Massively Parallel Model Matching

Geometric Hashing on the Connection Machine

Isidore Rigoutsos and Robert Hummel
New York University

Geometric hashing provides a parallel method for model matching. However, the design and implementation of parallel algorithms for AI applications is fraught with subtlety and challenge.

General-purpose parallel computer architectures are now designed with computationally intensive applications such as computer vision in mind. Nevertheless, using an existing parallel machine to develop a working vision system presents numerous problems. Our implementation of an object-recognition algorithm on a Thinking Machines Corporation CM-2 Connection Machine[1] may typify the choices and difficulties involved.

Developing realistic vision systems that can recognize rigid objects from a database of hundreds of models is a continuing goal of vision researchers. A model-based vision system extracts features such as edges and points from digital imagery and compares them with a database of models to identify objects within a scene. Many model-based vision systems are based on hypothesizing matches between scene features and model features, predicting new matches, and verifying or changing the hypotheses through a search process.[2,3] A new method, called *geometric hashing*,[4] offers a different and more parallelizable paradigm (see the sidebar on the following page).

Geometric hashing uses the collection of models in a preprocessing phase (executed off line) to build a *hash table* data structure. The data structure encodes the model information in a highly redundant, multiple-viewpoint way. During the recognition phase, when presented with a scene and extracted features, the hash table data structure indexes geometric properties of the scene features to candidate models. The geometric hashing scheme still requires a search over features in the scene, but it obviates a search over the models and the model features. Thus, its recognition phase offers computational efficiencies, and geometric hashing is highly amenable to parallel implementation.

In this article, we explore the parallelizability of geometric hashing and present two algorithms. The first one uses (1) parallel hypercube techniques to route information through a series of maps and (2) building-block parallel algorithms. The second algorithm uses the Connection Machine's large memory resources and achieves parallelism through broadcast facilities from the front end.

0018-9162/92/0200-0033$03.00 © 1992 IEEE

We will confine ourselves to the problem of recognizing dot patterns embedded in a scene after they have undergone translation, rotation, and scale changes. Each dot can represent a feature location extracted from an image. In a more general vision system, we would like to recognize patterns of lines, corners, and other features, attached to three-dimensional objects, undergoing rigid 3D transformations and perspectively projected onto an image plane. Conceptually, the geometric hashing algorithms will extend to such cases. But many implementation issues must be addressed, and the transformation classes of the features will be more complicated. In particular, the coordinates of the points in our study are represented by two-dimensional entities. More complex features will involve more dimensions and more complex transformations. In our study, we use patterns of 16 points; models involving more complex features will need to be described by a similarly small number of primitives.

As implemented, the algorithms recognize models consisting of patterns of points embedded in scenes, independent of translation, rotation, and scale changes. Thousands of models may be used, each containing approximately 16 points, with scenes consisting of hun-

Geometric hashing for point matching

Suppose we want to recognize patterns of points that may be translated, but for the moment, we assume no rotation, scaling, or other transformations. The model in Figure A consists of five dots. Suppose that we place dot 1 at the origin of a coordinate system. Then the other dots lie at four different (x, y) locations. Let's record in a quantized hash table, in each of the four bins where this information lands, the fact that model M_1 with basis point 1 yields an entry. Figure A shows this graphically, viewing only entries of the form $(M_1, 1)$. Similarly, the hash table contains four entries of the form $(M_1, 2)$, four entries of the form $(M_1, 3)$, and so on. Each entry is generated by placing the base point at the origin of the hash table and observing where the other points of the model land. The same process is repeated for each model. Of course, hash bins may receive more than one entry. As a result, the final hash table contains a list of entries of the form (model, base point) in each bin.

In the recognition phase, a single point from the scene is chosen as a candidate basis point. The coordinates of all other points are then calculated with this point placed at the origin. Each remaining point is mapped to the hash table, and all entries in the corresponding bin receive a vote. If there are sufficient votes for one or more (model, basepoint) combinations, then a subsequent stage attempts to verify the presence of a model with the designated point located at the chosen basis point. If points are missing from the scene because they are obscured, recognition is still possible, as long as

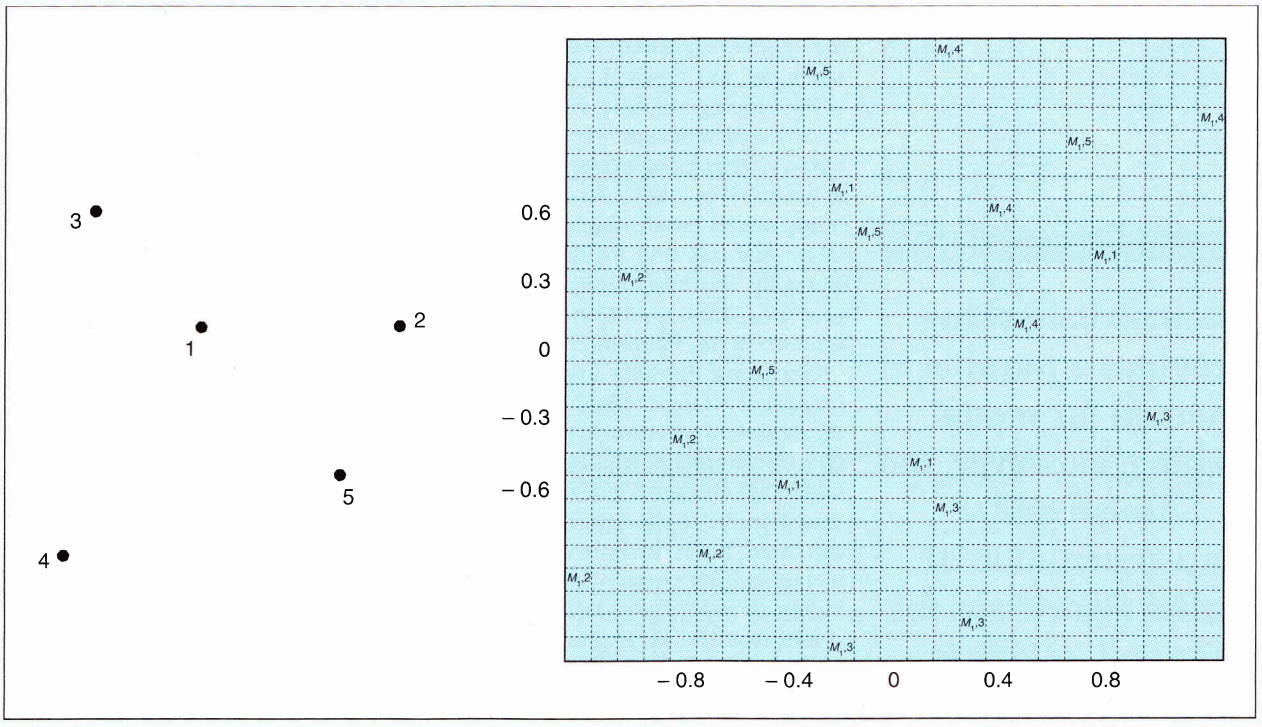


Figure A. Model M_1 consisting of five points and all resulting hash table entries.

dreds of points. Most of the scene points are spurious noise points, and points may be obscured or misplaced. The system searches over pairs of scene points and obtains recognition as soon as both points lie on the embedded object. However, multiple pairs can be probed at the same time, and many heuristics exist for choosing likely basis pairs. With 1,024 models and scenes consisting of 200 points, execution time on a 64K-processor CM-2 would be roughly 70 milliseconds per probe with a single basis set.

