Implementing GIS Procedures on Parallel Computers: A Case Study

James E. Mower
Department of Geography and Planning
147 Social Sciences
State University of New York at Albany
Albany, New York 12222
jmower@itchy.geog.albany.edu

ABSTRACT

The development of efficient GIS applications on parallel computers requires an understanding of machine architectures, compilers, and message passing libraries. In this paper, the author compares performance statistics for implementations of drainage basin modeling, hill shading, name placement, and line simplification procedures under control-parallel and data-parallel approaches. The suitability of each approach and its message passing strategies are discussed for each application. The paper concludes with caveats for the developer concerning the current state of parallel programming environments.

INTRODUCTION

Parallel computing tools are being increasingly exploited by the GIS community for tasks ranging from line intersection detection (Hopkins, S., R.G. Healey, and T.C. Waugh 1992) to the development of network search algorithms (Ding, Y., P.J. Densham, and M.P. Armstrong 1992). GIS professionals who turn to parallel computers for fast solutions to large computing problems are confronted with an array of architectural designs not encountered in sequential computing. Choosing an inappropriate parallel architecture or computing model can result in little or no performance benefits over a modern sequential computer. This paper will illustrate the suitability of various parallel computing models for GIS applications by examining the performance characteristics of code written by the author for:

1) drainage basin analysis,
2) analytical hill shading,
3) cartographic name placement, and
4) line simplification.

It will show that the appropriate selection of a parallel computing architecture and programming environment is essential to the visualization and efficient solution of the problem.

The differing computational demands of each problem domain highlight the strengths of competing parallel architectures. The name placement procedure, making extensive use of general interprocessor communication services, is contrasted with the hill shading procedure that uses simple grid communication facilities. Two procedures based on the Douglas line simplification

algorithm (Douglas and Peucker 1973), one written as a control-parallel procedure and the other as a data-parallel procedure, show the relative costs of message passing on multiple instruction stream, multiple data stream (MIMD) computers and on single instruction stream, multiple data stream (SIMD) computers.

The procedures described in this paper were implemented on Thinking Machines CM-2 and CM-5 computers. The CM-2 is a true SIMD computer, consisting of up to 64K bit-serial processors connected by hypercube and grid communication networks. The CM-5 is a synchronous-asynchronous multiple data (SAMD) computer, capable of operating in both control-parallel and data-parallel modes. The name placement procedure was implemented on a CM-2; all the other procedures were implemented on a CM-5.

Where appropriate, performance statistics are provided to show the consequences of making specific design choices. Particular emphasis will be given to the performance of the control-parallel and data-parallel variants of the Douglas algorithm, both running on the CM-5.

If a programming environment lacks a structure for stating clear solutions to a class of problems, attaining marginal increases in performance may be inconsequential to the developer. Comparisons of the Douglas implementations will show that the control-parallel model better captures the elegance of the sequential algorithm.

MACHINE ARCHITECTURES AND PROGRAMMING MODELS

This paper will examine the procedures in the context of SIMD and MIMD architectures. SIMD machines support data-parallel programming. Under this approach, a control or front-end processor broadcasts instructions to a parallel processor array. All processors in the array operate synchronously, each executing or ignoring the current instruction, depending upon the state of its local data. The data-parallel procedures described here are implemented in the C* programming language, a superset of ANSI C with data-parallel extensions.

MIMD machines allow processors to run asynchronously on their own instruction stream. In a master-worker control-parallel model, each worker processor runs an identical copy of a program as if each were a separate computer, exchanging data with the master through message passing operations. Because the instruction streams are independent, the rate at which each worker executes its instructions is determined by the length of its data stream (Smith 1993).

Workers notify the master processor when they are ready to receive new work or when they are ready to send completed work. During a cooperative or synchronous message passing operation, the receiving processor remains idle until the sending processor is ready to pass its message.

