %TMP, [%BP + %POINT]

        subcc %COUNT, 1, %COUNT
        bnz loop1
        add %POINT, SOD, %POINT

!   Increase pointer to A and B in sparc space

        add %A, %POINT, %A
        add %B, %POINT, %B

        move %COUNTSA, %COUNT
        move 0, %POINT
!   Select dp 1
        dpchgbk %AP, DP_1, %AP
        dpchgbk %BP, DP_1, %BP

!   Move array A and B into memory bank of dp 1
loop2:
        ldd [%A + %POINT], %TMP
        dpstd %TMP, [%AP + %POINT]

        ldd [%B + %POINT], %TMP
        dpstd %TMP, [%BP + %POINT]

        subcc %COUNT, 1, %COUNT
        bnz loop2
        add %POINT, SOD, %POINT

!   Increase pointer to A and B in sparc space

        add %A, %POINT, %A
        add %B, %POINT, %B

        move %COUNTSA, %COUNT
        move 0, %POINT
!   Select dp 2
        dpchgbk %AP, DP_2, %AP
        dpchgbk %BP, DP_2, %BP
```

```
!   Move array A and B into memory bank of dp 2
loop3:
        ldd [%A + %POINT], %TMP
        dpstd %TMP, [%AP + %POINT]

        ldd [%B + %POINT], %TMP
        dpstd %TMP, [%BP + %POINT]

        subcc %COUNT, 1, %COUNT
        bnz loop3
        add %POINT, SOD, %POINT

!   Increase pointer to A and B in sparc space

        add %A, %POINT, %A
        add %B, %POINT, %B

        move %COUNTSA, %COUNT
        move 0, %POINT
!   Select dp 3
        dpchgbk %AP, DP_3, %AP
        dpchgbk %BP, DP_3, %BP

!   Move array A and B into memory bank of dp 3
loop4:
        ldd [%A + %POINT], %TMP
        dpstd %TMP, [%AP + %POINT]

        ldd [%B + %POINT], %TMP
        dpstd %TMP, [%BP + %POINT]

        subcc %COUNT, 1, %COUNT
        bnz loop4
        add %POINT, SOD, %POINT

        move 0, %POINT
        move %COUNTSA, %COUNT

!   Change to all dp's
        dpchgbk %AP, ALL_DPS, %AP
        dpchgbk %BP, ALL_DPS, %BP
```

```
        dpchgsp %AP, %AP
        dpchgsp %BP, %BP

        set_vector_length_and_vmmode VLEN, always

        dfmovev ZERO, v6
        fnopv

!   Compare similar vectors

    L6: move 7, %COUNTSA
        fnopv; dfloadv  [%AP + %POINT]:SOD,v2
        dfsubv v2, v4, v10; dfloadv [%BP + %POINT]:SOD,v4
        dfabsv v10, v10
        dfaddv v10, ONE,v8
        dfmulv v8, v8, v8

!   Sum up values of v8 in v6
        dfaddv v6, v8, v6 ; memnop

!   Rotate vector B

    L7: dfmoves  v4[0],r112:16
        dfmoves  v4[2],v4[0]
        dfmoves  v4[4],v4[2]
        dfmoves  v4[6],v4[4]
        dfmoves  v4[8],v4[6]
        dfmoves  v4[10],v4[8]
        dfmoves  v4[12],v4[10]
        dfmoves  v4[14],v4[12]
        dfmoves  r112:16,v4[14]

!   Compare vectors with B rotated

        dfsubv V2,V4,V10;memnop
        dfabsv v10,v10
        dfaddv v10,ONE,v8
        dfmulv v8,v8,v8
        dfaddv v6,v8,v6;memnop
```

```
        subcc %COUNTSA, 1 ,%COUNTSA
        bnz L7
        nop

       subcc %COUNT, VLEN, %COUNT
       bgt L6
       add %POINT, SOD*VLEN, %POINT

       dpwrt *, 0, r8
       dpwrt *, 0, r9
       dpwrt *, 0, r72
       dpwrt *, 0, r73

!   Sum up v6 on each vu
       dfaddv v6, r8:64, r8:64
       fnopv
       dfadds r8, r72, r8

!   Eliminate values from extra comparisons
       ldd [%SAVE], %f4
       dpwrtd *,%f4, r72
       dfsubs  r8, r72, r8
       dfdivs r8, TWO, r8
       fnopv
       dfmovev ZERO, v6

!  Compare dissimilar vectors

       subcc %SEGS, 1, %SEGS
       cmp %SEGS,0
       beq L10
       move %SEGS, %COUNTA
       move 0, %POINTA
  L9B: move %POINTA, %POINTB
       add  %POINTB, SOD*VLEN, %POINTB
       move %SEGS, %COUNTB
       subcc %SEGS, 1, %SEGS

       fnopv; dfloadv [%AP + %POINTA]:SOD,v2
  L8B: dfsubv v2, v4, v10; dfloadv [%BP + %POINTB]:SOD,v4
       move 7, %COUNTSA
```

```
        fnopv; dfloadv [%AP + %POINTA]:SOD,v2
        dfabsv v10, v10
        dfaddv v10, ONE,v8
        dfmulv v8, v8, v8

!   Sum up values of v8 in v6
        dfaddv v6, v8, v6 ; memnop

!   Rotate vector B

  L7B: dfmoves  v4[0],r112:16
        dfmoves  v4[2],v4[0]
        dfmoves  v4[4],v4[2]
        dfmoves  v4[6],v4[4]
        dfmoves  v4[8],v4[6]
        dfmoves  v4[10],v4[8]
        dfmoves  v4[12],v4[10]
        dfmoves  v4[14],v4[12]

        dfmoves  r112:16,v4[14]
        dfsubv V2,V4,V10;memnop
        dfabsv v10,v10
        dfaddv v10,ONE,v8
        dfmulv v8,v8,v8
        dfaddv v6,v8,v6;memnop

        subcc %COUNTSA, 1 ,%COUNTSA
        bnz L7B
        nop

        add %POINTB, SOD*VLEN, %POINTB
        subcc %COUNTB, 1, %COUNTB
        bnz L8B
        nop

        add %POINTA, SOD*VLEN, %POINTA
        subcc %COUNTA, 1, %COUNTA
        bnz L9B
        nop

        dpwrt *, 0, r72
```

```
      dpwrt *, 0, r73

!   Sum up v6 on each vu
      dfaddv v6, r8:64, r8:64
      fnopv
      dfadds r8, r72, r8

!   Read the result from each vu and sum it up
      dprdd DP_0, r8, %f4
      dprdd DP_1, r8, %f2
      faddd %f4, %f2, %f4
      dprdd DP_2, r8, %f2
      faddd %f4, %f2, %f4
      dprdd DP_3, r8, %f2
      faddd %f4, %f2, %f4

!   Save the result
      std %f4, [%RES]

      dpretn
```

VITA

Karen Stoner Minser was born in Oak Ridge, Tennessee on September 2, 1954, attended the Oak Ridge school system, and graduated from Oak Ridge High School in 1972. She entered college at the University of Tennessee in 1972 and received a bachelors degree in Microbiology in 1976, graduating with High Honors and membership in the Phi Beta Kappa and Phi Kappa Phi honorary societies. In 1979, she received the Tennessee Teaching Certification from the University of Tennessee. She returned to school in 1990, entering the graduate program in computer science at the University of Tennessee where she received her Master of Science degree in computer science in August, 1993. She is a member of the computer science honorary society, Upsilon Pi Epsilon and received the Chancellor's Citation For Extraordinary Professional Promise in April, 1993.