SCCS-310
Computers & Geosciences
Vol. 19, No. 7, pp. 1051-1063.

GORDIUS: A DATA PARALLEL ALGORITHM FOR SPATIAL DATA CONVERSION

Kim Mills[1], Ferenc Csillag[2], Maher Kaddoura[3]

[1]Northeast Parallel Architectures Center, [2]Department of Geography, [3]School of Computer and Information Science, Syracuse University, Syracuse, New York 13244-4100, U.S.A.,
kim@nova.npac.syr.edu

Abstract

A data parallel algorithm is presented for spatial data (raster-to-vector) conversion. It operates on pure two-dimensional raster images and derives a fully topological vector data set, where polygons are defined by homogeneity criteria. The algorithm is implemented in data parallel C, and makes extensive use of scan functions that combine communication and computation along grid axes. Code segments listing each step in building a hierarchy of points, lines, and polygons are listed. Performance evaluations on a Connection Machine-2 reveal that run-time is not sensitive to the number of polygons in the image, and increases sublinearly with grid size.

Keywords: Raster-to-vector conversion, data parallel algorithm, GIS.

INTRODUCTION

Spatial data in digital form, from diverse sources, must routinely be combined for geographic information analysis. For example, census files (Marx, 1990), standardized topographic data sets (USGS, 1983), and satellite imagery (EOSAT, 1991) often must be combined for both local and regional, and especially global scale analysis (Mounsey and Tomlinson, 1989).

Spatial data can be represented in many ways, for example in raster or vector format. Raster data are based on a grid format of uniform size, with a logical record consisting of a grid cell or pixel. Vector data are based on an object in space and defined by a set of points and lines, with a logical record consisting of a polygon. Raster data are convenient for neighborhood-dependent operations such as filtering, because neighborhood information is implicitly encoded in the data structure. Covering large areas at high resolution however, requires very large data sets. Data encoded in vector format are typically more compact because one can store areal information in polygon format with points along a perimeter. Processing neighborhood information, or multiple sets of vector data may be cumbersome because one must first analyze the data to find all neighboring and intersecting polygons and define the set of points belonging to each polygon.

Different applications tend to require conceptually different data representations (Mark and Csillag, 1989), and while both raster and vector data formats have their own strengths and weaknesses, both forms of data are typically required. Several tedious steps are currently required to perform routine tasks, such as merging a vegetation cover map derived from a Landsat image with a hydrological map of lakes and streams on a USGS Digital Line Graph.

Flexible and efficient processing systems are required for converting attributes of location in raster data to vector objects that have specific location attributes retrievable by an object-identifier, with explicitly recorded spatial relationships. An earlier, systematic assessment of conversion algorithms notes the use of customized solutions to this problem, and the lack of generally accepted strategies (Peuquet, 1981). Most reviewed procedures originated in image processing, and were published in Pavlidis (1982). All implementations of raster-to-vector conversion follow a two-step procedure of first extracting boundaries (lines, or chains) and then building topology. Extracting boundaries is comprised of two specific subproblems: scanning linework, and digitally representing a surface. In either case, the task is to define a one-cell wide connected set. This is primarily achieved by visiting each grid cell in the input data set, and identifying its attribute-relationship with its neighbors. This is called thinning, or skeletonization with scanned linework, and boundary-finding with surfaces. Clarke (1990) reviews some straightforward algorithms for extracting boundaries. Building topology starts with revisiting extracted lines (i.e., a one-cell wide connected set) and proceeds with identifying and characterizing regions (polygons) enclosed by lines. In principle, a sequential raster-to-vector algorithm runs in time proportional to the number of gridcells (preprocessing) and the length of lines (building topology).