IMPLEMENTING THE PROCEDURES

Drainage basin modeling and hill shading

The data-parallel drainage basin and hill shading procedures map elevation samples in a U.S.G.S. 1:24,000 series DEM to processors arranged in a 2-dimensional grid. The procedures execute the following computations:

1) removal of most false pits through elevation smoothing,
2) calculation of slope and aspect to determine cell drainage direction,
3) propagation of drainage basin labels,
4) removal of remaining false pits through basin amalgamation,
5) location of stream channels through drainage accumulation, and
6) calculation of hill shaded value from cell slope and aspect and light source azimuth and altitude.

To take full advantage of a SIMD machine, a data-parallel program must execute instructions on all of the array processors, and hence on all of its input simultaneously. For drainage basin analysis and hill shading, Mower (1992) shows that steps 1, 2, and 6 meet this criterion. Typically, though, many data-parallel procedures operate on a subset of the processors over any particular instruction. Steps 3, 4, and 5 all require that values propagate away from a set of starting cells. In steps 3 and 4, drainage basin labels propagate away from pits. On the first iteration, only processors associated with pits are active. On subsequent iterations, progressively larger numbers of processors are activated along the 'wave front' expanding away from each pit; cells that were active on the previous iteration become inactive (Figure 1).

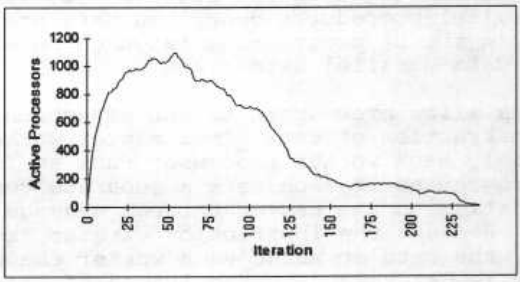

Figure 1. Processors that are active over each iteration of step 3 for a 1:24,000 DEM sampled at 30 meter intervals. The entire DEM is represented by approximately 120,000 processors.

Step 5 promotes an opposite pattern of activation. On the first iteration, each cell starts with one unit of water and queries its 8-case neighbors for cells that drain toward itself. After accumulating the water supplies of

the uphill neighbors, the next iteration begins. If a cell no longer finds uphill neighbors with a positive water supply, it becomes inactive. The number of active processors gradually declines until the end of the step when none but those associated with pits are active (Figure 2).

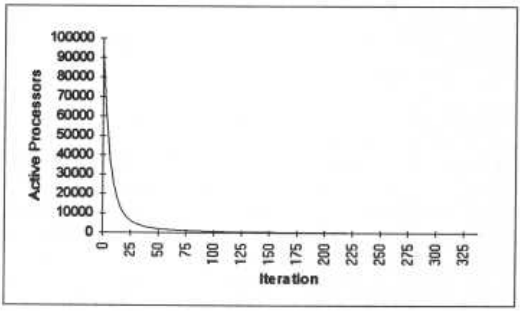

Figure 2. Active processors over each iteration of the drainage accumulation procedure for the same size window as Figure 1.

Steps 1, 2, and 6 of the data-parallel algorithms for drainage basin modeling and hill shading are relatively simple to implement on a SIMD machine. They are also fast—hill shading an entire 1:24,000 DEM requires less than one second. For each step, the iterative control loop of a sequential implementation is replaced with a parallel context operator that restricts the computations to all but the edge cells of the matrix. Operations that require values from 8-connected neighbors receive them through fast grid communication functions. Steps 3, 4, and 5 are harder to implement, requiring that processors determine their state of activation with respect to their local data.

Operations that refer to a data object larger than the grid cell are sometimes clumsy or inefficient to implement. To find the pour point for a drainage basin in step 4, cells on the edge of the basin are activated. A scanning function finds the pour point as the activated cell with minimum elevation that also drains to a basin with a lower pit than its own. Finally, cells in the basin with elevations below the pour point are activated to raise them to the level of the pour point in a flooding procedure. These operations pertain to a small percentage of the total cells in the DEM, leaving the rest of the processors inactive over their duration.

The author is currently developing a control-parallel algorithm for drainage basin modeling, using the drainage basin, rather than the grid cell, as the basic data object. This approach applies a standard suite of drainage basin modeling procedures to each basin as defined initially by its pit. Using a master-worker model, the master processor assigns pits to workers as they become available. Each worker runs asynchronously on

its basin, requesting a new basin from the master on completion.

