Automatic Mapping and Load Balancing of Pointer-Based Dynamic Data Structures on Distributed Memory Machines

Robert P. Weaver
and
Robert B. Schnabel

CU-CS-584-92 February 1992

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

To appear in *Proceedings of 1992 Scalable High Performance Computing Conference*, IEEE Press, April 1992.

Automatic Mapping and Load Balancing of Pointer-Based Dynamic Data Structures on Distributed Memory Machines[1]

Robert P. Weaver
and
Robert B. Schnabel

CU-CS-584-92 February 1992

Department of Computer Science
Campus Box 430
University of Colorado at Boulder
Boulder, Colorado 80309-0430 USA
weaver@cs.colorado.edu
bobby@cs.colorado.edu

[1]Research supported by Boeing Computer Services, NSF Grant No. CDA-8922510, and NSF Grant No. ASC-9015577.

Abstract

We describe an algorithm for automatically mapping and load balancing unstructured, dynamic data structures on distributed memory machines. The algorithm is intended to be embedded in a compiler for a parallel language (DYNO) for programming unstructured numerical computations. The result will be that the mapping and load balancing will be transparent to the programmer.

The algorithm iterates over two basic steps: (1) It identifies groups of nodes ("pieces") that disproportionately contribute to the number of off-processor edges of the data structure and moves them to processors to which they are better connected. (2) It balances the loads by identifying groups of nodes ("flows") that can be moved to adjacent processors without creating new pieces.

Our initial results are promising, giving us good load balancing and a reasonably low number of inter-processor edges.

1. Introduction

We are developing a C-based parallel language that is designed to support numerical computation with dynamically changing data structures on distributed memory machines. The language, called DYNO, allows the programmer to build a pointer-based data structure as the basic data structure of the computation. During the computation, the programmer can access data from any node of the data structure that is reachable by pointers, and can add nodes to and delete nodes from the data structure, as well as modify connections between nodes. The distribution of the data structure across the processors of the parallel machine is handled by the compiler/run-time-system during the computation in a manner that is transparent to the programmer.

The efficiency of the resulting parallel program depends critically upon two factors: (1) grouping neighboring nodes of the dynamic data structure together on a processor, i.e. minimizing the number of pointers that cross between processors, and (2) balancing the computational load between processors. This paper presents a heuristic, dynamic algorithm that is intended to accomplish both of these goals using no program-specific information except the connectivity of the dynamic data structure. To our knowledge, this is the first time that such a general-purpose mapping and load balancing algorithm has been embedded in a data-parallel programming language for scientific computation and its associated run-time system.

2. Related research

While a number of research groups have explored languages for parallel programming on distributed memory machines, few have offered support for unstructured or dynamic data. Chief among the projects that have addressed this issue are Kali [7], the PARTI/ARF project at ICASE [4], FORTRAN D [6], and Vienna FORTRAN [2]. Of these, the closest to our research is the PARTI/ARF project in which the compiler generated run-time system links to a user-supplied data partitioner.

One of the difficulties in providing support for unstructured and dynamic data in the compiler/run-time-system is that the designer has little choice but to provide some support for mapping and load balancing. Furthermore an optimal solution to the "mapping problem" is well known to be NP complete [1]. Many researchers have proposed heuristic algorithms for mapping such data structures onto processors for the case in which it is important to minimize the number of inter-processor links. Examples can be found in [3], [5], and [8]. An example of a scheduler embedded in a more conventional language with explicit message passing is described in [9]. Most of these approaches are not general enough to solve the problem we have posed. The ones that have sufficient generality are not efficient, i.e. they exhibit $O(N^2)$ behavior or worse.

3. An overview of DYNO

DYNO (DYnamic diNO) is a C based language, and is related in this and several other ways to our earlier language for numerical computation on regular data structures on distributed memory machines, DINO [10]. The starting point for DYNO is the concept of a pointer being able to point to an object on another (virtual) processor. The programmer might typically define a distributed graph, using C structs as the nodes connected by distributed pointers as the edges. In the simplest case, every pointer crosses a processor boundary, so there will be a virtual processor for each node in the graph, that is, the data structure implicitly defines a virtual parallel machine. To the extent that the programmer wants to think of the "shape" or "structure" of the virtual machine, it has the same "structure" as the data structure does. The programmer can also combine distributed and regular pointers so that each virtual processor has many nodes of the data structure.