The earliest reported raster-to-vector implementation the authors know of is OEDIPUS (Dutton, 1980). This algorithms processes raster data in three-row blocks, evaluates 8 neighborhoods of each cell, then relies on WHIRLPOOL (Dougenik, 1980) to concatenate segments. POLYVEC in IDRISI (Eastman, 1990) also follows a two-step procedure (Eintwaechter, 1992). Several important cartographic tasks are supported by specialized raster-to-vector conversion (Peuquet, 1981), with all sequential line-extraction approaches (line-following, or line-scanning) requiring processing times that are linearly proportional to total line length, assuming the entire grid can be stored in memory. The cost of "book-keeping" can become overwhelming if the entire map, or the portion containing a line, cannot be held in memory. The final step of building topology is typically independent of the previous steps (Burrough, 1986), and, in most cases, is preceded by some line-smoothing function. More recent implementations follow similar approaches, using for example, neighborhood template-matching (Greenlee, 1987), or Fourier series-based shape descriptions (Illert, 1981), although performance figures are not reported.

Parallel processing is coming to be recognized as an attractive approach to spatial analysis because of the performance improvements possible, and the match between the inherent structure of spatial data, and massively parallel machine architectures. For example, Mower (1992) studied the advantages of both single instruction multiple data (SIMD) and multiple instruction multiple data (MIMD) architectures for spatial analytical algorithms, including labeling drainage basins, and automated line simplification. Franklin et al. (1989) reported a new approach to the polygon overlay problem, and Hopkins et al. (1992) demonstrated

performance improvements and scalability of that algorithm on transputer arrays. Mills et al. (1992) illustrated the potential for expanding the scope of visibility analysis on a parallel system by orders of magnitude over that of previous work on sequential computers. These examples suggest that parallel processing is extremely powerful and general in purpose, and well suited to spatial analysis.

In this study, we implement an algorithm for spatial data conversion on a massively parallel computing system. Two basic approaches to parallel computing include data parallelism, and explicit message passing. In the data parallel model, parallelism is achieved by associating a set of data elements with a set of physical processors, and acting *simultaneously* on the data (Hillis and Steele, 1986). Virtual processing extends this notion of parallelism through simulating an arbitrarily large number of processors by allocating multiple virtual processors per physical processor. Alternatively, in the message passing model of parallel computing, both data and program are distributed to processors which execute *asynchronously* and communicate by sending messages. We define a data parallel algorithm for raster-to-vector data conversion, and implement this algorithm on a SIMD (single instruction multiple data) architecture.

This paper is organized as follows. The methods section describes GORDIUS, the raster-to-vector conversion algorithm in detail. The four steps we use to implement the algorithm are illustrated in commented code and accompanying figures. In the results section, we report on the performance of this parallel conversion algorithm, and the effect of image size, number of polygons, and geometrical complexity of polygon shapes. In the discussion and conclusion section, we summarize the potential for applying parallel computers to real-world problems in geographical information analysis, and describe future plans for applying GORDIUS.

METHODS

An overview of the process for converting data in raster format to a topological vector data structure is illustrated in Figure 1. A simple map (Figure 1A) and its associated raster representation (Figure 1B) must be converted to a vector format (Figure 1C) and represented by a set of points, lines, and polygons encoding their hierarchical (topological) relationships (e.g. starting and ending-node, left and right polygon for lines).

We implement the parallel algorithm in four steps which are detailed below. To map the raster-to-vector conversion problem to a parallel architecture, we take advantage of data distribution and interprocessor communication characteristics of the Connection Machine-2, a massively parallel SIMD architecture. The resulting parallel algorithm is a new approach, rather than simply a conversion of a sequential algorithm. Sections of C* code are listed for each step in the conversion process, and illustrated in accompanying figures.

Description of the algorithm

The principal task in constructing a topological data structure is building a hierarchy of points, lines, and polygons. Points are the only objects associated with spatial coordinates, and are characterized by the

line(s) they belong to. Polygons are characterized by their boundary lines, and lines are the key linking elements having pointers to points (start and end) as well as to polygons (left and right).