Geometric hashing connectionist algorithm

The first parallel algorithm is data parallel over the hash-table bin entries and scene points, but serial over the bases. We assume that our database contains M models; each model m has an associated set, $S_m = \{(x_{m,k}, y_{m,k})\}_{k=1}^{n}$, containing the coordinate pairs of the mth model's n points.

Preprocessing. For the preprocessing phase, where the hash table is created from the model set, the algorithm has time complexity $\Theta(M \log n)$ if Mn^3 processors are used on a concurrent-read exclusive-write SIMD Hypercube. We iterate sequentially over the M models. Initially, each of the low-order n processors contains the (x,y) coordinates of one of the n points in S_m for the mth model. The set $(S_m \times S_m) \times S_m$ is then computed using the triple-product algorithm (see the sidebar, "Building-block algorithms," on the next page). Each of the first n^3 processors of set V_2 now contains a triplet: $[(x_i, y_i), (x_j, y_j), (x_k, y_k)] \in (S_m \times S_m) \times S_m$. The first two points of each triplet define an ordered basis and thus a coordinate system. Accordingly, each processor with a reasonable basis pair and a distinct third point can compute the coordinates of the third point of the triplet relative to that coordinate system. (Note that some processors will be turned off at this point.) These coordinates can then be converted to a hash bin number. This phase is completed once each such processor communicates its information to the appropriate hash bin.

To efficiently communicate model and basis-point information to the hash bins, the algorithm performs two passes through the models. In the first pass, it counts the number of entries that will occur in each hash bin. Each processor with an entry destined for a hash bin sends an additive write with increment 1 to an accumulator in that bin. By performing a parallel prefix sum of the resulting counts, the processor pool can be organized into a one-dimensional array so that each hash bin occupies a contiguous block of (virtual) processors and the block length equals the number of expected entries. Further, a map gives the index of the head processor for each block of processors representing a hash bin.

In the second pass, we again iterate sequentially over the models. This time, for each processor that computes a hash table entry, the information is sent to the appropriate processor in the hash table group. The location of the appropriate processor can be maintained by

there is a sufficient number of points hashing to the correct bins. The list of entries in each bin may be large, but because there are many possible models and basis sets, the likelihood that a single model and single basis set will receive multiple votes is quite small, unless a configuration of transformed points coincides with a model. In general, we do not expect the voting scheme to give only one candidate solution (see Lamdan and Wolfson[5]). The goal of the voting scheme is to reduce significantly the number of candidates for the verification step.

The case where the dot patterns can also undergo rotation and translation is treated analogously. Two points are now needed to define a basis, and point locations are measured relative to a coordinate system defined by the pair of points. When we place the basis midpoint at the center of a coordinate system (see Figure B), the remaining points of model M_1 land in three locations. A quantized hash table will now record, in each of the three bins where the dots land, the fact that (in this case) model M_1 with basis (4,5) yields an entry in this bin. During recognition, two points from the scene are chosen as the basis. The coordinates of the other scene points are calculated relative to the coordinate system defined by the basis pair.

Each of the remaining points is again mapped to the hash table; all entries in the corresponding bin receive a vote; and we continue as before. If the dot pattern is also allowed to undergo scale changes, we use the length of the selected basis as the length of the unit vector of the coordinate system.

For the algorithm to be successful, it is sufficient to select as a basis tuple any pair of points belonging to some model. It is not necessary to hypothesize which model nor which model points are the corresponding points, since all models and basis pairs are redundantly stored within the hash table. Classification or perceptual grouping of features can be used to make the search over scene features more efficient, for example, by making use of only special basis pairs.

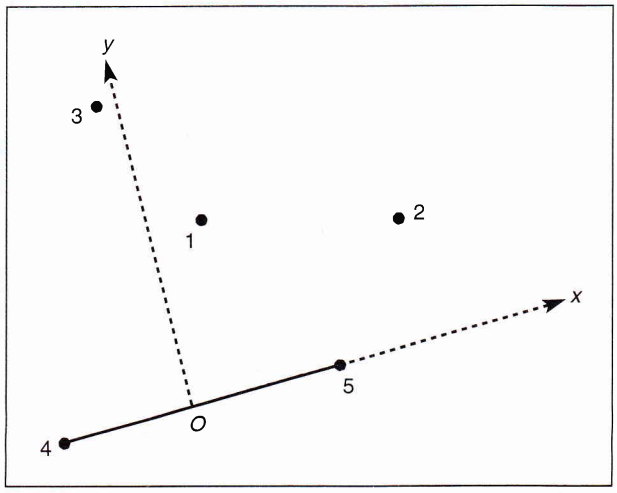


Figure B. Coordinate system for normalizing models using two-point bases.

Building-block algorithms

Certain building-block algorithms are fundamental to the programming of a hypercube-based SIMD architecture. We will need the following components.

Triple product. Given three finite sets $A = \{a_i\}_{i=1}^{L_1}$, $B = \{b_j\}_{j=1}^{L_2}$, and $C = \{c_k\}_{k=1}^{L_3}$, the triple product $A \times B \times C$ is the set of all the ordered triplets $\{(a_i, b_j, c_k)\}_{i=1, j=1, k=1}^{L_1, L_2, L_3}$.

One way to compute the triple product is to perform an outer product twice. An outer product for the Connection Machine is succinctly described in Little, Blelloch, and Cass.[6] An extension of the method leads to a direct triple product computation, which we now describe.

Using standard gray-code embedding algorithms, we configure the hypercube as a three-dimensional array of size $L_1 \times L_2 \times L_3$. (We assume, purely for convenience, that the L_i are powers of 2.) The processors are indexed by their coordinates (i, j, k), and initially data element a_i is contained in processor $(i, 0, 0)$, b_j in processor $(0, j, 0)$, and c_k in processor $(0, 0, k)$.