DYNO is an explicitly parallel language, as opposed to a serial language with annotations that help the compiler parallelize the code. The programmer writes one piece of code that will execute on each virtual processor concurrently. That is, the language utilizes a single-program multiple-data paradigm, applied to a dynamic data structure.

Most communication, i.e., access to a value that is not on the virtual processor that is executing, is automatic. The compiler generates the necessary communications for any reference to a value that is made via a distributed pointer. In order to insure that communication during the computation is deterministic, DYNO imposes a requirement that a virtual processor can read any data but can only write to its own data, i.e., not to data accessed by distributed pointers. While this restriction may affect the way in which the programmer writes code, it does not restrict the range of computation the language can express. We relax this requirement within the `MODIFY` statement in order to allow the programmer to change the data structure, as discussed below.

For its communication model, DYNO uses a variation on the "copy in/copy out" model that has been used in languages such as Kali [7]. In any "block" of code, an off-virtual-processor access of data produces the value that was current at the beginning of the block. DYNO allows the programmer to specify the beginning of each such "block" using an "`UPDATE`" statement. This permits the programmer to control the grain size of the communication. An important difference between DYNO and Kali is that, because a virtual processor can only write to its own data, there is no "copy out" part.

DYNO also provides a mechanism for the programmer to add nodes, delete nodes, or change connections between nodes of the distributed data structure while the program is executing. DYNO provides a "`MODIFY`" statement, within which a programmer may change distributed pointers. Because a modification must affect several virtual processors simultaneously, the programmer can write to other virtual processors than the current one. To preserve some measure of determinism we use additional semantics within the `MODIFY` statement that we do not discuss here.

A prototype DYNO compiler is currently under development.

4. Mapping and load balancing

The DYNO compiler and run-time system must provide substantial support to implement the programming model summarized above. First, the compiler and run-time system has to restructure the program to contract virtual processes to real processors. Second, it has to handle communication automatically. Third, is has to do automatic mapping and load balancing of distributed data structures both initially and after each `MODIFY` statement. The third of these tasks is the most challenging and is the focus of this paper.

Since our algorithm treats the virtual processor structure as an undirected graph where the nodes are virtual processors and the edges are connections between virtual processors, we will use the term "node" for a virtual processor, "edge" for a connection, and "processor" for a real processor for the remainder of this paper. We will use "N" for the number of nodes and "E" for the number of edges.

We have set four main goals for the mapping/load balancing algorithm:

- Balance the number of nodes on each processor within a small percentage.
- Make the number of edges crossing processor boundaries fairly close to the minimum possible value.
- Keep the amount of computation tractable, $\mathbf{O}(N)$ if possible.
- Use little or no programmer supplied information.

In order to make our initial approach to the mapping problem manageable, we made the following simplifying assumptions:

- There are a bounded number of edges to each node.
- Connection patterns are likely to exhibit some characteristic of neighborliness (nodes are not be likely to be connected arbitrarily to other nodes).
- The amount of computational work on each node is roughly equal.
- The amount of communication on each edge is roughly equal.

We believe that these assumptions are satisfied by many problems, although there are certainly some problems for which the equal work assumption will not be satisfied.

4.1. Algorithm overview

At a high level, the algorithm is simple. Assume, for now, that the graph is already distributed in some manner among the processors. First we identify "pieces" of the graph that when moved from one processor to another result in more "compact" groups of nodes on the processors, and we move these pieces to the preferred processors. We define a piece to be any connected group of nodes, and by compact we mean few off-processor edges. Second, we move additional nodes along existing edges from one processor to another to balance the numbers of nodes. We call these node movements "flows". To allow for the fact that the algorithm is heuristic, we iterate these two steps until we achieve an acceptable balance.

Conceptually, we have tried to address the first two main goals, minimizing inter-processor connections and balancing the number of nodes, in these two separate steps. Since the goals may conflict somewhat, one difficulty is to insure that each step does not undo the benefit of the other.

As the first step indicates, our basic approach to the problem of minimizing inter-processor connections is to focus on "pieces" instead of nodes. We are interested in identifying pieces that have a high and uniform connectivity. We use the term "connectivity" to denote the number of edges between two pieces. By uniform we mean that if we subdivide the piece, the two sub-pieces do not have a connectivity that is radically different from the original piece. We have found that if we only examine individual nodes instead of aggregates, the connectivity information is not very useful.