We imposed the following set of limitations to simplify raster-to-vector conversion: (1) no more than four lines intersect at any point; (2) all line-segments follow the directions of the axes; (3) homogeneity (i.e., boundary) is defined by equality; and (4) only closed polygons are identified. The first two assumptions are clearly consequences of pure raster representation and may require post-processing (e.g. line generalization). The latter two assumptions, although arbitrary, are realistic land-use/land cover mapping applications based on satellite imagery, and can be upgraded without drastic modifications.

Implementation of the conversion algorithm includes the following steps. First, all cell boundaries are checked for homogeneity, and boundaries, boundary points, and intersection points identified (Figure 2A). The information derived from this step is listed in the first three columns (point identifier, x, and y coordinate) of the Point table in Figure 1C. By summing *boundary_values* illustrated in Figure 2A, we assign a unique value to each raster cell defining its *boundary_type*. Once points are identified, line-segments, (the links between points) are defined (Figure 2B). Identifying the start and end points of each line allows us to define values for the line identifier, the start point, and the end point in the Line table, and the line identifier of the Point table of Figure 1C. Links between points are established by moving from each boundary-point along the four image axes. Finally, polygons are built by starting at any line segment and turning "most clockwise" when at the "end intersection-point"; i.e., turning South for East running lines, West for South running lines, North for West running lines, and East for North running lines (Figure 2C). This step creates the Polygon table, and defines the left polygon, right polygon, and internal points of the Line table.

Implementation of the conversion algorithm

Using a data parallel computing model to implement the algorithm, the two-dimensional raster image is mapped, one cell per processor, to a two dimensional grid on the Connection Machine-2 (16K processors). Program instructions are issued from a front-end computer to all processors which synchronously carry out the operations. Beginning with the gray level values of the raster image (Figures 1A, 1B), we follow four steps to implement the data parallel raster to vector conversion algorithm: step 1 computing boundary type, step 2 connecting boundaries, step 3 making lines, and step 4 making polygons.

Step 1. Computing boundary type

To define the presence/absence of boundaries, we compare the gray level in each cell with the gray levels of North, East, West, and South neighbors. Where gray levels differ, a boundary exists. Fifteen types of boundaries are possible and are uniquely identified by a *boundary_type* value according to the coding scheme illustrated in Figure 2A.

To compare *gray_level* values between a cell and its four neighbors, we use the C* grid communication function *from_torus_dim*. Grid communication occurs in regular patterns based on the coordinates or grid position of the data element, allowing movement of data along axes dimensions everywhere in the raster

image. The expression *boundary_type* += sums the values of the coding scheme illustrated in Figure 2A. In four sequential steps (North, East, South, and West), the unique *boundary_type* value for all cells in the raster image are calculated. For example, cells with only an East boundary have *boundary_type* values of 2; cells with East and South boundaries have *boundary_type* values of 6; and cells surrounded on all four sides have *boundary_type* values of 15. Code segment 1 defines the four steps in computing the *boundary_type*.

C* Code Segment 1. Computing Boundary Type

```
/* Test for North Boundary:
     using the grid communication function from_torus_dim, get gray_level
     of nearest neighbor (one position) to North (negative direction along
     axis 1), where gray_levels differ, a boundary exists, add unique
     coding scheme value to boundary_type */
   neighbor_gray_level = from_torus_dim(&gray_level,1,-1);
     where ( gray_level != neighbor_gray_level  )
       boundary_type += 1;
/* Test for East Boundary:
     get gray_level of nearest neighbor to East (positive direction
     along axis 0), where gray_levels differ, add to boundary_type */
   neighbor_gray_level = from_torus_dim(&gray_level,0,1);
     where ( gray_level != neighbor_gray_level )
       boundary_type += 2;
/* Test for South Boundary:
     get gray_level of nearest neighbor to South (positive direction
     along axis 1), where gray_levels differ, add to to boundary_type */
   neighbor_gray_level = from_torus_dim(&gray_level,1,1);
     where ( gray_level != neighbor_gray_level )
       boundary_type += 4;
/* Test for West Boundary:
     get gray_level of nearest neighbor to West (negative direction
     along axis 0), where gray_levels differ, add to boundary_type */
   neighbor_gray_level  = from_torus_dim(&gray_level,0,-1);
     where ( gray_level != neighbor_gray_level )
       boundary_type += 8;
```