The algorithm has two phases. During the first phase, the a_i data is spread along a row in the direction of the y axis, the b_j data is spread in the direction of the z axis, and the c_k data is spread in the direction of the x axis (see Figure C). In the second phase, the data on each plane is spread into the entire cube, first spreading the data on the (x, y) plane along the z axis, then the data on the (y, z) plane along the x axis, and finally the data on the (x, z) plane along the y axis. Upon completion, processor (i, j, k) will have received datum a_i from $(i, j, 0)$, datum b_j from processor $(0, j, k)$, and datum c_k from processor $(i, 0, k)$ and thus has the triple product element (a_i, b_j, c_k).

The operation of spreading data along a single axis that occurs during both phases can clearly be performed in $O(L_i)$ time, since nearest neighbors are adjacent in the hypercube, but can in fact be completed in $O(\log L_i)$ time. This is because we may use a recursive doubling scheme to spread the data rapidly along the axis. (Algorithms of this kind are described by Hillis and Steele.[7]) In the parlance of the Connection Machine's Paris language, the operation is a scan_with_copy. Power-of-two communication along each axis is provided by $O(1)$ communication cycles due to the gray-code embedding. Specifically, if $g(i)$, $i = 0, 1, \ldots, n-1$, is a gray code (n a power of 2), then it can be shown that $g(i)$ and $g((i + 2^k) \bmod n)$ differ in at most two bits, and thus can be connected by two communication cycles on a hypercube.This is true for any value of k.

Histogramming. Given a collection of data $\{a_i\}_{i=1}^{N}$, such that each a_i is an element of a finite collection of possible values, say $a_i \in \{1, 2, \ldots, V\}$, the histogram is a count of the number of elements equal to each possible output value, that is, $H(k) = \#\{i \mid a_i = k\}$.

Little, Blelloch, and Cass[6] describe three approaches to histogramming: sequential iteration through the value set, additive writes, and sorting. Let us consider the sorting method in more detail.

We first sort the data so that $a_{\pi(i)}$ forms a nondecreasing sequence. For example, the Batcher bitonic sort algorithm[8] operates on a hypercube machine in $O(\log^2 N)$ time. After sorting, each processor can determine if the data in the processor to its left is different. If so, it marks itself as the head of a constant-data block. Since each processor needs to be able to communicate with its neighboring processor for this step, the processors should be configured as a one-dimensional array embedded in the hypercube, using a gray-code embedding (the Batcher sort process is still efficient in this configuration). Next, each head processor counts the number of processors in its constant-data block by means of a segmented parallel prefix sum. Finally, each head processor sends the information about the cardinality of its block to the appropriate histogram bin, $a_{\pi(i)}$. Since the destinations of the messages are distinct and ordered relative to the source indices, these messages can be sent using an $O(\log N + \log V)$ contention-free algorithm of the sort described by Nassimi and Sahni.[9] The total complexity of histogramming by sorting is thus $O(\log^2 N + \log V)$.

For our purposes, the histogram vector is not needed; rather, we only need knowledge of the few maximum-vote-getting values. To this end, the final stage of sending messages can be omitted, and the maximum counts among the marked processors can be determined and relayed to the front end. Thus, the process of finding the few maximum histogram bins can be accomplished in $O(\log^2 N)$ time.

Further, we can do better than the Batcher sorting algorithm. Lin and Kumar[10] provide a hypercube-based radix sort algorithm; in the sidebar entitled "Simple radix sort on a hypercube," we outline a simpler method that also has time complexity $O(\log V \times \log N)$.

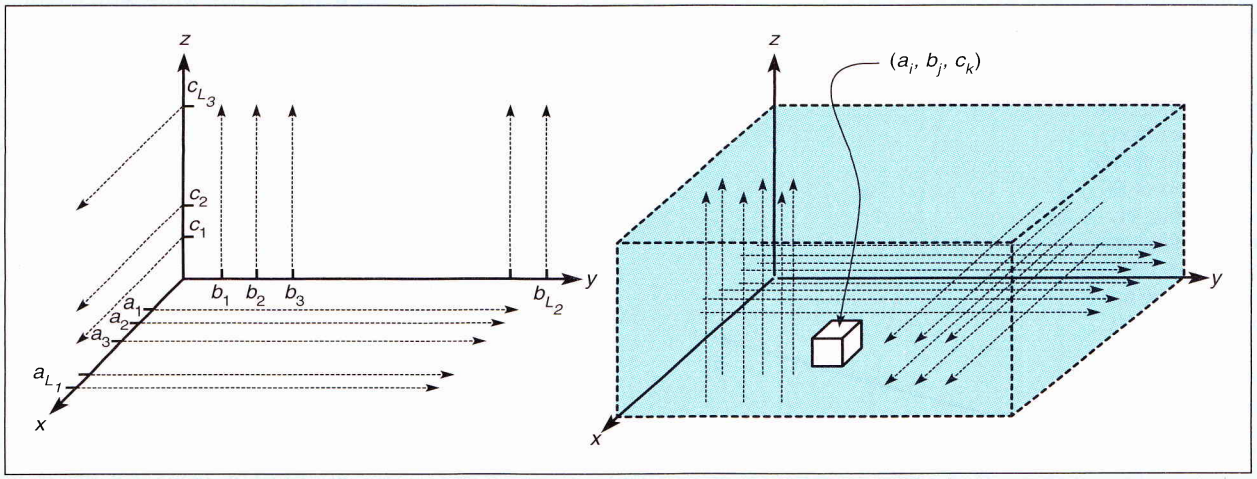


Figure C. The two stages of the parallel triple product computation.