There are two particularly interesting kinds of pieces: (1) The situation in which a processor has two or more pieces with no connectivity between them. We call these disconnected pieces. (2) The situation in which a piece has greater connectivity between itself and some other processor, than it does between itself and all other nodes on its own processor. We call these outlying pieces.

If we identify all the disconnected and outlying pieces and move them to processors to which they have maximum connectivity, then we have a fully connected structure of nodes on each processor that is more compact than the structure we started with. This is our heuristic approach to minimizing the number of inter-processor edges and constitutes the first major step of the algorithm. The methods for identifying disconnected and outlying pieces are described below.

The remaining step is to balance the number of nodes on each processor. We address this problem by first computing "flows" for each processor pair. A flow is the number of nodes that will be moved from one processor to another to achieve overall balance. Then we attempt to select nodes to satisfy these flows in a manner that avoids creating new disconnected or outlying pieces. We do this by identifying all the nodes on the source processor that are connected to the destination processor and using these as the basis to construct a compact group of nodes that is of the correct size.

Since the algorithm is heuristic, we have to take additional steps to correct possible inadequate results. We do this in two ways. First, the flow step necessarily assumes that there are no disconnected pieces on any processor. It is possible, however, for the piece step to fail to resolve all the disconnected pieces. In this case, we invoke a more complex routine after the piece step that guarantees that there are no disconnected pieces remaining on any processor. Second, we iterate over the two steps a number of times to allow for the possibility that one pass through the algorithm will not adequately distribute the graph.

4.2. Algorithm description

The overall algorithm that implements this approach is as follows. Each processor executes the following steps iteratively until the machine is adequately distributed.

A. Relocate outlying and disconnected pieces:

   1. Identify outlying and disconnected pieces
   2. Chose a destination processor for each piece.
   3. Send each piece to its destination.

B. Balance the number of nodes on each processor:

   1. Exchange with all processors the connectivity between this processor and every other processor.
   2. Compute the flow for each processor pair in a nearly optimal manner for that pair.
   3. Select the nodes to satisfy the flows from this processor.
   4. Send those nodes to their destinations.

All the processors synchronize at the end of step A and at the end of step B.

The three most complex parts of this algorithm are identifying the outlying pieces, computing the flows, and selecting the nodes to satisfy the flows. We now discuss each of these.

The part of the algorithm that identifies outlying pieces works by selecting an "outlying" starting node, then clustering or growing a piece around that node while tracking the ratio of the on-processor connectivity to off-processor connectivity. The on-processor connectivity is the connectivity of this piece to the remaining nodes on this processor. The off-processor connectivity is the connectivity of this piece to that other processor with which it has the highest connectivity. The algorithm selects the starting node by picking the node farthest from the "center" of the piece being examined using an efficient algorithm discussed below. If the connectivity of the initial cluster meets the definition of an outlying piece, additional nodes are added to the cluster as long as the relative connectivity remains the same or increases. That cluster becomes an outlying piece. We increase the size of the cluster by more than one node at a time, because the relative connectivity can decrease if we just add one node even though the relative connectivity averaged over several new nodes is still increasing.

The part of the algorithm that computes the flows attempts to make each processor end up having approximately the same number of nodes. To do this, it selects one processor at a time beginning with the processor with the fewest nodes. For each of the processors that needs nodes, it selects a "best" route to satisfy that need by moving nodes from the other processors along existing edges. (Note that at this stage the algorithm is only determining how many nodes should move between given pairs of processors, but not which nodes.) We use a simple heuristic for "best" that attempts to combine (1) using the shortest route, (2) obtaining the needed number of nodes from as few sources as possible, and (3) minimizing the number of intersecting routes. The last criterion is included because, all other things being equal, the algorithm that selects the particular nodes to satisfy a flow is more likely to work properly on a processor with fewer routes passing through it than one with many routes.

When the algorithm selects the nodes to satisfy the flows, it must be concerned with inflows (nodes coming from other processors) and outflows. The algorithm does the selection by creating a compact group or cluster of nodes with good connectivity to its destination or source processor for each flow. It does this in three steps. First, the algorithm selects a cluster of nodes for each flow using only nodes that have connectivity to the processor associated with the flow. Second, if there is more than one inflow, it connects all the inflow clusters into one cluster. Finally, it adds nodes to each cluster in a round-robin fashion until each cluster has reached its appropriate size. It selects the nodes to add to a particular cluster by looking at nodes that are immediate neighbors of the cluster. This strategy leaves us with one cluster to which all inflows will be attached and one cluster for each outflow, minimizing the possibility of creating new disconnected or outlying pieces.