Step 2. Connecting boundaries

To connect boundaries, we use the *boundary_type* values defined in step 1, and a set of nine boundary configurations which define all possible combinations of *boundary_type* values, and occur only where lines start and end. We define a four-cell template around each cell in the raster image according to the pattern

illustrated in Figure 3A. The upper left position in the four-cell template defines the reference cell. The processor mapped to the reference cell stores all values associated with the template. The reference cell has a value of *boundary_type* (defined in step 1). *Boundary_type* values for template positions 2, 3, and 4 within the template are communicated to the reference cell (processor) and assigned to *boundary_type*2, 3, and 4. This set of values is then passed to a set of nine functions to define the boundary configuration within the four-cell template. Two configuration functions are listed in C* code segment 2.

C* Code Segment 2. Connecting Boundaries

```
/* Communicate boundary_type values within four-cell template, assign to
     reference cell at template position 1 (Figure 3 ) */
   void connect_boundaries() {
   int:image boundary_type_2,boundary_type_3,boundary_type_4;
   with(image) { /* select all positions in image */

/* get (from_torus_dim) value of boundary_type from template position 2
     (nearest neighbor in positive direction along axis 1) */
   boundary_type_2=from_torus_dim(&boundary_type,1,1);

/* get (from_torus_dim) value of boundary_type from template position 3
     (nearest neighbor in positive direction along axis 0) */
   boundary_type_3=from_torus_dim(&boundary_type,0,1);

/* in a two steps grid communication (from_torus), get boundary_type
     value from one positive position along axis 0 (template position 4
     to position 2), then one position along axis 1 (template position
     2 to reference cell in Figure 3) */
   boundary_type_4=from_torus(&boundary_type,1,1);

   /*   cross and T_up are two of nine possible configurations */
   cross(boundary_type,boundary_type_2,boundary_type_3,boundary_type_4);
   T_up(boundary_type,boundary_type_2,boundary_type_3,boundary_type_4);
```

The *cross* boundary configuration, one of nine possible configurations within the four-cell template is listed in C* code segment 3. The type of boundary configuration defines whether the Boolean variables *up*, *down*, *left*, and *right* occur within the template. These variables identify the start and end points of lines, and are local to the the function *connect_boundaries* which calls a set of nine boundary configuration functions. In the example illustrated in Figure 3A, a *cross* has *up*, *down*, *left*, and *right*; while a *T_up*

configuration has *up*, *left*, and *right*. The nine configuration functions define boundaries only at the ends of East-West and North-South lines. We do not store intermediate points along a line.

C* Code Segment 3. "Cross" Boundary Configuration

```
void cross(int:image boundary_type_1,int:image boundary_type_2,int:image
  boundary_type_3,int:image boundary_type_4)  {
bool :image temp_bool_1=0,temp_bool_2=0,temp_bool_3=0,temp_bool_4=0;

/* various combinations of boundary_type values determine
   whether up, down, left, right occur within the four cell template */
where(boundary_type_1==6  || boundary_type_1==7  ||
  boundary_type_1==14     || boundary_type_1==15)
    temp_bool_1=1;
where(boundary_type_2==12 || boundary_type_2==13 ||
  boundary_type_2==14     || boundary_type_2==15)
    temp_bool_2=1;
where(boundary_type_3==3  || boundary_type_3==7  ||
  boundary_type_3==11     || boundary_type_3==15)
    temp_bool_3=1;
where(boundary_type_4==9  || boundary_type_4==11 ||
  boundary_type_4==13     || boundary_type_4==15)
    temp_bool_4=1;

/* a cross configuration consists of Boolean variables up and down
   and left and right */
where(temp_bool_1 && temp_bool_2 && temp_bool_3 && temp_bool_4){
  up=1; left=1; right=1; down=1; }
```

Step 3. Making Lines

The Boolean variables *up*, *down*, *left*, and *right* define the end points of lines. The purpose of this step is to connect end points into lines. Making lines is carried out in the following steps and illustrated in Figure 4.

