Scalable Parallel Geometric Hashing for Hypercube SIMD Architectures *

Isidore Rigoutsos
Courant Institute of Mathematical Sciences

Robert Hummel
Courant Institute of Mathematical Sciences
& The Center for Neural Science

New York University
251 Mercer Street
New York, NY 10012
rigoutso@cs.nyu.edu hummel@cs.nyu.edu
Telephone: (212) 998 3471 FAX: (212) 995 4122

Abstract

Geometric hashing has recently been introduced as a new paradigm for model based object recognition [3, 5, 7]. The geometric hashing algorithm allows one to find instances of model point patterns in a scene, subject to noise, obscuration, and transformation. In this paper we concentrate on the cases of similarity and rigid transformations and examine the parallelizability of the algorithm. We describe two scalable algorithms for hypercube SIMD architectures. A number of important building block algorithms and several variations to the basic approach are discussed. For a single probe with a two point basis, the algorithms have time complexities that are $\mathcal{O}(\log^2 M + \log S \log M)$ and $\mathcal{O}(S + \log M)$ respectively, using Mn^3-processors; M is the number of models in the database, n is the number of points per model, S is the number of feature points in the scene, and the model of computation is a SIMD hypercube. In order to locate an instance of a model, it is necessary to probe over $\mathcal{O}(S^2)$ basis pairs until a basis pair is found where both points belong to a model embedded in the scene. We have implemented both algorithms on the Connection Machine: using the resulting implementations, it is possible to recognize models consisting of patterns of points embedded in scenes, independent of translation, rotation, and scale changes, when there are thousands of models containing approximately 16 points each, with scenes consisting of hundreds of points, where most of the scene points are spurious noise points, and where embedded model points in the scene may be obscured or misplaced. With 1024 models and a scene of 200 points, we can achieve a probe time of 70 milliseconds on a 64K-processor CM-2 model.

Keywords: Hypercubes, Data Parallelism, SIMD Architectures, Histogramming, Model Based Image Understanding, Pattern Recognition, Rotation/Translation/Scale Independence, Geometric Hashing, Inverse Indexing.

*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 thank Lew Tucker and Alan Mainwaring for discussions and encouragement. Haim Wolfson contributed to both the research and manuscript, and his assistance is gratefully acknowledged.

1 Introduction

In a model-based vision system, features such as edges, lines, line-endings, corners, and textures are extracted and localized in digital imagery, and then compared with a database of models in order to identify, locate, and complete objects observed in 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, 4, 10]. A recent method, called *geometric hashing* [3, 5, 7], offers a different and more parallelizable paradigm.

In geometric hashing, the collection of models are used in a preprocessing phase (executed "off-line" and only once) in order to build a *hash table* data structure. The data structure encodes the information about the models in a highly redundant multiple-viewpoint way. During the recognition phase, when presented with a scene and extracted features, the hash table data structure is used to index geometric properties of the scene features to candidate matching models. A search is still required over features in the scene. However, the geometric hashing scheme no longer requires a search over the features in the model sets. The result is that the recognition phase offers computational efficiencies over more traditional model-based vision methods.

Geometric hashing search is highly parallel. As we will explain, there is parallelism available in the search over scene features, and there is parallelism in the indexing process.

In this paper, we explore the parallelizability of geometric hashing, and present two algorithms. We also present a number of modifications that are useful for efficient parallel implementation of geometric hashing for the particular case of point features. A number of building-block parallel algorithms that are used in the first of the approaches is also described; a radix-sort based histogramming algorithm for a hypercube-connected SIMD parallel machine is a particularly novel component of that approach.

We have implemented the algorithms on the Connection Machine. Using the resulting implementations, it is possible to recognize models consisting of patterns of points embedded in scenes, independent of translation, rotation, and scale changes, when there are thousands of models containing approximately 16 points each, with scenes consisting of hundreds of points, where most of the scene points are spurious noise points, and where embedded model points in the scene may be obscured or misplaced. The system will need to search over pairs of points in the scene, and recognition will be obtained as soon as a pair of points are chosen so that both points lie on the embedded object, although multiple pairs may be probed at the same time, and many heuristics exist for choosing likely basis pairs. With 1024 models and scenes consisting of 200 points, the implementation yields an execution time of approximately 70 milliseconds per probe, with a fixed basis set, on a 64K-processor CM-2 model. If multiple basis sets were probed at once, the execution time would increase, but not in proportion to the number of basis pairs.

