Rotation Invariant Partitioning for Concurrent Scientific Visualization*

Michael E. Palmer[a] and Stephen Taylor

Mail Code 256-80, California Institute of Technology, Pasadena, CA 91125 USA

[a]mep@scp.caltech.edu

The contributions of this work apply to volume visualization of unstructured data on concurrent multicomputers. We describe an image partitioning method designed to accomodate *dynamic load balancing* of the rendering computation among parallel processors. We present a *center of mass vector measure* to estimate workload distribution; this method is superior to traditional scalar measures in accuracy and in adeptness at tracking moving data. The primary drawback of image partitions in general is that some changes of viewpoint may require that all data elements move to a different processor. This led us to invent a new type of partition, which we call the *rotation-invariant partition*, which are a hybrid of image and object partitions. They are designed to eliminate communication incurred by rotation of the observer's head and reduce that incurred by translation, by scrolling across the screen with the bulk of the data.

1. RELATED WORK

Designing algorithms for MIMD, message-passing concurrent computers presents several challenges: First, the workload of computation must be evenly balanced among the processors. Second, memory use must be evenly balanced among processors. We also want to avoid the replication of data on more than one processor. Third, interprocessor communication must be minimized since on some concurrent architectures, it is at least two orders of magnitude as expensive as a local memory fetch.

One approach taken by concurrent rendering algorithms has been to assign to each processor a fixed set of the elements or objects in the data set; such a partitioning of the problem is called an *object partition* [2,3,5,1]. The natural coordinate system of the data is called object space. When an object partition is used, rays cast through the data will, in general, intersect elements held by many different processors; they will therefore need to be communicated from processor to processor as they traverse the volume of data. Another partitioning method is is called the *image partition*, one in which the partition is stationary relative to the screen image [8,7]. With image partitions, communication needs arise in delivering to each processor all the data elements which fall within its subvolume of image space, which change with each shift in viewpoint.

*This research is sponsored by the Department of Defense, AASERT award number N0014-93-1-0843, under a parent grant from the Advanced Research Projects Agency, ARPA Order 8176, monitored by the Office of Naval Research under contract N00014-91-J-1986.

Methods of spatial subdivision can be further divided into those which employ a uniform, static subdivision [5,1], and those that divide space in an adaptive manner, either by iterative [6,2,8], or recursive [10] subdivision methods. Adaptive subdivisions may be adapted to the data just once at initialization [10], or dynamically in response to changing conditions of workload during execution [6,8]. One-time load balancing is cheaper, but when load varies dramatically during execution, dynamic load balancing will more than pay for itself in better load balance.

2. A PARTITIONING METHOD

2.1. A Recursive Subdivision of Image Space

We present a partitioning method, based on an image partition, which is designed to accommodate dynamic load balancing by allowing flexible reassignment of processor power to emerging concentrations of work load: the screen is cut recursively in two, by a series of $\log N$ cuts, where N is the number of processors. The first, and subsequent odd divisions, are parallel to the Y axis; the second, and subsequent even divisions, are parallel to the X axis. The left of Figure 1 illustrates assignments of screen areas to processors, while the right shows the corresponding assignments of volumes of image space to processors. Here there are $N = 8$ processors, numbered from zero to seven. A processor owns the entire volume of image space behind its portion of screen area. In the figure, the entire screen is recursively divided by $\log N = 3$ sets of lines: first by the line A1, then by the lines B1 and B2, and finally by the lines C1, C2, C3 and C4.