A. Define extent of segmented scan for *East_West_lines*:
Cells where right is active define the western end of a line, and cells where left is active define the eastern end of a line. These cells define the start and stop points for scan lines.

B. Scan *East_West_line*:
A C* scan function spreads the unique *cell_id* value from the western-most cell (where right is active) along

the scan line (Figure 4A). Scan with maximum replaces all *temp_cell_id* values, which were previously set to zero, with a unique value. The unique *cell_id* value is also assigned to the variable *East_West_line* to identify the line.

C. Define *North_* and *South_barriers* along *East_West_line*:
Where *East_West_lines* exist (cells that have non-zero values of temp_point), Boolean *South_barriers* are set to true along the line, and a shift one element position down the North-South axis of the image defines the *North_barrier* (Figure 4B).

Cells where Boolean variables up and down are true are used in a similar process to define *North_South_lines* and associated *East_West_barriers*. The *North_*, *South_*, *East_*, and *West_barriers* are used in the next step to build the polygons.
Using this approach, we simultaneously build all *East_West_lines*, then all *North_South_lines*.

C* Code Segment 4. Making Lines

```
void make_lines() {
bool: image stop = 0;
int:  image temp_cell_id=0;
int i;
with(image) {
/* where statement selects subset of cells with left and right occur,
     defining the East and West ends of lines */
   where(left)
     stop =1;
   where(right) {
     stop =1;
     temp_cell_id =cell_id;} }

/* scan with maximum function replaces all temp_cell_id values
     (which were previously set to 0) in the line with a unique
     cell_id from Western most cell */
  temp_cell_id=scan(temp_cell_id,0,CMC_combiner_max,CMC_upward,
    CMC_start_bit,&stop,CMC_exclusive);
  East_West_line[0]=temp_cell_id;
/*  where East_West_lines exist, a South_barrier is defined,
      a North_barrier is defined by shifting South_barrier one position
      along North-South axis (see Figure 3 B), used to make polygons */
  S_barrier=temp_cell_id;
    to_torus_dim(&N_barrier,S_barrier,0,1);
```

```
/* where statement selects subset of cells with up and down occur,
     defining the North and South ends of lines */
   where(up) { stop =1; }
   else {stop =0;}
   where(down) {
     stop =1;
     temp_cell_id = -cell_id; }
   else{temp_cell_id = 0;}

/* scan with maximum function replaces all temp_cell_id values
     (which were previously set to 0) in the line with a unique
     cell_id from Northern most cell */
   temp_cell_id =scan(temp_cell_id,1,CMC_combiner_min,
     CMC_upward,CMC_start_bit,&stop,CMC_exclusive);
     North_South_line[0] = temp_cell_id;
/* where North_South_lines exist, an East_barrier is defined,
      a West_barrier is defined by shifting East_barrier one position
      along East-West axis, used to make polygons */
    E_barrier=temp_cell_id;
       to_torus_dim(&W_barrier,E_barrier,1,1); } }
```

Step 4. Making Polygons

The remaining step is to build polygons from lines. This task requires grouping distinct elements into a collection of disjoint sets, and is a form of the connected component problem (Cormen et al., 1991). As in other applications of the connected component problem described in the image analysis literature (Gay, 1985; Dunlavey, 1983), we build polygons by filling the area bounded by a closed set of lines with a unique polygon identification value.

At the beginning of this step, each cell is assigned a unique *cell_id*, and each cell is initially considered a polygon. Through a sequence of segmented scans (down, up, right, left) we spread a minimum *cell_id* value throughout the region bounded by a set of *N_*, *E_*, *S_* and *W_barrier*s, uniquely labeling the polygon. Segmented scans are parallel operations, allowing us to simultaneously build all polygons in the image (see Figure 4).