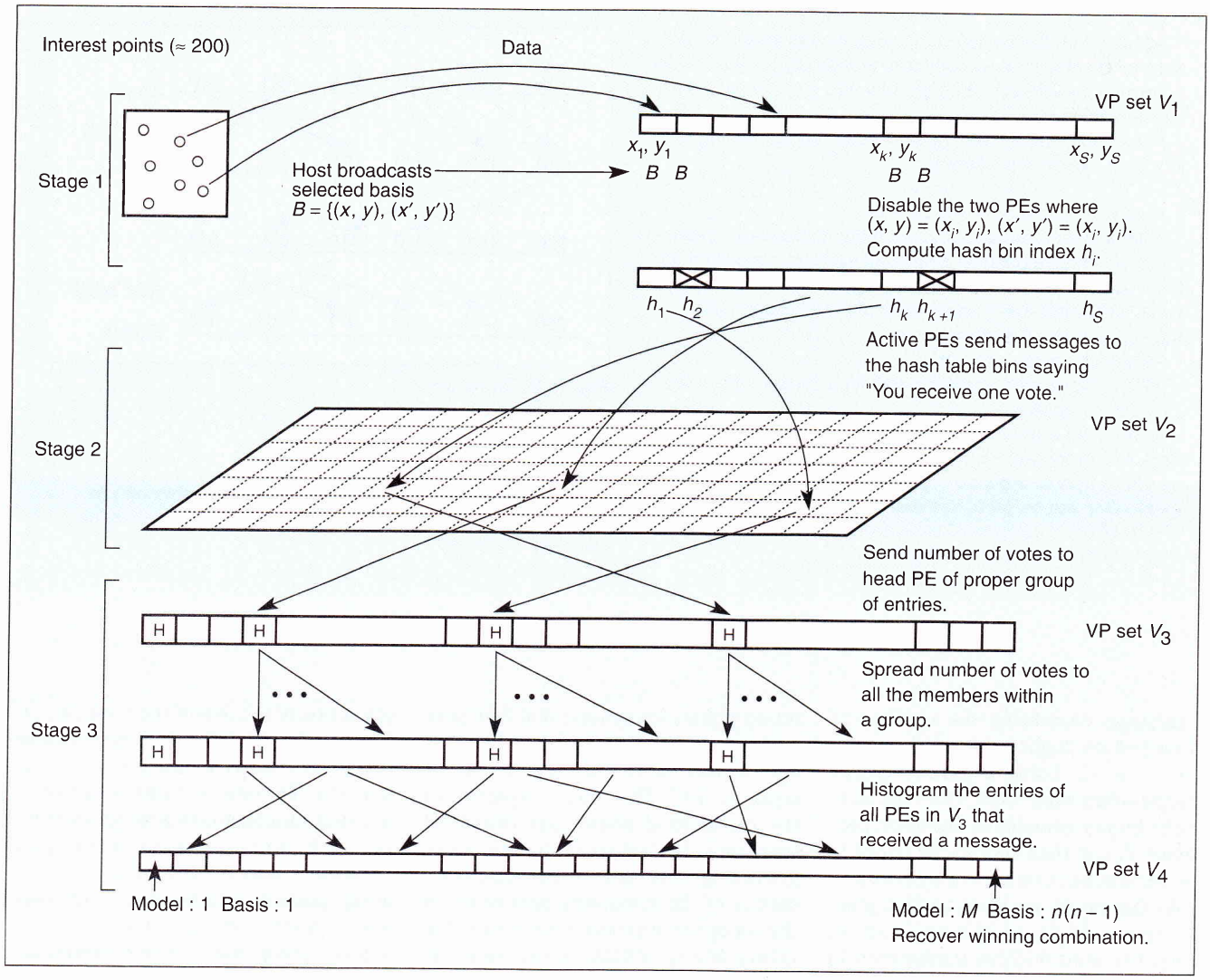


Figure 1. Recognition phase of the parallel geometric hashing connectionist algorithm. Note how tokens flow from one set via connections to the next set.

incrementing the map entry pointing to the head of the corresponding hash bin's block. Collisions can occur if more than one processor wishes to append to a single hash bin's list. By using a SIMD version of a parallel fetch-and-add instruction, each processor requesting a destination virtual processor in a hash table bin is assured a distinct address while indivisibly (and concurrently) incrementing the counter pointing to the next empty location of the block.

The hash table is now contained in two data structures. The first structure, one processor for each hash bin h, contains pointers to a head processor T_h of the block of entries in the second data structure. The second data structure consists of at most $Mn(n-1)(n-2)$ hash bin entries of the form (m, i, j).

Recognition. In Figure 1, a schematic diagram of the recognition phase, we use the virtual processor set (VP set) concept found in Connection Machine literature. A VP set is simply an abstract finite set that will be mapped to virtual processors. The VP sets in the recognition phase are the feature coordinate set V_1 and the hash table sets V_2 and V_3. We assume that a set of S interest points have been extracted from the scene and that each coordinate pair resides in the local memory of one of the S processors of the VP set V_1. For the hash table, VP set V_2 contains the pointers to the heads of the blocks of entries, and V_3 is the one-dimensional array of concatenated lists of hash entries.

In the first stage, the front end selects a basis pair in the scene and broadcasts the pair's coordinates to the S processors of V_1. Each processor in V_1 combines the coordinates of its interest point with the broadcast pair to compute the index of a hash bin. In the second stage, messages saying "you receive one vote" are sent by the processors of V_1 to the appropriate processors in V_2. The messages are sent using additive writes and general routing; multiple votes destined for the same recipient processor combine in the routers. In the last stage, every V_2 processor h that received one or more messages relays the number of votes it received to the block of processors T_h through $T_{h+1} - 1$ of V_3. This operation can be done, for example, using a modified version of Nassimi and Sahni's Generalize algorithm.[9] Alternatively, every V_2 processor h can send

Simple radix sort on a hypercube

Assume that the values in the sequence to be sorted, $\{a_i\}_{i=1}^{N}$, are represented in binary bit form, and let $\{b_{k,i}\}_{i=1}^{N}$ be the sequence of the kth-from-the-right bits. We sort the values in a stable fashion.

For k beginning at zero, and successively increasing to $\log V - 1$, we do the following:

- Mark all processors with $b_{k,i} = 0$.
- Rank these processors: Each marked processor determines its relative position among all marked processors using a parallel prefix sum (Nassimi and Sahni[9] describe a Rank algorithm). Let r_i be the rank of a processor if it is marked and t be the maximum r_i.
- Mark all processors with $b_{k,i} = 1$.
- Rank these processors as well; let s_i be the rank of the ith such processor.
- Move the a_i data: Every processor with $b_{k,i} = 1$ sends its data a_i to processor $t + s_i$, while every processor with $b_{k,i} = 0$ sends its data to processor r_i. Because the paths of communication are ordered, this routing can be completed in $O(\log N)$ time, using the Concentrate and Distribute algorithms from Nassimi and Sahni.[9]

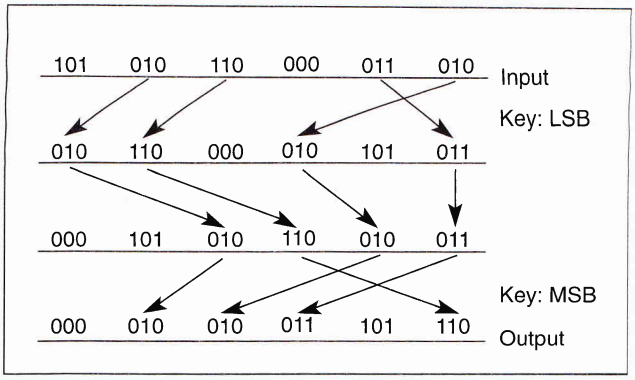


Figure D. Radix sort.

After the first iteration, all items are stably sorted with respect to their low-order bit. Upon termination, the sequence $\{a_i\}_{i=1}^{N}$ will be sorted.

a message containing the number of votes (which might be zero) to processor T_h in V_3. Using a parallel prefix computation with "copy from the left" as the binary associative operator, processor T_h can then spread the count to the remaining members of its group.