There is one underlying routine that is used by the algorithm in a number of places and that is interesting. This is the routine that determines how far a node is from the center of the group of nodes on a processor. If we were to compute, for each node, the sum of the distances from that node to all other nodes, where distance is the number of edges in the shortest path, the greatest sum of distances would give us the node farthest from the "center". To avoid having an $O(N^2)$ algorithm, we actually compute the sum of the distances from each node to a randomly selected, fixed size set of nodes. The resulting algorithm has time complexity $O(N)$ as it is a fixed number of single-source shortest path computations where E is a fixed multiple of N and the weight of each edge is the same.

5. Preliminary test results

We have tested our algorithm in two situations so far. First we applied it to static, regular, two dimensional rectangular meshes in which each node is connected to its nearest neighbors. Second, we applied it a dynamic sequence of data structures generated by moving a simulated shock wave across a two dimensional space. The latter set of tests is illustrated in Figures 1 - 6.

The two dimensional rectangular mesh experiments were conducted for eight processors, and mesh sizes ranging from 30 x 30 to 70 x 70. We found that the algorithm does a nearly optimal allocation of nodes in one iteration. That is, the number of nodes on the eight processors varies by no more than 1% from the mean number of nodes, and the ratio of off-processor connectivity to on-processor connectivity is typically about one and one-half times optimum and at worst two to three times optimum with the smaller meshes turning in the worst performance. What is mildly surprising is that the shapes that are generated are far less regular than would likely be generated by a human. (The same phenomenon is apparent in Figures 1 - 6.) This irregularity, while perhaps unappealing visually, does not significantly decrease the quality of the solution as measured by the load balance and off-processor connectivity. It appears to be an inherent characteristic of automatically generated partitions. Some simple timings of these tests suggest that the algorithm has time complexity $\mathbf{O}(N)$ in this case.

In the shock wave simulation, we used a uniform coarse grid as the basic background (40 by 40 points). In the neighborhood of the shock wave, we cut the inter-point spacing in half. We cut the inter-point spacing in half again right on top of the shock wave. The shock wave itself had the shape of a cubic function. At each step in the shock wave simulation, we moved the location of the shock wave one coarse grid point to the right horizontally, initially leaving points mapped to the same processor that had responsibility for those coordinates in the prior step. Then we let the algorithm rebalance the machine. We repeated this movement process thirty times, so that the shock wave moved from the left side to the right side of the underlying grid. The simulation was run for eight processors.

We used three different scenarios for determining the initial mapping of the grid to the eight processors, and then ran the complete simulation starting from each of these configurations. In the first case, the coarse grid was initially block-mapped by partitioning it into four equal parts horizontally and two equal parts vertically. Since the shock wave resides initially in the leftmost vertical slice, two processors had about twice as many grid points as the other six. We then let the algorithm rebalance the graph; the result is shown in Figure 1. The result of this simulation after the shock waved was moved and the graph rebalanced 30 times is shown in Figure 2. In the second case, the coarse grid was initially block-mapped by partitioning it into two equal parts horizontally and four equal parts vertically. This is a slightly more favorable starting configuration since the shock wave resides initially on four of the eight processors. In the third case, we used no initial partition, but simply let the algorithm determine an initial partition of the first graph (the coarse grid with the shock all the way at the left side), and proceed from there. The initial partition that was determined is shown in Figure 5, and the partition after 30 shock wave moves and rebalancing steps is given in Figure 6.

Table 1 summarizes some results from the experiment from the first scenario, for which

Table 1:

Iteration	Node Imbalance	Off-Processor Edges
0	2.1%	10.7%
5	1.4%	9.8%
10	4.8%	9.5%
15	2.1%	14.2%
20	2.8%	11.1%
25	3.5%	11.8%
30	2.1%	15.5%

we obtained the worst results (although the difference from the results of the other two scenarios is not very large). It shows, after every 5 moves of the shock wave, the maximum percentage difference between the number of nodes on any processor and the average number of nodes in the graph, and the maximum percentage of edges on any processor that cross processor boundaries.