Once the polygons are made, we extract neighborhood information– which polygons are adjacent– by listing the line, *left_* and *right_polygon* for each cell at the end of either an *East_West_line* or a *North_South_line*. For example, a cell at the West end of an *East_West_line* (where left is true) is illustrated in Figure 5. The line number for this point is uniquely identified by the line running between

polygon_id a and *polygon_id* b in this example. The polygon to the left of this line is stored as the *polygon_id* of the cell indicated in Figure 5, and the right polygon is obtained by communication with the cell below the active cell.

C* Code Segment 5. Making Polygons

```
make_polygons () {
bool : image unfinished_polygon=1;
int  : image temp_min_cell_id;

with(image) {
  temp_min_cell_id= cell_id;
  /* while loop continues if any unfinished_polygon exists */
  while (|=(unfinished_polygon == 1)) {
  polygon_id=temp_min_cell_id; /* each cell is initially a polygon */

  /* a sequence of scan operations spreads a unique cell_id throughout
      region bounded by the North_ South_, West_ and East_ barriers,
      polygons are uniquely labeled */

  temp_min_cell_id = scan(temp_min_cell_id,1,CMC_combiner_min,CMC_upward,
    CMC_start_bit,&N_barrier,CMC_inclusive);
  temp_min_cell_id =scan(temp_min_cell_id,1,CMC_combiner_min,
    CMC_downward,CMC_start_bit,&S_barrier,CMC_inclusive);
  temp_min_cell_id = scan(temp_min_cell_id,0,CMC_combiner_min,
    CMC_upward,CMC_start_bit,&W_barrier,CMC_inclusive);
  temp_min_cell_id = scan(temp_min_cell_id,0,CMC_combiner_min,
    CMC_downward,CMC_start_bit,&E_barrier,CMC_inclusive);

  /* unfinished polygons have different minimum cell and polygon id values */
  unfinished_polygon = (temp_min_cell_id != polygon_id);
}

  /* extract neighborhood information by getting polygon id from
       adjacent polygons */
  East_West_line[1]=from_torus_dim(&temp_min_cell_id,1,1);
  East_West_line[2]=temp_min_cell_id;
  North_South_line[1]=temp_min_cell_id;
```

```
  North_South_line[2]=from_torus_dim(&temp_min_cell_id,0,1);
} }
```

RESULTS

Communication is the important parallel computing issue in this application. Our spatial data conversion algorithm is based on grid communication operations–nearest neighbor, and segmented scans, which are implemented in hardware and optimized for speed on the Connection Machine-2. Every processor can send data to its immediate neighbor in an n-dimensional grid. Scans require processing time proportional to O(log N) where N is the communication distance or size of the raster image. Performance of our data parallel conversion algorithm is not sensitive to the number of polygons being built from the raster image. In general, the algorithm is sensitive only to the size of the image, and not to the number of polygons, nodes, or neighbors.

In step 1, we compare the gray level of each cell with its four nearest neighbors to define a boundary type. As each cell computes this simultaneously, the algorithm is insensitive to the number of points. In step 2, we again use nearest neighbor communication to simultaneously construct a four-cell template, and define a boundary type for each cell in the image. Then we call a sequence of boundary configurations to define the start and end points of lines. In step 3, we build lines using scan functions that simultaneously connect all East-West lines, followed by all North-South lines. In step 4, we fill the region bounded by a set of lines to label the polygon. All polygons are labeled simultaneously.

Polygon shape has an impact on the number of iterations required to fill the polygon with a unique *cell_id*, labeling the polygon. Polygons having a regular geometry, such as the example illustrated in Figure 4, require a minimum of two iterations of the polygon fill loop listed in code segment 5; as polygons become more irregular, a larger number of iterations are required. In a preliminary analysis of the effect of polygon shape on performance of the algorithm, we set up an experiment to generate images with random gray values, and timed program performance as a function of polygon shape. Using a fixed image size of 128x128 elements, we varied the number of gray levels to produce images with relatively complex shapes (a smaller number of polygons with a higher degree of spatial autocorrelation) vs. relatively simple shapes (a larger number of polygons with a lower degree of spatial autocorrelation). As Figure 6A illustrates, better program performance is associated with relatively simple polygon shapes (i.e., low points/polygon ratio). In contrast to our parallel algorithm, sequential algorithms for raster to vector conversion are sensitive to the number polygons in the image (Figure 6B). We used the routine POLYVEC in IDRISI, which provides boundaries but not topology (Eastman, 1990) strictly to illustrate the sensitivity between sequential algorithm performance and number of polygons.