At this point, we want to histogram the entries of the processors in set V_3 using the multiplicities determined in the previous step. Efficient histogramming methods should be utilized; in particular, the radix sort algorithm (see sidebar above) offers advantages. In our current implementation, purely for ease of coding, we use additive writes instead. Consequently, a fourth VP set, the histogram bin set V_4, is required. Each processor of V_4 is associated with one histogram bin representing a triplet (m, i, j). The processors of V_3 vote for their (m, i, j) entries by sending an additive write message to the appropriate histogram bin. The increment in these messages is the value of the votes they received in the algorithm's third stage. The additive writes combine in the routers, resulting in histogram counts in the $Mn(n-1)$ histogram bins. Lastly, a global-max operation (or a thresholding) of the vote tallies over the processors of the set V_4 recovers the winning (m, i, j) combinations. These combinations are communicated to the front end to verify the existence of matching models.

For the time complexity of the recognition phase, we assume that Mn^3 processors are available and that the number of hash table bins is less than or equal to Mn^3. The time complexity of the recognition phase, per broadcast basis pair, is dominated by the histogramming step. In fact, the time complexity of the remaining operations of the recognition phase is no worse than $O(\log(Mn^3))$, which is the same as $O(\log(Mn))$. The complexity of histogramming depends on the particular method. Using Batcher's bitonic sort[9] for histogramming, the time complexity of the recognition phase is $O(\log^2(SMn))$, where S is the number of feature points. Using the radix sort algorithm lowers it to $O(\log(SMn)\log(Mn))$.

Hash-location broadcast algorithm

The above algorithm approaches geometric hashing as a connectionist process, with information flowing via patterns of communication. Our second algorithm uses the Connection Machine as an intelligent memory source and is inspired by the inverse indexing method of data retrieval.

Hash-table data structure. The hash table is organized differently in this algorithm. The data can be regarded as a collection of records of the form (m, i, j, k, x, y), where (x, y) is the hash location that point k maps to under basis (i, j) in model m. The information can be stored in a multidimensional table indexed by (m, i, j, k), where i, j, and k are integers between 1 and n. Not all of the Mn^3 array locations will be used. Locations where point k occurs in the basis (i, j) will be empty, and all bins corresponding to poor basis combinations (a basis with two very close points) will also be empty. The (m, i, j, k) information in each record can be recovered from the self-index of the entry location in the array, although we found it convenient to store the data explicitly along with the (x, y) values.

Preprocessing. The construction of the hash table is a simplification of the previous algorithm's preprocessing phase. After loading the model point data into the appropriate processors, M simultaneous triple products of $S_m \times S_m \times S_m$ are computed for $m = 1, 2, \ldots, M$ in $O(\log n)$ time. This forms the four-dimensional array. Then every location that can compute a reasonable hash location computes the corresponding (x, y) value and stores the information locally.

Recognition. Figure 2 shows a schematic diagram of the recognition phase. The VP set V_1 contains the feature coor-

dinate set, obtained from the S interest points (S is typically 200) that have been extracted from the scene, and V_2 contains the four-dimensional hash-table data structure preloaded into the memory.

We operate sequentially through the basis sets and the scene points. First, the front end selects a basis pair in the scene and broadcasts the coordinates of the points in the pair to the S processors of V_1. The two processors whose interest points form the selected basis do not participate in the coordinate computation; the remaining $S - 2$ processors then compute the hash location of the locally stored scene point in the coordinate frame of the broadcast basis. These operations involve minimal data movement and thus are extremely fast.

In the second stage, the data from the $S - 2$ processors in V_1 are successively broadcast to all processors of set V_2. Each broadcast coordinate has the form (u,v) and gives a hash table location where a vote should be tallied. Each processor in V_2, indexed by (m, i, j, k), contains a hash location (x, y), which the processor can compare against (u,v). If the two locations are sufficiently close, table location (m, i, j, k) records a hit indicating a vote for model m and basis (i, j). (An extremely useful modification to geometric hashing permits weighted voting for model-basis pairs according to the relative proximity of (u,v) to (x, y).) The vote tallying continues by accumulating hits in each hash table location for each of the $S - 2$ points in the image.

When the tallying is complete, a third stage uses a segmented, parallel, tree-based sum operation to add the votes over k among locations (m, i, j, k). The result is the total number of votes that model m with basis (i, j) obtains for the given scene and basis selection. Finally, a global-max or thresholding operation among the processors with locations holding the sum of votes determines the winning model/basis combinations. A final verification step determines the quality of each match.

Strictly speaking, each of the $S - 2$ broadcasts will require $O(\log(Mn^3))$ time, since there are Mn^3 processors in the V_2 data set. However, the theoretical complexity can be decreased, at the expense of requiring S storage locations in each processor, assuming that $S \leq Mn^3$. Assuming for simplicity that $S = n^2$, all $S - 2$ broadcasts can be done simultaneously by having each processor in V_1 send its data to a unique processor in a two-dimensional $(n \times n)$ slice of the four-dimensional data set V_2. This routing can be completed in time $O(\log(n^2))$. This slice of data can then be spread to the rest of V_2, in parallel slices, requiring no more than $O(\log Mn)$ time.

Observe that at this point the entire set of the computed coordinate pairs is distributed among the n^2 processors of a slice, one coordinate pair per processor. The processors within a slice can now exchange their data so that the entire list of computed coordinate pairs becomes available to each of them. This can be achieved simply by a recursive doubling procedure that communicates data between pairs of processors and forms lists of coordinate pairs. Note that the entries of those lists will not appear in the same order in each processor. This recursive doubling procedure can be completed in $O(S)$ time.