A related parallel implementation of geometric hashing is reported by Bourdon and Medioni [1]. They achieved good parallelization of the preprocessing phase, but less parallelization of the recognition phase. In their implementation, the parallel capability of the Connection Machine is used to perform the numerical computations of the hashing process, but the communication aspect of the voting process is left to the host. Further, they deal with a small number of models in the model base. In particular, the hash table must be shared by all processors. In our study, we have attempted to distribute the hash table, so as to achieve sizable speed-up in the recognition phase in the presence of thousands of models. We are especially concerned with efficient histogramming methods in order to perform rapid voting.

2 Geometric Hashing: A brief introduction

We begin with an extremely brief summary of the geometric hashing idea. We give a high-level description of geometric hashing, then give an example for the case of recognition of dot patterns which may be subjected to translation and obscuration in the scene, but not rotation nor scale transformation, and finally present the algorithm for dot patterns subjected to rigid transformation or similarity transformation.

Features are extracted from the image, and a subset is designated as a *basis set.* The feature values are measured relative (in some sense) to the basis set. The values of the features are used as indices into a *hash table*, where records are kept of model features that map to the same location with respect to some basis set chosen from the model. Each such hashed feature votes for the set of possible models and basis sets stored at that location in the hash table. An important aspect of the geometric hashing paradigm is that an object is multiply encoded, using many different basis set selections. In this way, as long as a reasonable basis set is chosen in the recognition process, an identification will be made, since the basis set will be one of the many sets used to encode the object. Conceptually, the method can be seen as a sequence of measurements and maps, where each map projects to locations determined from the previous map by a process of fixed links and measured values.

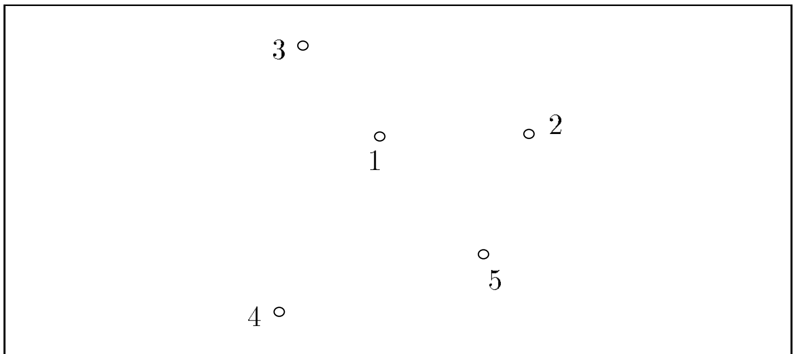

Figure 1. A model consisting of five dots.

At first glance, geometric hashing seems related to the Hough transform for feature extraction. However, the analogy goes only as far as the existence of a voting mechanism in both schemes. A discussion of the difference between geometric hashing and the Hough transform may be found in [5].

For concreteness, let us explain the geometric hashing scheme for dot patterns, where we wish to perform recognition of patterns that may be translated (but without rotation, scaling, or other transformations). Figure 1 shows a model that 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 locations (x, y). Let us record in a quantized hash table, in each of the four bins where this information lands, the fact that this model (M_1) with basis point "1" yields an entry in this bin. This is shown graphically in Figure 2, viewing only the entries of the form (M_1,1). Similarly, the hash table contains four entries of the form (M_1,2), and four entries of the form (M_1,3), etc. Each entry is located by placing the base point at the origin of the hash table, and observing where the other points of the model land. The resulting hash table, with the entries for all base points of model M_1, is shown in Figure 2. The same process is repeated for each model. Of course, it can happen that any one hash bin receives more than one entry.