To examine the effect of image size on algorithm performance, we generated a set of images with random gray level values, this time varying the size of the image. For the Connection Machine-2 (16K), we examined program performance as a function of image sizes for 128x128, 128x256, 256x256, and 512x512 elements (run time for the 512x512 image took approximately 7.5 seconds and is not shown in Figure 7

for scaling purposes). In comparison, a sequential raster to vector conversion code, which does not include topology, ran in 64 seconds on a SPARCstation IPC for a 256x256 image (Li, 1992).

As Figure 7 illustrates for a set of experiments, program running time grows as a sub-linear function of the number of raster elements. This efficiency is due to the virtual processing mechanism of the Connection Machine-2, which allows a single physical processor to simulate a number of "virtual" processors. While each virtual processor stores a raster element, grid based communication between processors is supported by hardware for efficient nearest-neighbor accesses and scans along grid axes.

DISCUSSION AND CONCLUSION

More than a decade ago, Peucker and Chrisman (1979) noted the lack of "flexibility, comparability and topology" in data structures for geographic (or spatial) databases. Since then, a considerable amount of effort has been devoted to the extraction and creation of topological information–the position of a geographic entity with respect to its neighboring entities, from "pure" coordinate data. Associated issues include storage, manipulation and processing of topological information for digital cartographic purposes (Peuquet, 1981).

Our data parallel algorithm for raster to vector conversion provides a very fast solution to a well-specified "real-world" problem. Parallel implementation of computationally intensive algorithms, and their application to large data sets provides a flexibility of analysis with the potential for opening new lines of inquiry, such as "transparent" data representation (Goodchild, 1991). Future improvement of the algorithm will include more sophisticated homogeneity criteria, e.g. variance-based measures. We plan to apply the algorithm to a suite of spatial data sets to further examine the issue of polygon geometry. Preliminary simulation studies suggest that parallel computing is an extremely powerful tool for error modeling (Csillag, 1992.) We view development of an industry-scale GIS project on a massively parallel computing system as an important next step in our research and development plans.

A research issue for the future is a comparison of the performance of our algorithm on the Connection Machine-2 with its performance on the Connection Machine-5 and DECmpp-12000. Implementation on the CM-5 introduces the issue of data parallel programming on a MIMD architecture. Implementation on the DECmpp- 12000 will allow us to compare the performance of grid-based communications on a hypercube topology (CM-2) with a mesh topology (DECmpp). Conversion of the algorithm to a MIMD architecture will raise issues of how to distribute the data over a number of processors, implement sequential algorithms for data conversion within nodes, and matching up subregions within each node with homogeneous sub-regions of neighboring nodes (identifying polygons that span subregions), and maintaining a global list of *polygon_id*s.

REFERENCES

Burrough, P.A., 1986, Principles of geographical information systems: Clarendon Press, Oxford, p. 63.

Clarke, K.C., 1990, Analytical and digital cartography: Prentice Hall, Englewood Cliffs.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L., 1991, Introduction to algorithms: Chapter 22: Data structures for disjoint sets, MIT Press, Cambridge, Massachusetts, p. 440-461.

Csillag, F., 1992, Linking error models of fields and objects: a simulation study, unpublished manuscript.

Dougenik, J., 1980, WHIRLPOOL: a geometric processor for polygon coverage data: Proceedings Auto-Carto 4, p. 304-311.

Dunlavey, M., 1983, Efficient polygon-filling algorithms for raster displays: ACM Transactions on Graphics, v. 2, no. 4, p. 264-273.