The node imbalance results appear reasonable. The algorithm sets an approximate tolerance of 5% for node imbalance, and the partitions almost always satisfy this tolerance. The algorithm could enforce a tighter tolerance if that were desirable, which it might be if the computation to communication ratio was very high. We have not yet experimented with what effect this would have on the cost of the algorithm or the percentage of off-processor edges.

The worst case percentage of edges that cross inter-processor boundaries after any rebalancing step in scenario one was 15.5%, 13.3% in scenario two, and 12.5% in scenario three. In comparison, the percentage of edges that cross inter-processor boundaries for the worst processor in an optimum mapping is on the order of 4.5%. From this perspective the results of the three tests appear reasonable to us, although it might well be possible to do better as we discuss in section 6.

6. Summary and discussion

We initially set four goals for our algorithm. The first, to balance the number of nodes on each processor within a small tolerance, has been achieved in our tests so far and it seems clear that our algorithm will achieve it in general. The second, to minimize the number of edges crossing inter-processor boundaries, so far appears to be met reasonably well but leaves considerable room for improvement; this number is near optimum for simple data structures and between two and three times optimum for more complex data structures. The third, to keep the amount of computation tractable, also appears to be met in our tests so far. In simple cases, the time complexity is $O(N)$ where N is the number of nodes on one processor. In more complex cases, it will become $O(NP)$, where P is the number of processors. Finally, the fourth goal, to use little or no programmer supplied information, has been achieved. No such information is used in our method although it can be used in conjunction with an initial programmer-supplied mapping if desired. Clearly, more testing is required to determine the effectiveness and speed of the algorithm more conclusively.

Our experience has also pointed out several limitations to the current algorithm. Some we believe can be addressed by changes in the algorithm, while others are more fundamentally related to the heuristic nature of the algorithm.

First, the piece part of the algorithm (step A) appears to have significant limitations as it stands. An examination of the worst case off-processor to on-processor connectivity ratios makes it clear that in almost every case a better job of identifying pieces would have reduced these ratios. One reason for this shortcoming is that because the algorithm only adds groups of nodes of some fixed size to the candidate pieces, it fails to find all the pieces it could, in particular, small pieces. We believe this could be fixed by using a more sophisticated version of this part of the algorithm that considers varying size groups. In the examples we have looked at, this would reduce the percentage of edges that cross inter-processor boundaries by one to two percentage points. This gain may justify the additional cost.

The second reason for the limitations to the effectiveness of the piece step is more complex. In examining various test cases, it is apparent that the algorithm sometimes ends with a set of nodes on a single processor that does not appear compact, although almost all nodes belong to one of several (generally two) compact pieces. An example is a two dimensional structure with the shape of a dumbbell, i.e. two rectangular pieces connected by a thin strip. We could sometimes reduce the percentage of edges that cross inter-processor boundaries if we could break such groups up into several pieces and redistribute them, although the gains might not be large. The present algorithm does not identify these situations effectively, since it is basically oriented to finding protruding pieces from one central connected piece, but a more general version could identify such situations. More research is necessary to determine whether the gains from this addition to the algorithm would justify the added costs.

A third, more fundamental limitation is that there is only a limited amount of information available on a processor about the state of affairs on other processors. For example, in Figure 1, the two processors in the lower left hand corner would have lower off-processor connectivity if we were to combine them and split them in half horizontally. While we could probably devise an algorithm that could identify this situation for any adjacent pair of processors, solving the general case would negate much of the benefit of a distributed algorithm.

In conclusion, we believe that these initial results are promising. We have provided a start towards solving a critical problem in efficiently and automatically solving unstructured problems on parallel machines, namely automatically mapping and load balancing the unstructured or dynamic data effectively. While our research is in the context of an explicitly parallel language, DYNO, we expect that related techniques will be needed by any system that automatically solves unstructured problems on parallel machines starting from a high level programming language description of the algorithm. In the future, sophisticated compiler/run-time systems, by combining the type of mapping and load balancing algorithm presented here with automatic compiler and run-time analysis that provides a detailed picture of the distributed data structure, may be able to handle mapping and load balancing in a manner that is totally transparent to the programmer.

References

[1] S. Bokhari. "On the Mapping Problem." *IEEE Transactions on Computers*. 30, 3 (March 1981)207 - 14.

[2] B. Chapman, P. Mehrotra, and H. Zima. "Vienna FORTRAN — A FORTRAN Language Extension for Distributed Memory Multiprocessors." *ICASE Technical Report No*. 91-72. To appear in *Languages, Compilers and Runtime Environments for Distributed Memory Machines*, Elsevier Press.