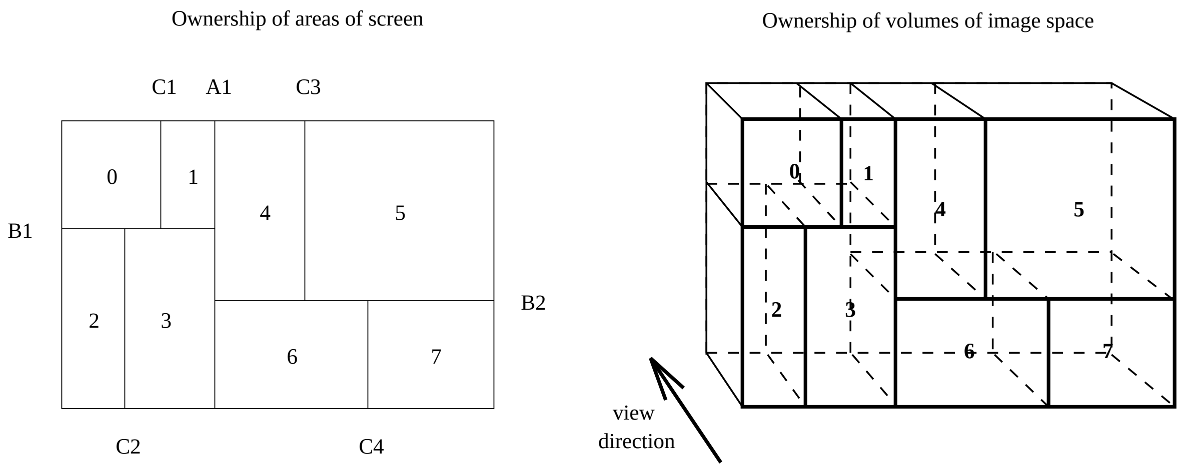

Figure 1. Subdivision of Screen Area and Image Space

2.2. Overview of Algorithm

The basic cycle repeated to generate images from a sequence of viewpoints is the following. We first broadcast the current view matrix to all processors. The processors calculate the new positions in image space of the elements they hold. Let the *owner* of an element be the processor within whose subvolume the element lies. With each shift in viewpoint, some data elements may fall into a new processors' subvolume of image space. These elements are transported to their new owners. After this exchange is complete, each processor sorts its data elements and composes them together to generate the volume rendered image for its portion of the screen. Measurements of the distribution of the work load among the processors are used to load balance for the next frame. Load balancing is effected by appropriate movement of the lines dividing the subvolumes of image space.

3. IMPROVED DYNAMIC LOAD BALANCING

We present a novel means to measure the distribution of work in image space using *center of mass vectors*. As each processor composes its elements onto the display screen, it finds their collective center of mass. The reader will recall that the center of mass of a collection of masses m_i at positions x_i is the vector: $\vec{\mathrm{CM}} = \frac{\sum_i \vec{x_i} m_i}{\sum_i m_i}$

We use the center of each data element as the position $\vec{x_i}$ and gives each element a mass of one (1).

Using the center of mass of the element distribution, instead of the common alternative of simply *counting the elements* in each area, allows a qualitatively superior load balance: first, because the center of mass more accurately describes the density distribution than a simple count; furthermore, because the center of mass is a vector, it can be matrix-multiplied by the difference between the last and current view matrices to *anticipate to the next frame* an appropriate balance. [9].

Load balancing is achieved by the movement of the lines dividing areas of the display screen, shown in Figure 1, appropriately for new distributions of workload. Each processor knows the numerator and denominator of the above equation. The line between them is placed on the center of mass of their combined areas — this by definition places half of the work on each side of the line. Likewise, all the lines further up the tree can be similarly balanced, until line A1 on the left of Figure 1 is placed on the center of mass of the entire screen. Details were previously discussed in [9].

4. ROTATION-INVARIANT PARTITIONING

Image partitioning methods suffer from a significant problem: even under the constraints of viewpoint coherence, there are some changes of viewpoint that require the movement of *all* elements of the data. This incurs an enormous communications expense.

This problem led us to invent a new class of partitioning methods, a hybrid of image and object partitions, which we call *rotation-invariant* partitions. The essence of the approach is to map the screen onto a two-dimensional torus, and allow the processors to scroll across the screen, following movements of the data. Let *processor space* be the two-dimensional space fixed to the surface of this torus, but allowed to translate arbitrarily on it. The lines dividing the screen are fixed to processor space. If the origin of processor space is translated by a certain vector, so are all the lines; some of them may be wrapped around the edges of the screen. The origin of processor space is initially placed in the center of the screen.