This approach will perform best on large regions containing many basins. For smaller regions having fewer basins than processors, dissimilarities in running time among the workers will lead to relatively low efficiency. This approach also lacks the elegance of the data-parallel procedure with regard to steps 1, 2, and 6, ignoring their inherent parallel structures.

Name placement

The name placement procedure employs a data-parallel model equating processors with the locations of populated places. (Mower 1993a). These features are distributed irregularly across the earth and cannot be represented efficiently in a two-dimensional array. Instead, their processors are represented as a vector. Since the position of a processor in the vector array does not implicitly represent the geographic coordinates of its place, the coordinates must be stored explicitly.

As the procedure scans the database for places that fall within the user's window, it determines the neighborhood for each place as the total area over which its label could be located. Places later compare their neighborhoods against one another for intersection. If a place finds an overlapping neighborhood, it adds the processor identifier of the overlapping place to its list of neighbors. Currently, data is read from modified U.S.G.S. Geographic Names Information System (GNIS) files of populated places for names appearing on 1:24,000 series maps.

For its point symbol and label to occupy non-overlapping map locations, each place queries the locations of the point markers and labels of its neighbors. Since the geographic location of each place is arbitrary with respect to the position of its processor in the list, interprocessor communication must occur over a general communication network. Depending upon the topology of the network and upon the relative addresses of the processors, messages will require varying numbers of steps to reach their destination. If messages contend for a limited number of network router nodes, the overall time to pass them will increase. Therefore, general communication functions frequently take longer to perform than do grid functions, usually requiring one step.

After querying the user for an input window and map scale, the name placement procedure begins by placing all point features (currently populated places) that fall in the user's window onto the map with their labels to the upper right of their point symbols. Depending on map scale, feature density, feature importance, type size, and point symbol size, a number of feature symbols and labels will overlap. Each processor checks the features in its neighborhood for overlap conditions. If it finds that its label overlaps one or more features of greater importance

(currently determined as a simple function of population), it tries moving its own label to an empty position. If no such position exists, it tries moving its label to a position that is occupied by no features of greater importance than itself. If that doesn't work, the feature deletes itself. All features iterate through this cycle until no conflicts remain.

The pattern of processor activation for name placement is quite different from those for drainage basin analysis and hill shading which are generally similar across data sets. The running time of the name placement program varies directly with the maximum number of neighbors found by any processor. This is a function of map scale—at some very large scale, no neighborhoods overlap; at some very small scale, all neighborhoods overlap. As map scale decreases, the maximum number of neighbors found by any map feature increases, requiring the activation of a greater number of processors. The number of active processors and the lengths of the neighbor lists decline as overlaps are resolved through label movement or feature deletion, increasing the execution speed of subsequent iterations.

The author compared the performance of the SIMD version implemented on a CM-2 to a functionally equivalent sequential version running on a Sun Microsystems SPARCstation 2. For a 1:3,000,000 scale map of New York State, the SPARCstation required one hour and 25 minutes. The same map completed in slightly over 4 minutes on the CM-2. The author also found that the running time of the CM-2 version increased linearly at a small constant rate with the maximum number of neighbors (as a function of map scale) of any mapped feature. With a 20 fold increase in the maximum number of neighbors, running time increased by a factor of only 2.15 (Figure 3).

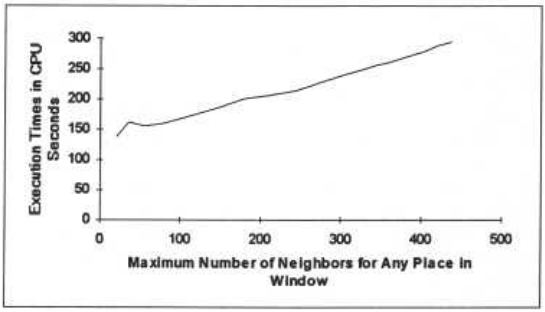

Figure 3. Graph of execution time as a function of the maximum number of neighbors for any place on the map.