[3] N. Chrisochoides, C. Houstis, E. Houstis, S. Kortesis, and H. Rice. "Automatic Load Balanced Partitioning Strategies for PDE Computations." *1989 International Conference on Supercomputing* (1989) 99 - 107.

[4] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. "Distributed Memory Compiler Methods for Irregular Problems — Data Copy Reuse and Runtime Partitioning." *ICASE Technical Report No*. 91-73. To appear in *Languages, Compilers and Runtime Environments for Distributed Memory Machines*, Elsevier Press.

[5] K. Dragon and J. Gustafson. "A Low-Cost Hypercube Load-Balance Algorithm." *The Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*. (March 1989) 583 - 89.

[6] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. "Fortran D Language Specification." Center for Research on Parallel Computation Technical Report CRPC-TR90079 (1990).

[7] P. Mehrotra and J. Van Rosendale. "Programming Distributed Memory Architectures using Kali." *ICASE Technical Report No. 90-69*. (1990)

[8] Y. Moon and J. Sklansky. "A Class of Mapping Algorithms for Hypercube Computers." *Proceedings of the Fifth Distributed Memory Computing Conference* (April 1990) 903 - 08.

[9] T. Ngai, S. Lundstrom, and M. Flynn. “Automated Run-Time Scheduling of Unstructured Scientific Computation on Scalable Multiprocessors.” To appear in *Unstructured Scientific Computation on Scalable Multiprocessors*, MIT Press.

[10] M. Rosing, R. Schnabel and R. Weaver. “The DINO Parallel Programming Language. *Journal of Parallel and Distributed Computing* 13, 9 (September 1991) 30 - 42.