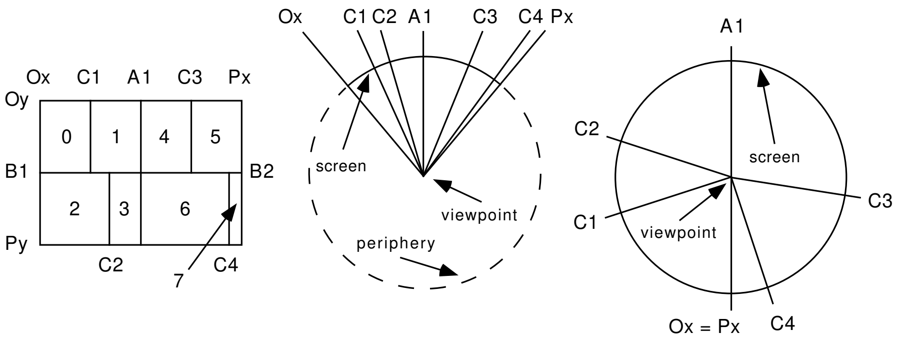

Figure 2. Three Views of a Set of Partitioning Lines

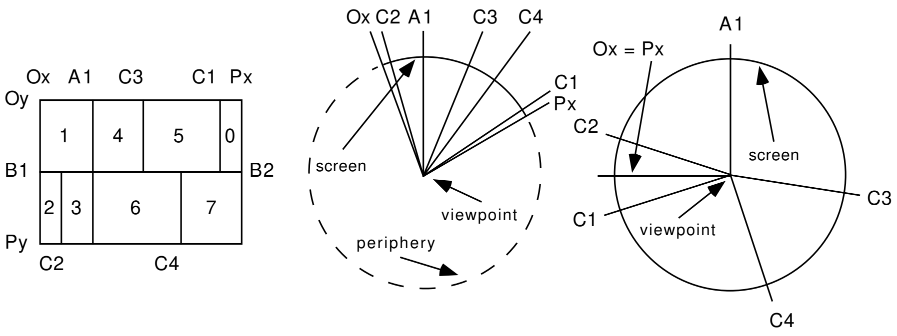

Figure 3. Three Views of the Lines after a Head Rotation

Such a partition is illustrated in Figure 2. The figure shows three representations of the same distribution of partitioning lines. On the left of the figure is a view of the display screen from the point of view of the observer. The screen is recursively divided by three sets of lines labelled A1, B1, B2, and C1, C2, C3, C4. Ox and Px are the left and right boundaries of the screen in the X direction; Oy and Py are the boundaries in the Y direction. Each of the vertical lines, i.e. A1, C1, C2, C3, C4, Ox, Px, corresponds to a Y-Z plane through image space. In the center of the figure is a view of image space from above. Perspective is used so the display screen appears as a solid arc. The viewpoint of the observer is in the center of the circle. Each of the Y-Z planes previously mentioned is shown here as a line radiating from the viewpoint. The dashed arc passes through the volume of image space which is currently in the periphery of the observer's field of view. On the right side of of the figure, Ox and Px have been stretched around the circle until they become a common point, and the periphery has been spliced out; the screen has been mapped onto a torus in the X direction. In our rotation-invariant partitions, the screen is also mapped onto a torus in the Y direction, but this is not visible in the figure.

Figure 3 shows what happens after a head rotation of twenty degrees to the right. Note in the center of the figure that Ox and Px have moved twenty degrees clockwise around the circle; Lines A1, C2, C3, and C4 have remained stationary, but line C1 has scrolled off the left side of the screen and wrapped around to the other side. The left and right sides of the figure show alternate views of the same set of line positions.

If the screen fills a V degree field of view (in Figures 2 and 3, the angle between Ox and Px is eighty degrees) then during a head rotation of 360 degrees, each processor will scroll across the screen from left to right $\frac{360}{V}$ times.

To make the load balancing lines follow the bulk movement of the data at each frame, the origin of processor space is simply translated by some vector T in between frames. The T we use approximates how much the center of mass of the currently visible data elements has moved in between frames. We find the center of mass of all data elements for the last frame, matrix-multiply it by the difference between the current and last view matrices, and discard the Z coordinate. We then translate the origin of processor space by this vector.