Dutton, G. ed., 1980, Current research at the laboratory for computer graphics and spatial analysis, Harvard Graduate School of Design, Cambridge, MA, p.29.

Eastman, R., 1990, IDRISI 3.2.2., A grid-based geographic analysis system, Clark University, Worchester, MA.

Eintwaechter, T. (1992) personal communication.

EOSAT, 1991, Products and services catalog: Earth Observation Satellite Company, Iselin, NJ.

Franklin, W.R., Narayanswami, C., Kankanhalli, M., Sun, D., Zhou, M, and Wu, P., 1989, Uniform Grids: A technique for intersection detection on serial and parallel Machines. Proc. AutoCarto-9, ACSM-ASPRS, Falls Church, Virginia. pp. 100-109.

Gay, A. C., 1985, Experience in practical implementation of boundary- defined area fill, *in* NATO Advanced study institute on fundamental algorithms for computer graphics, Earnshaw, R. A., ed. p. 153-160.

Goodchild, M.F., 1991, Geographical information science: Int. J. Geog. Info. Systems, v. 6, p. 31-45.

Greenlee, D.D., 1987, Raster and vector processing for scanned linework: *in* Chrisman, N., ed. Proceedings Auto-Carto 8, p. 640- 647., American Congress on Surveying and Mapping, Bethesda, MD.

Hopkins, S., Healey, R.G., and Waugh, T.C., 1992, Algorithm scalability for line intersection detection in parallel polygon overlay. Proc. 5th International Symposium on Spatial Data Handling, IGU Comm. GIS, Charleston, South Carolina. pp. 210-218.

Hillis, W. D., and Steele, G. L., 1986, Data parallel algorithms: Comm. ACM, v. 29, no. 12, p. 1170-1183.

Illert, A., 1991, Automatic digitization of large scale maps: *in* Mark, D., and White, D., eds. Technical Papers Auto-Carto 10, p. 113-123., American Congress on Surveying and Mapping, Bethesda, MD.

Li, B., 1992, Opportunities and challenges of parallel processing in spatial data analysis: initial experiments with data parallel map analysis, *in* GIS/LIS '92. ASPRS, ACSM, AAG,URISA, AM/FM. November 10-12, San Jose, CA, p. 445-458.

Mark, D.M. and Csillag, F., 1989, The nature of boundaries on area-class maps: Cartographica, v. 26, p. 65-78.

Marx, R.W., 1990, The TIGER system: automating the geographic structure of the United States Census, *in* Peuquet, D. Marble, D.F., eds. Introductory readings in geographic information systems: Taylor and Francis, London, p. 120-141.

Mills, K., Fox, G., and Heimbach, R., 1992, Implementing an intervisibility analysis model on a parallel computing system, *Computers & Geosciences*, v. 18, no. 8, p. 1047-1054.

Mounsey, H. and Tomlinson, R., eds., 1989, Building databases for global science: Taylor and Francis, London.

Mower, J. E., 1992, Building a GIS for parallel computing environments, Proc. 5th International Symposium on Spatial Data Handling, Aug. 3-7., IGU Comm. GIS, Charleston, South Carolina. pp. 219-229.

Pavlidis, T., 1982, Algorithms for graphics and image processing: Springer Verlag, Berlin.

Peucker, T.K. and Chrisman, N., 1979, Cartographic data structures: The American Cartographer, v. 2, p. 55-69.

Peuquet, D., 1981, An examination of techniques for reformatting digital cartographic data, Part 1: The raster-to-vector process: Cartographica, v. 18, p. 34-48.

Peuquet, D. J., 1984, A conceptual framework and comparison of spatial data models: Cartographica, v. 21, p. 66-113.

Thinking Machines Corporation, 1990, C* Version 6.0 programming guide: Thinking Machines Corporation, Cambridge, Massachusetts. 254 p.

USGS, 1983, Digital cartographic data standards: Geological survey circular 895, U.S. Geological Survey, Reston, VA.

LIST OF FIGURES