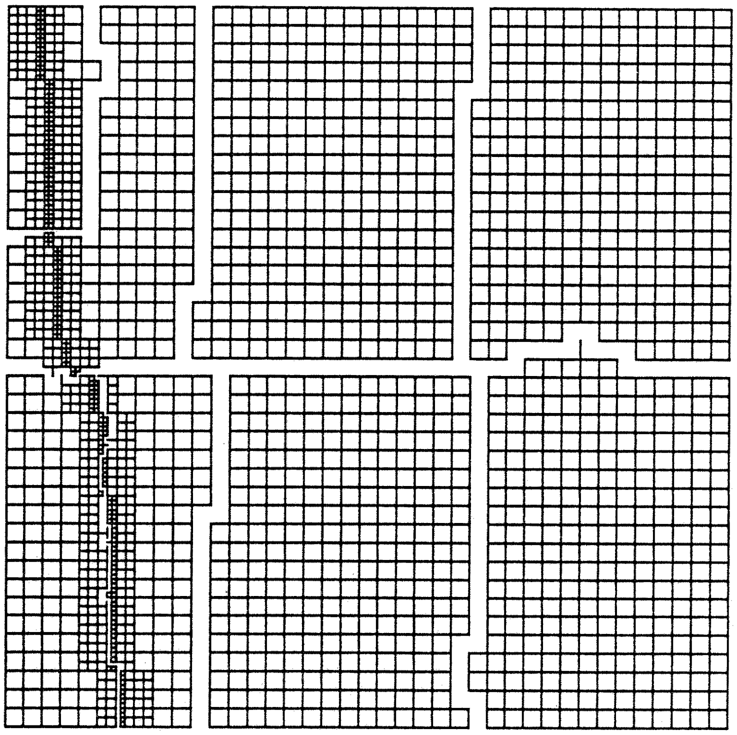
Figure 1 — Scenario One Initial Mapping

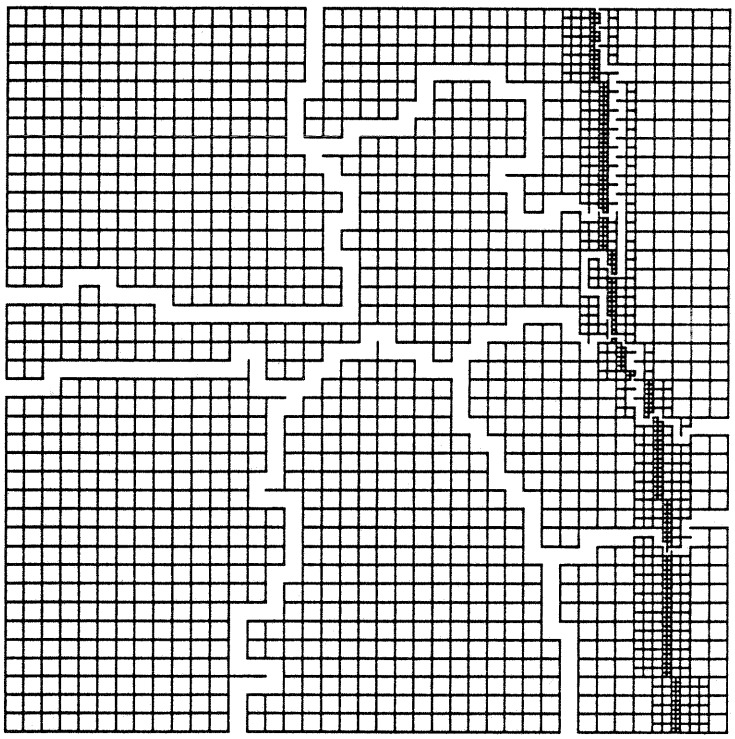
Figure 2 — Scenario One Final Mapping

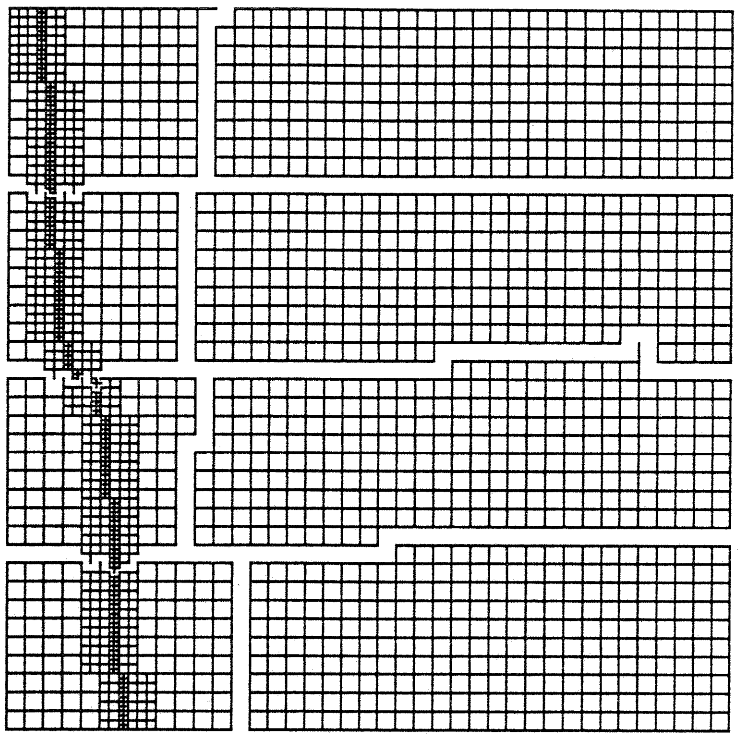

Figure 3 — Scenario Two Initial Mapping