These partitions are designed to eliminate communications costs due to head rotations about the X and Y axes. Communication incurred by panning (X or Y pure translation) is not completely eliminated by this technique, but is reduced, especially for elements that are far from the observer.

4.1. Rotation-Invariant Partition Example

When the the observer's viewpoint is outside the data volume, as in the three frames to the left side of Figure 4, the data is concentrated in one area of the screen, and the dynamic load balancing algorithm can move the lines dividing the screen to follow the movement of the data, with the result that fewer data elements must cross a line. However, when the user is in the interior of the data, surrounded by it, then data scrolling off the screen is replaced by data scrolling on to the other side. The lines cannot move to follow the data. Much of the data must cross lines between processors at each frame — each time requiring expensive communication between processors.

The three frames to the right of Figure 4 show the rotation-invariant partitioning method operating on unstructured data from a Computational Fluid Dynamics simulation of the Titan IV rocket, generated by Johnson Wang, of the Aerospace Corporation, and Stephen Taylor, of Caltech, using the Intel Delta Machine. The figures show only a slab (several slices) of the data for clarity; data elements are also "outlined" to illustrate the grid structure. In frame 15, the two yellow bow shocks from the nosecone and booster of the rocket are the prominent features. The observer pans, zooms into the data and then makes a long pan to the right. Sixteen processors were used, and each is outlined in a different shade of grey according to processor number; the brightest is processor fifteen. Note that the processors scroll across the screen in response to data movement, rather than remaining fixed to the screen edges. For example, processor fifteen, with the brightest outline, does not remain in the lower right corner of the screen. Two intersecting red lines show the current center of mass of the data.