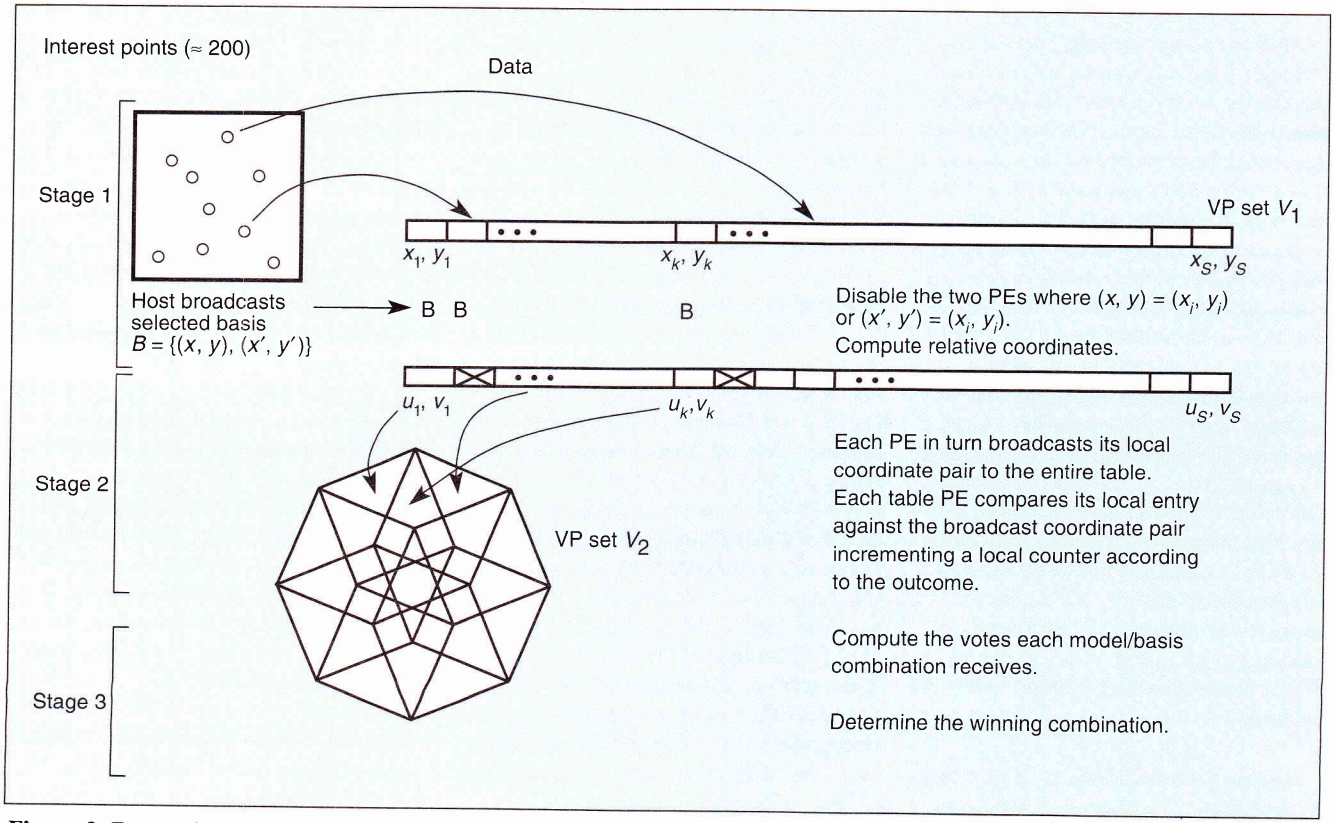


Figure 2. Recognition phase for the hash-location broadcast algorithm.

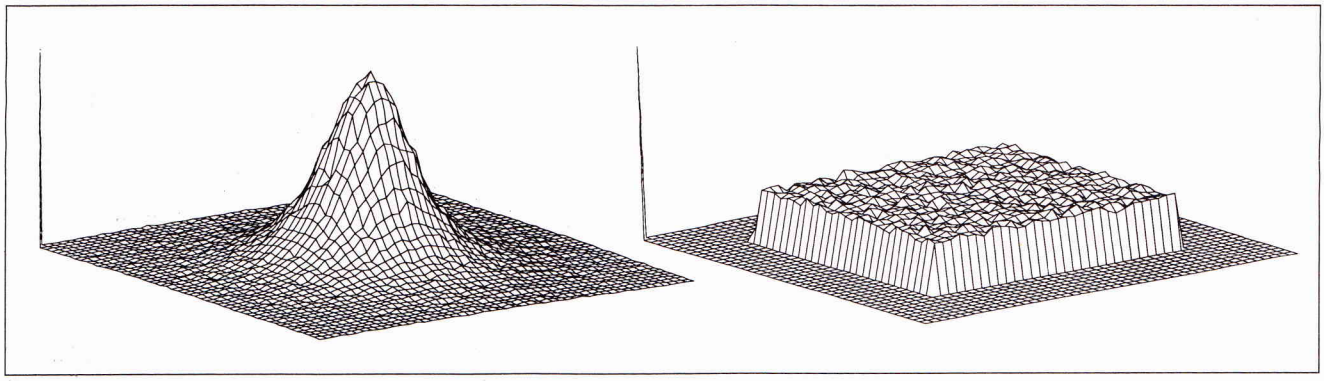


Figure 3. Hash table equalization for similarity transformations. The model points have a Gaussian distribution over R^2. Shown at left, the hash table before equalization; at right, the hash table after equalization.

The time complexity of the entire recognition phase is dominated by the second stage. Indeed, the time complexity of the first stage is $O(\log S)$. The second stage, using the data-spreading trick just described, results in time complexity no worse than $O(S + \log(Mn))$, which is also the complexity of the recognition phase.

Implementation on a Connection Machine

Implementing carefully crafted parallel algorithms on existing architectures frequently involves more compromises than one might expect. In our case, our algorithms have assumed the existence of Mn^3 processors, which for $M = 1{,}024$ and $n = 16$ would entail a 4-million-processor machine. Although there are 64K-processor Connection Machines, it is more usual to have access to a 32K- or 16K-processor model and to do prototyping on an 8K-processor model. We can use Connection Machine software facilities to simulate a larger parallel machine by mapping multiple virtual processors to each physical processor, which then execute in round-robin fashion. However, the virtual processor ratio (VPR) would then equal 512 on an 8K-processor model, which incurs an impractical amount of overhead. Accordingly, we must modify the algorithms somewhat and employ other efficiencies.

Hashing connectionist algorithm. Rather than give each hash table entry a separate processor, we can store the entire list of entries for a hash bin in a single processor's local memory. The lengths of the lists will vary over the hash table, but the required number of processors drops to the number of desired hash bins. The preprocessing phase of creating the hash table becomes far less efficient, due to the processor's need to randomly access local memory as entries are appended to the lists. Further, collision contention becomes more delicate. But, provided no single list becomes exorbitantly long, memory requirements are not a problem.

For the recognition phase, the entries in the hash bins that receive votes must be histogramed (that is, counted) with the multiplicity of the number of votes that each hash bin receives. For 1,024 models having 16 points each, the entries consist of 18-bit codes (10 bits for the model number, and 4 bits for each of the two basis points). Rather than histogramming by sorting, we opt for a message-passing strategy. We set up $2^{18} =$ 256K buckets (which requires a virtual processor ratio of 32 on the 8K-processor machine), one bucket for each (m, i, j) combination. Each hash bin that receives one or more votes from the scene points then synchronously walks down its list of entries, sending messages to the corresponding (m, i, j) buckets. On the 8K-processor machine, each hash bin has, on the average, 420 entries in its list. The time needed for list traversal is clearly dominated by the longest list. This process currently accounts for 99 percent of recognition-phase execution time, and it uses the parallel architecture less efficiently than the radix-sort approach to histogramming.