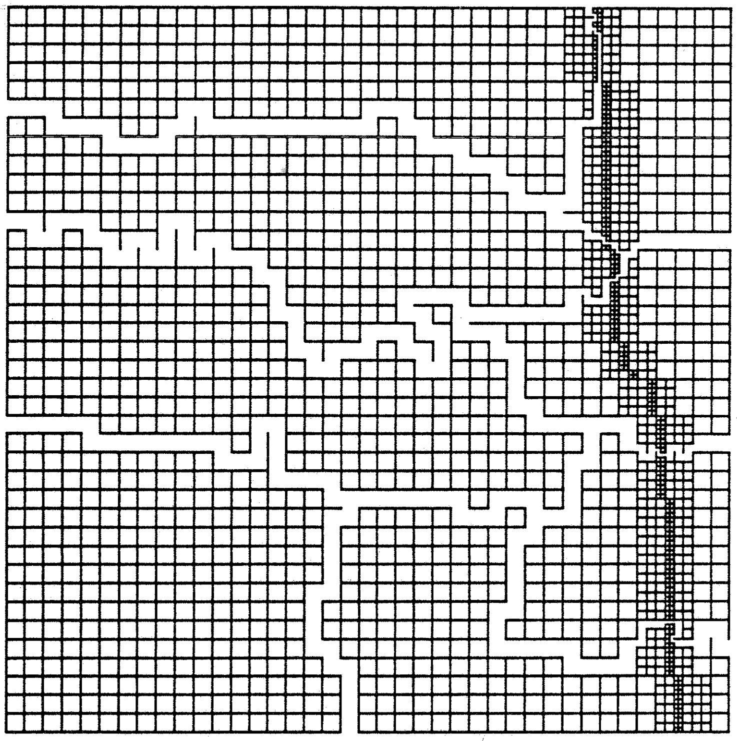

Figure4 — Scenario Two Final Mapping

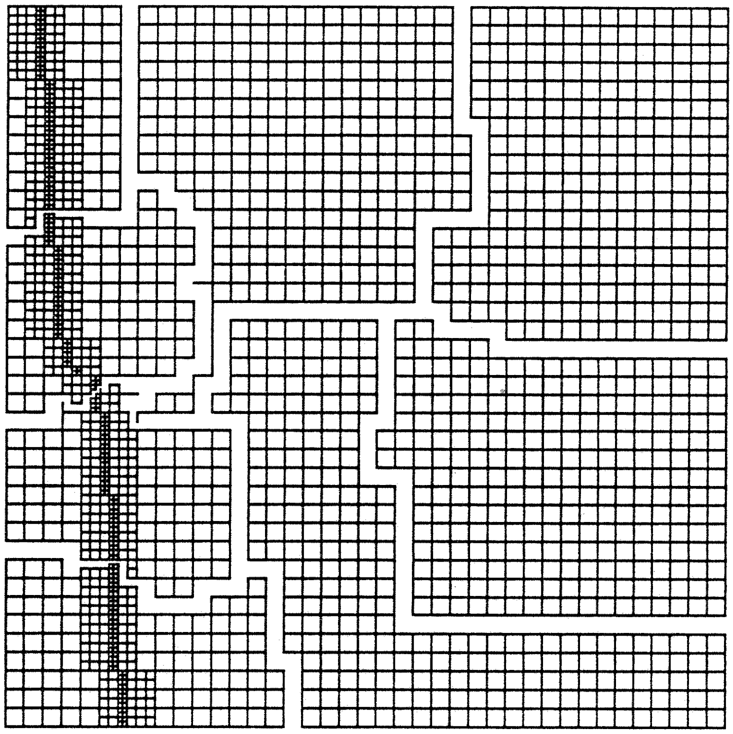
Figure 5 — Scenario Three Initial Mapping

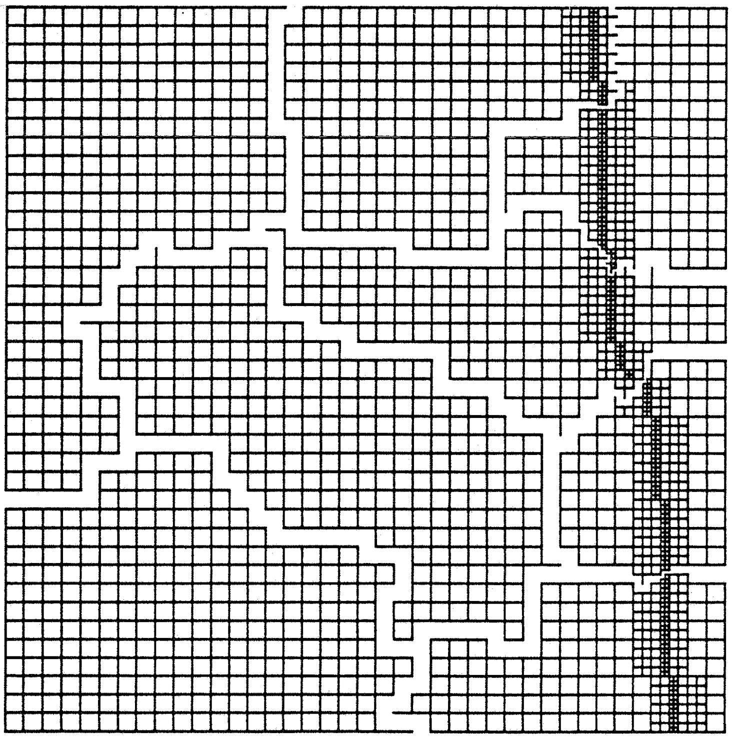
Figure 6 — Scenario Three Final Mapping