Line simplification

The Douglas algorithm for line simplification is implemented in both data-parallel and control-parallel approaches. Under the data-parallel approach, each processor represents the vertex of a cartographic line, extracted from U.S.G.S. 1:2,000,000 series DLG files.

Under the control-parallel approach, copies of a simplification procedure run asynchronously on each processor. Two versions of this approach are implemented. In the first version, each processor is responsible for opening a line file, reading its contents, simplifying all lines in the file within the user's window, and writing the output to a global file. In the second version, a master processor performs all input and distributes work, segmented by lines, to the next available worker processor that requests it. Each worker simplifies its line and writes its output to a global file. The master and the workers negotiate the distribution of work and data through synchronous (cooperative) message passing.

Under the data-parallel approach, the set of processors dedicated to a line are divided into scan sets. A scan set consists of processors representing the starting node, the ending node, and the vertices between them. A processor representing a starting node finds the equation of the baseline connecting itself to the ending node. The vertex processors calculate their perpendicular distances to the baseline. A scanning function on the front-end finds the vertex processor in each scan set having the greatest calculated distance from its baseline, marking it significant if the distance is greater than the user's tolerance value. Each significant vertex segments the line into new scan sets. The procedure continues until it finds no new significant vertices.

Of the three approaches to line simplification, the first is the easiest to implement. It simply applies a recursive sequential implementation of the Douglas procedure separately to each file that happens to intersect the user's window. No message passing is required between the master processor and a worker once its file is opened and processing begins. For each file, the number of lines per unit area and the amount of overlap between its region and that of the user's window determine the length of its processing time. Given an even distribution of lines across the region, files overlapping the edge of the user window will finish sooner than those in the middle of the window.

The second approach uses a master-worker strategy to balance the workload more evenly across the processors. It also runs a recursive sequential implementation on each node but segments the data by line rather than by file. The master processor polls the workers for any that are idle, sending each available worker an individual line to simplify. When a worker has completed its line, it writes it to a global output file.

The second approach requires 2 types of messages to be sent: 1) a message from a worker announcing that it is ready to receive a line for processing and 2) a message from the master sending a worker a line to simplify. Both messages are passed cooperatively, requiring the receiving partner to remain idle until the message originates from the sending partner.

Mower (1993b) compares execution times for the data-parallel and control-parallel approaches to the Douglas algorithm. The parallel approaches were compared to a sequential version built from the control-parallel-by-file code. To control for differences in operating system performance, the sequential version was linked to the same parallel libraries as the control-parallel versions but run on a single CM-5 processor.

Surprisingly, the control-parallel version segmented by files and the sequential version both performed substantially faster than the version segmented by lines. Although the latter provides better load balancing, it does so through expensive message passing operations. Figure 4 compares the amount of time required to execute each version on varying numbers of lines. Except for message passing and I/O operations, the source code for the three versions are identical.

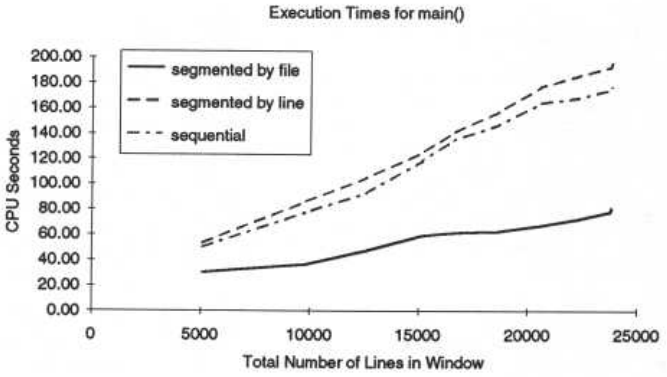

Figure 4. Graph of execution times for each of three parallel implementations of the Douglas Procedure.

Because test results of a data-parallel prototype implementation gave very slow execution times compared to the control-parallel prototypes, it was not developed to a final testing version. The cause for its poor performance can be attributed mainly to its inefficient use of SIMD processors. The prototype implementation processes one line at a time, leaving the majority of the processors idle. The procedure would activate a greater number of processors if more lines were processed simultaneously but would still perform the same number of sequential scans over each scan set. It is unclear whether this would lead to a substantial improvement in its performance characteristics.