To make the process as efficient as possible, we suggest two enhancements. First, it is clearly desirable to keep the lists of entries as even as possible over the hash bins. By employing a rehashing function, we can effectively requantize the hash table such that the expected density of list lengths becomes uniform.[11] Figure 3 plots the hash bin occupancies before and after requantization for a typical database of models.

Second, we can use certain symmetries in the hash table; in particular, if an entry of the form (m, i, j) hashes to a location (x, y), then there will be an entry (m, j, i) in location $(-x, -y)$. Thus, when calculating a point's hash location, we can remap points from the lower half plane to the upper half plane, and either confuse entries of the form (m, i, j) and (m, j, i) (which could cause some degradation in discriminability) or mark such remapped hashings as "basis-inverted." Accordingly, only half the hash table is required, and the entry lists become, on the average, half as long when spread over the existing processor set.

Hash-location broadcast algorithm. To reduce the virtual processor ratio in the second algorithm's implementation, we assign one processor to each index (m, i, j) and store the n entries associated with $k = 1, 2, \ldots, n$ in its local memory. The computed hash location for each of the S scene points is broadcast to all processors, and each processor compares the location with the n locations stored in its local memory. Thus, Sn comparisons are needed (per basis probe).

Efficiencies can be achieved by the symmetries mentioned above. Each of the entries (m, i, j, k, x, y) is mirrored by

an entry of the form $(m, j, i, k, -x, -y)$, so by comparing a broadcast location (u,v), where $v < 0$, with $(-x, -y)$ instead of (x, y), we can omit half of the hash table entries. Further, through additional preprocessing of the local hash table entries, and by broadcasting the S points in an appropriate order, processors can use a two-dimensional version of list merging and avoid looking through the entire lists. This enhancement, which requires nonuniform random access to local processor memory (supported in a limited fashion on the Connection Machine), reduces the theoretical complexity to $O(S + n)$ but does incur other overhead.

Languages. Although a number of special-purpose parallel languages have been developed for the Connection Machine, we found C code running on the front end, enhanced with system calls to the Connection Machine using its Paris package, the most suitable for our needs. The Paris package includes many of the building-block and routing algorithms that we have mentioned; thus, it gives us the greatest level of control over the machine. For the broadcast-based algorithm, just about any language would suffice, and the Paris primitives represent a fast development path.

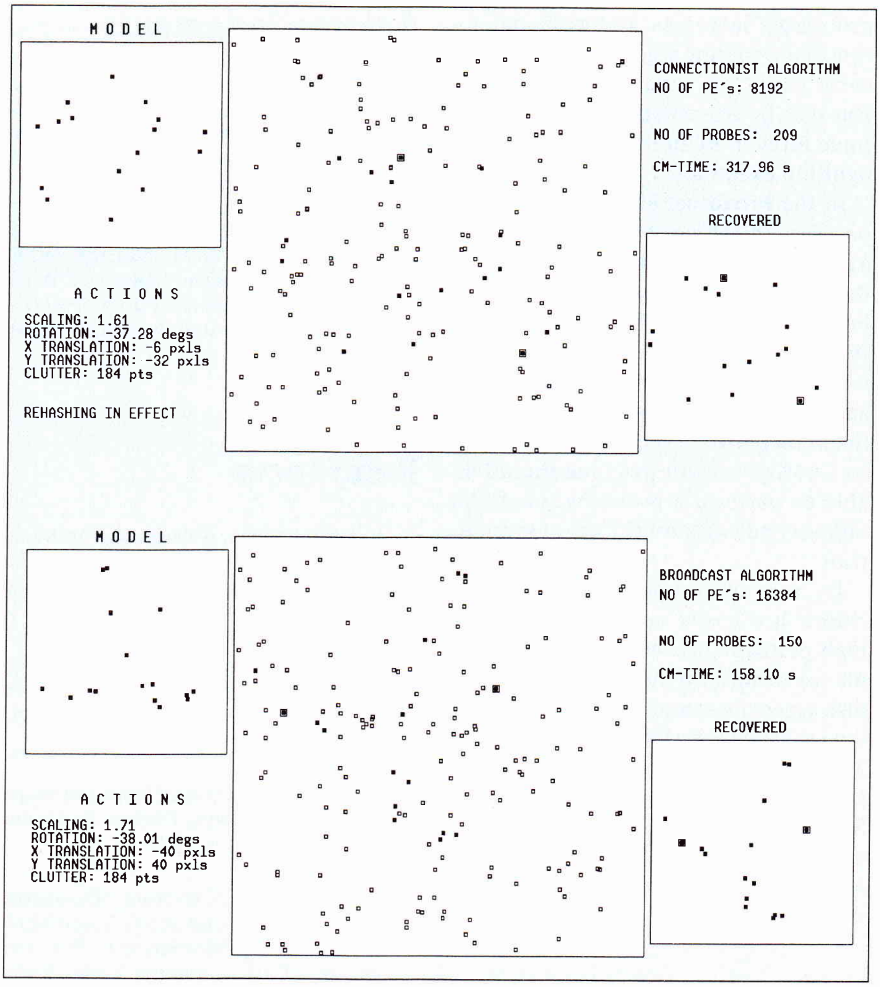


Figure 4. Recognition examples for the connectionist algorithm (top) and the broadcast algorithm. In the center panels, the embedded model is shown in solid dots, and the randomly selected basis points that resulted in recognition are shown boxed. Full recognition required 209 probes in the top example, 150 in the lower one. The database contained 1,024 models.

Performance results

We generate models (dot patterns) of 16 points each, using either a uniform distribution over a region or a Gaussian distribution. After generating 1,024 models, we construct scenes of approximately 200 points, such as those shown in Figure 4. A single model is embedded in the scene, translated, rotated, and scaled. Noise is added to the scene points through quantization round-off error.