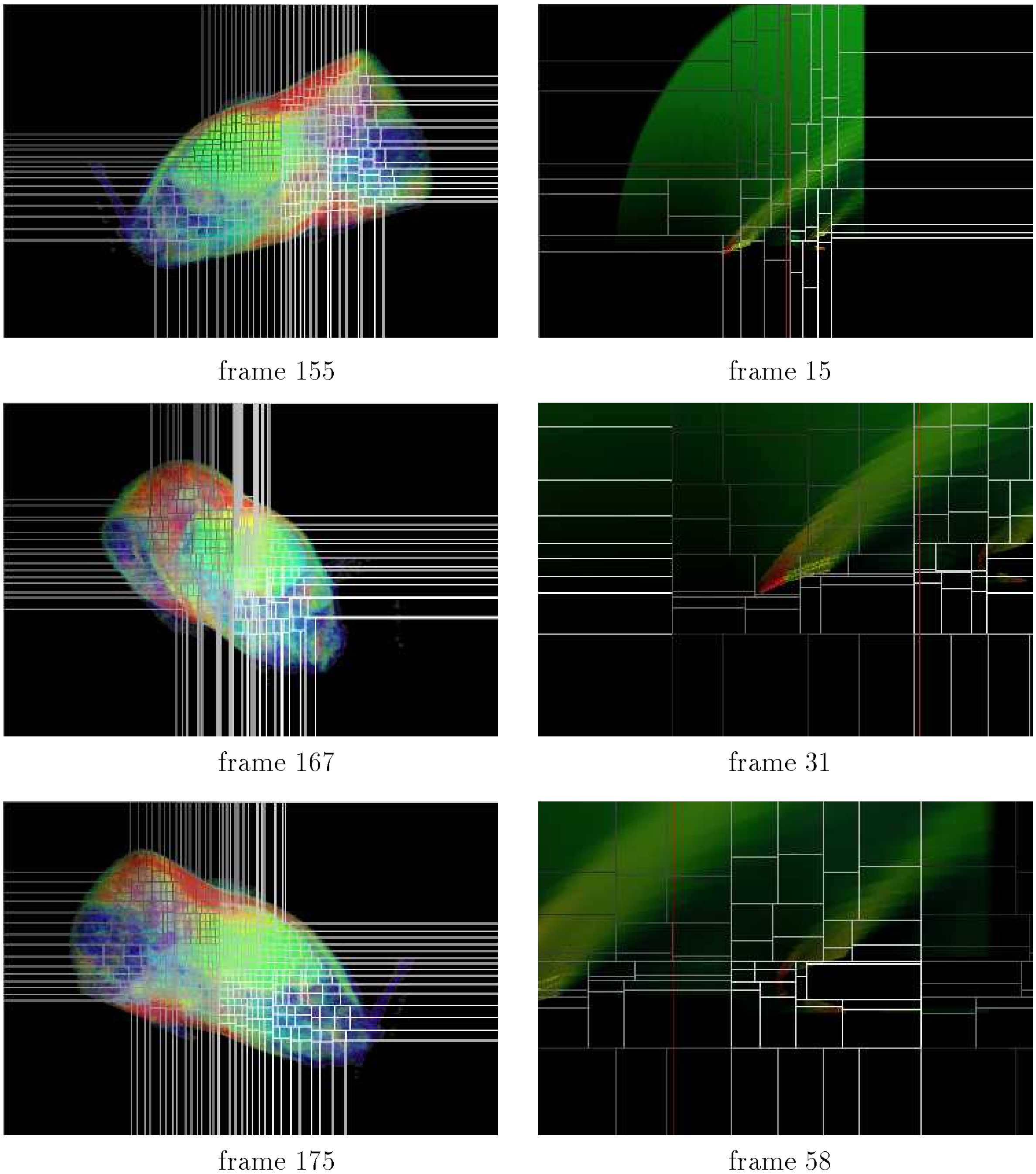

Figure 4. Following Moving Data with 512 Processors, Mouse Head MRI (left) and Rotation-Invariant Partition, Titan IV CFD Data (right)

4.2. Rotation-Invariant Partition Experimental Results

Our quantitative measurements are from the same sequence of head movements shown to the right of Figure 4, but the entire dataset was used: 1.4 million unstructured hexahedral data elements. On 512 processors of the Intel Delta Machine, frames were obtained at a rate of 32 seconds per frame. When image generation, the main computational expense, was turned off — leaving only communication — the frame rate was 25 percent faster.

We ran experiments with 128, 256, and 512 processors, getting qualitatively similar results for each number of processors; we show only the 512-processor results here for brevity. The experiments were meant to measure the performance characteristics of the rotation-invariant partition for a typical sequence of head motions. At frame 0, a sampled one percent of the dataset is loaded and the load balancing lines are allowed to converge until frame 15. At this point the entire dataset is loaded and the detailed sequence of events in the movie were: Frames 15-40: Various diagonal panning motions by observer; Frames 22-32: Observer zooms in close to the data (Note overlap with above pans); Frames 40-65: Long straight pan to the right.

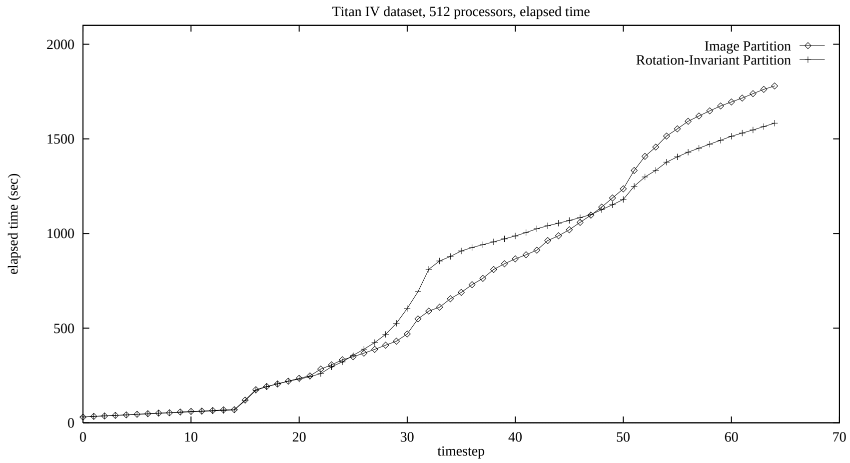

Figure 5. Elapsed Time versus Timestep, 512 Processors

Figure 5 compares the two algorithms, showing the cumulative elapsed time at each time step. Here the slope of the line for the rotation-invariant partition is steeper during the zoom of frames 22-32 — each frame takes longer — but it catches up during the pans of frames 33-40 and 40-65 to end with a smaller total elapsed time.