CONCLUSION

This paper has shown the performance benefits and liabilities incurred through the application of various parallel architectures, programming approaches, and

message passing strategies to drainage basin modeling, hill shading, name placement, and line simplification implementations. Specifically, it found that:

1) data-parallel procedures execute fast when all processors are kept active and message passing is restricted to grid operations;

2) data-parallel procedures may perform slower than equivalent sequential procedures when the conditions in 1) are not met;

3) synchronous message passing operations slow execution substantially in control-parallel procedures; and

4) of the procedures reviewed in this paper, name placement and hill shading currently offer the best performance improvements over equivalent sequential procedures.

Some observations on parallel programming

SAMD machines provide the necessary flexibility for implementing the procedures described in this paper, offering control-parallel and data-parallel programming models under a variety of message passing schemes. At the moment, however, some of these tools work better than others. The author's experience in the development of parallel procedures for GIS applications on the CM-5 has led to the following observations on its programming environments:

1) Parallel programming languages and compilers are changing rapidly. Modern parallel computers support parallel versions of several high-level programming languages including C, FORTRAN, and Lisp. A manufacturer will generally supply new compilers with new or updated platforms. Changes from one version of a compiler to the next may require the user to spend many hours debugging or optimizing code for the new environment. The author has found that the best defense against compiler changes is to use the simplest data structures and control structures that the language offers. Unfortunately, many of the new compilers are actually beta test versions and do not perform good code optimization. In that case, the user must select those language features that are known to compile into the fastest executable code. Expect to rewrite your code frequently.

Sometimes languages themselves change. Early versions of the C* programming language implemented by Thinking Machines, Inc. looked very much like C++. After version 6, the language was completely rewritten and now looks much more like standard C. Most of the parallel extensions changed completely between versions.

2) Debugging is difficult. Data-parallel debuggers have become much more usable with the introduction of X-Windows-based data visualization tools. Some of these

tools are still limited in their ability to show members of parallel structures or elements of parallel arrays. Traditional debugging methods using print statements from within a program may not be successful in printing values of parallel or local variables. As a result, the programmer must often resort to indirect techniques such as introducing scalar debugging variables within a program.

Control-parallel debugging with print statements is somewhat easier to perform since the processors run standard versions of sequential programming languages. In this environment, the programmer generally tries to reduce the amount of debugging output or to simplify it. On a MIMD machine of 32 processors, the output of a single print statement in straight-line code would appear 32 times, once for each copy of the program. For procedures with large numbers of print statements embedded within complex control structures, the output of the statements can be intermingled, making for confusing reading.

The relevance of these observations will fade as parallel programming environments stabilize. With an understanding of their current limitations, the GIS developer can use these tools to bring about large increases in the performance of spatial data handling procedures.

REFERENCES

Ding, Y., P.J. Densham, and M.P. Armstrong 1992, Parallel Processing for Network Analysis: Decomposing Shortest Path Algorithms for MIMD Computers, Proceedings, 5th International Symposium on Spatial Data Handling, Charleston, pp. 682-691.

Douglas, D.H. and T.K. Peucker 1973, Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature, The Canadian Cartographer, 10(2):112-122.

Hopkins, S., R.G. Healey, and T.C. Waugh 1992, Algorithm Scalability for Line Intersection Detection in Parallel Polygon Overlay, Proceedings, 5th International Symposium on Spatial Data Handling, Charleston, pp. 210-218.

Mower, J.E. 1992, Building a GIS for Parallel Computing Environments: Proceedings, 5th International Symposium on Spatial Data Handling, Charleston, pp. 219-229.

Mower J.E. 1993a, Automated Feature and Name Placement on Parallel Computers, Cartography and Geographic Information Systems, 20(2):69-82.

Mower J.E. 1993b (manuscript submitted for review), SIMD Algorithms for Drainage Basin Analysis.

Smith, J.R. 1993, The Design and Analysis of Parallel Algorithms, Oxford University Press, Oxford.