In both implementations, the front end randomly selects a pair of scene points (a probe) as the basis for possible recognition. A connectionist algorithm probe takes 1.52 seconds on an 8K-processor machine, which drops to 0.24 seconds on a 32K machine, using rehashing but not the symmetries described in the preceding section. If the symmetries were used, the probe time would drop accordingly. The plots in Figure 5 show the connectionist algorithm operating in a roughly linear regime — that is, we are achieving linear speedup due to the heavy loading. In fact, as the number of

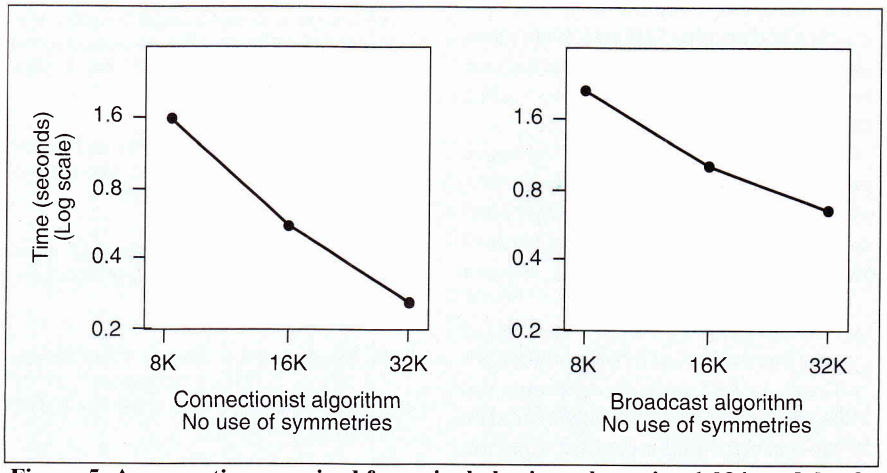


Figure 5. Average time required for a single basis probe, using 1,024 models of 16 points each, as a function of the number of processors in the Connection Machine; the scenes contain 200 points.

processors increases, reduced contention in the routing algorithm gives us, in some cases, an apparent extra boost. But such improvements would not continue forever. Figure 4 shows some recognition examples.

In the broadcast algorithm, the 8K-processor machine processes a probe at a rate of 10 milliseconds per scene point, that is, approximately 2.0 seconds for a probe using a 200-point scene. Neither symmetries nor list merging efficiencies were employed. Experiments with a 16K- and 32K-processor model indicate nearly linear increases in speed (see Figure 5), so a 64K-processor machine should be able to perform a probe in about 300 milliseconds without the use of symmetries.

By way of comparison, both algorithms are easily coded on a typical high-performance workstation. Performance results are highly dependent on disk access rates and available memory, but we have seen probe times of roughly 35 seconds for the equivalent of the hash-location broadcast algorithm on a Sun Sparcstation-2. (Remember that each of the processors in a Connection Machine is merely a slow bit-serial ALU.)

Recognizing all models embedded in a scene, if there are many, would require many probes. Achieving processing times of seconds instead of minutes would require (1) access to a large parallel machine, (2) smart methods for choosing basis pairs in the scene, and (3) further performance enhancements. In the third area, we could make better use of symmetries and employ *foldings*, where multiple model/basis combinations are coalesced into a single bucket, to reduce the time for histogramming.

Both algorithms exhibit sublinear growth in execution time as the number of models M increases (using $O(M)$ processors), which is a hallmark of geometric hashing methods. However, the connectionist algorithm has better asymptotics as the number of points S in a scene increases, and it performed slightly faster in our implementations with 200-point scenes. The hash-location broadcast algorithm is simpler, and more amenable to improvements, and should ultimately prove superior when the number of scene points is not too large. ■

Acknowledgments

This research was supported by AFAL contract F33615-89-1087 and by an IBM Graduate Research Fellowship. Access to a Connection Machine was made possible through the DARPA Connection Machine Network Server Program; we especially thank the University of Maryland UMIACS center. We thank L. Tucker, A. Mainwaring, and E. Zelnio for discussions and support. H. Wolfson contributed substantially to the research, and his assistance is gratefully acknowledged.

References

1. L. Tucker and G. Robertson, "Architecture and Applications of the Connection Machine," *Computer*, Vol. 21, No. 8, Aug. 1988, pp. 26-38.

2. W.E.L. Grimson, *Object Recognition by Computer: The Role of Geometric Constraints*, MIT Press, Cambridge, Mass., 1990.

3. D.G. Lowe, *Perceptual Organization and Visual Recognition*, Kluwer Academic Publishers, Boston, 1985.

4. Y. Lamdan and H. Wolfson, "Geometric Hashing: A General and Efficient Model-Based Recognition Scheme," *Proc. Second Int'l Conf. Computer Vision*, Computer Society Press, Los Alamitos, Calif., Order No. 883, 1988, pp. 238-249.

5. Y. Lamdan and H. Wolfson, "On the Error Analysis of Geometric Hashing," *Proc. Computer Vision 91 and Pattern Recognition Conf.*, Computer Society Press, Los Alamitos, Calif., Order No. 2148, 1991, pp. 22-27.

6. J. Little, G.E. Blelloch, and T.A. Cass, "Algorithmic Techniques for Computer Vision on a Fine-Grained Parallel Machine," *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. 11, No. 3, Mar. 1989, pp. 244-257.

7. W.D. Hillis and G. Steele, "Data Parallel Algorithms," *Comm. ACM*, Vol. 29, No. 12, Dec. 1986, pp. 1170-1183.

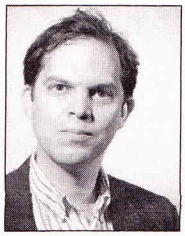
8. M.J. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, New York, 1987.

9. D. Nassimi and S. Sahni, "Data Broadcasting in SIMD Computers," *IEEE Trans. Computers*, Vol. C-30, No. 2, Feb. 1988, pp. 101-107.

10. W. Lin and V.K. Prasanna Kumar, "Efficient Histogramming on Hypercube SIMD Machines," *Computer Vision, Graphics, and Image Processing*, Vol. 49, No. 1, Jan. 1990, pp. 104-120.

11. I. Rigoutsos and R. Hummel, "Implementation of Geometric Hashing on the Connection Machine," *Proc. Workshop on Directions in Automated CAD-Based Vision*, Computer Society Press, Los Alamitos, Calif., Order No. 2147, 1991, pp. 76-84.

Isidore Rigoutsos is a PhD candidate in the Computer Science Department at New York University's Courant Institute. His interests include computer vision, parallel computing, neural networks, and applied mathematics. Rigoutsos received his BSc in physics from the National University of Athens, Greece and his MSc in computer science from NYU. He is a student member of the IEEE, the IEEE Computer Society, and the ACM.



Robert Hummel is an associate professor of computer science at New York University. His interests include computer vision, artificial intelligence and connectionism, and parallel computing. He is a member of the NYU Center for Neural Science and is involved in the planning for the NYU Center for Modeling and Simulation. In 1989, he was a visiting faculty member at Vrije Universiteit in Amsterdam, and he is currently on sabbatical, visiting Project Epidaure at INRIA Roquencourt in France.

Hummel may be contacted by electronic mail, hummel@cs.nyu.edu, or at his current address, INRIA — Project Epidaure, BP 105, 78153 Le Chesnay Cedex, France. Rigoutsos is reachable at rigoutso@cs.nyu.edu or at Courant Institute — NYU, 251 Mercer St., New York, NY 10012.