We attribute the fact the the regular image partition is faster during frames 22-32 to the combination of zooming with panning. During a zoom, since many data elements are leaving the screen, the center of mass of the remaining elements will seem to gravitate towards any high concentration of elements; this will not in general be the same direction and magnitude as a pan that is going on at the same time; therefore in this case our tracking method will not in this case follow a pan correctly.

Note that this is not a failure of the rotation-invariant partition method *per se* but rather a failure of the particular way we chose to translate the origin of processor space

— i.e. translating it by vector T where T is the amount that the center of mass of the visible data elements has moved between frames. Choosing not to translate the origin of processor space at all reduces the rotation-invariant partition to the original image partition (with a different fixed origin). If we had chosen a T of zero when the observer was zooming, we could have done just as well as the image partition during zoom.

In summary, our method of following the data works well for pans, but not for zooms. An area for further exploration is defining a translation vector T that works well for a variety of head motions.

5. CONCLUSIONS

In addition to techniques for accurate *dynamic load balancing* of concurrent volume rendering, we propose a new class of methods to partition the rendering problem among concurrent processors. These *rotation-invariant partitioning methods* are a hybrid of image and object partitioning methods. These partitions are designed to eliminate communication incurred by viewpoint rotation and reduce that incurred by viewpoint translation. They allow processors to scroll across the screen, following the bulk movement of the data, rather than requiring large movements of data between processors. Superior methods for tracking the bulk movement of data most efficiently for a variety of head motions will be the focus of further experiments.

REFERENCES

1. J.G. Cleary, B.M. Wyvill. "Analysis of an algorithm for fast ray tracing using uniform space subdivision." *The Visual Computer*, 4:65-83, Springer-Verlag 1988.
2. M. Dippé, J. Swensen. "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis." *ACM Computer Graphics* 18(**3**):149-158, 1984.
3. J. Goldsmith, J. Salmon. "Automatic creation of object hierarchies for ray tracing." *IEEE Computer Graphics Appl.*, 7(**5**):14-20. May, 1987.
4. H. Kobayashi, T. Nakmura, Y. Shigei. "Parallel processing of an object space for image synthesis using ray tracing." *The Visual Computer* (1987), 3:13-22, Springer-Verlag 1987.
5. H. Kobayashi, S. Nishimura, K. Kubota, T. Nakamura, Y. Shigei. "Load balancing strategies for a parallel ray-tracing system based on constant subdivision". *The Visual Computer* (1988), 4:197-209, Springer-Verlag 1988.
6. K. Nemoto, T. Omachie. "An adaptive subdivision by sliding boundary surfaces for fast ray tracing." In: *Proc Graphics Interface '86, Canadian Information Processing Society*, pp. 43-48.
7. U. Neumann. *Volume Reconstruction and Parallel Rendering Algorithms: A Comparative Analysis.* Ph.D. Thesis, Dept. of Computer Science, University of North Carolina, 1993.
8. J. Nieh, M. Levoy. "Volume rendering on scalable shared-memory MIMD architectures." *ACM SIGGRAPH 1992 Workshop on Volume Visualization*, pp. 17-24.
9. M.E. Palmer. *Immersing the Scientist in Data: Interactive Visualization of Unstructured Scientific Data on Concurrent Architectures.* Master's Thesis, Dept. of Com-

puter Science, California Institute of Technology, Pasadena, CA. Caltech-CS-TR-94-06. 1994.

10. T. Priol, K. Bouatouch. "Static load balancing for a parallel ray tracing on a MIMD hypercube." *The Visual Computer* (1989), 5:109-119, Springer-Verlag 1989.