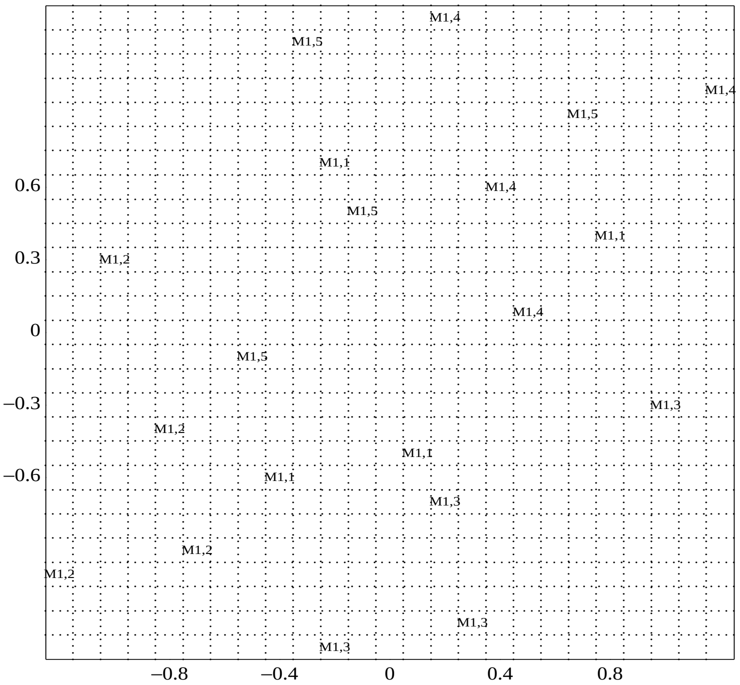

Figure 2. The hash table entries for all base points of Model M_1.

As a result, the final hash table contains a list of entries of the form $(model, basepoint)$ in each bin.

In the recognition phase, a single point from the scene is chosen as the basis point. The coordinates of all other points are then calculated with this point placed at the origin. Each point is mapped to the hash table, and all entries in the corresponding bin receive a vote. If there are sufficient votes for a particular $(model, basepoint)$ combination, $(m, \mathbf{p})$, then a subsequent stage attempts to verify the presence of model m with the designated point $\mathbf{p}$ located at the chosen basis point. Note that if a point is missing from the scene because it is obscured, recognition is still possible, as long as there are a sufficient number of points that hash 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 the configuration of transformed points coincides with a model. In general, we do not expect the voting scheme to give only one candidate solution (see [6]). The goal of the voting scheme is to reduce significantly the number of candidates for the verification step which might be quite tedious and time consuming (see [13]).

The cases where the dot pattern has undergone rotation and translation are treated analogously with two points now needed to define a basis. Refering again to the model in Figure 1, suppose that points "4" and "5", in this order, are selected as the basis. Let us position the model in such a way that the basis aligns with the horizontal axis of a coordinate system whereas the basis' midpoint coincides with the $(0, 0)$ of that system (Figure 3). Then the remaining dots lie at three locations (x, y). A quantized hash table will now record, in each of the three bins where the dots land, the fact that the 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 with the basis positioned as described above. Each of the points is again mapped to the hash table, and all the entries in the corresponding bin receive a vote; we continue as above. If the dot pattern is also allowed to

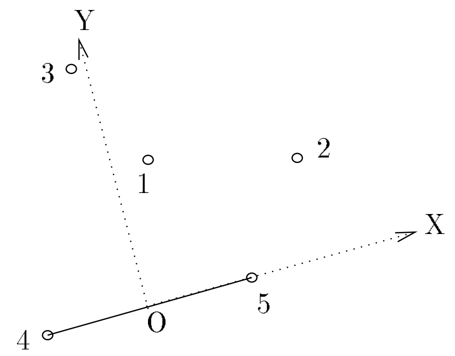

Figure 3. Using two point bases.

undergo scale changes, then one need only use the length of the selected basis as the length of the unit vector of the coordinate system.

Since the voting can be done simultaneously for all models and all possible bases on a model, *for the algorithm to be successful it is sufficient to select scene points belonging to some model, as a basis tuple.* In such a case, the model with the appropriate basis will receive a high score in the voting procedure. Classification or perceptual grouping of features can be naturally incorporated into this method by concentrating only on some special basis-pairs (see [7] for a method which utilizes concavity entrances of object boundary curves).

3 Building-block Algorithms

Two building-block algorithms will be used in the first of the two geometric hashing implementations. The algorithms are a *triple-product* computation, and *histogramming.* We now describe the algorithms for a hypercube-based SIMD machine. We assume a CREW model of computation such that any pattern of concurrent reads to neighboring processors uses unit time. (Accesses are permitted along different dimensions in the same clock cycle).

3.1 Triple Product

The triple product problem is defined as follows: given three finite sets $A = \{a_i\}_{i=1}^{l_1}$, $B = \{b_j\}_{j=1}^{l_2}$, and $C = \{c_k\}_{k=1}^{l_3}$ compute the triple product $A \times B \times C$, i.e. 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 twice an outer product, so as to compute first $A \times B$, and then $(A \times B) \times C$. An outer product for the Connection Machine is succinctly described in [9]. A straightforward 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 by l_2 by l_3. (We must assume, at this point, that the l_i are powers of 2). The processors are indexed by their coordinates (i, j, k) which we regard as lattice points in (x, y, z)-space. We assume that, 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 4). When completed, the processor $(i,j,0)$ has the datum a_i, the processor $(0,j,k)$ has the datum b_j, and the processor $(i,0,k)$ has the datum c_k. In the second phase,

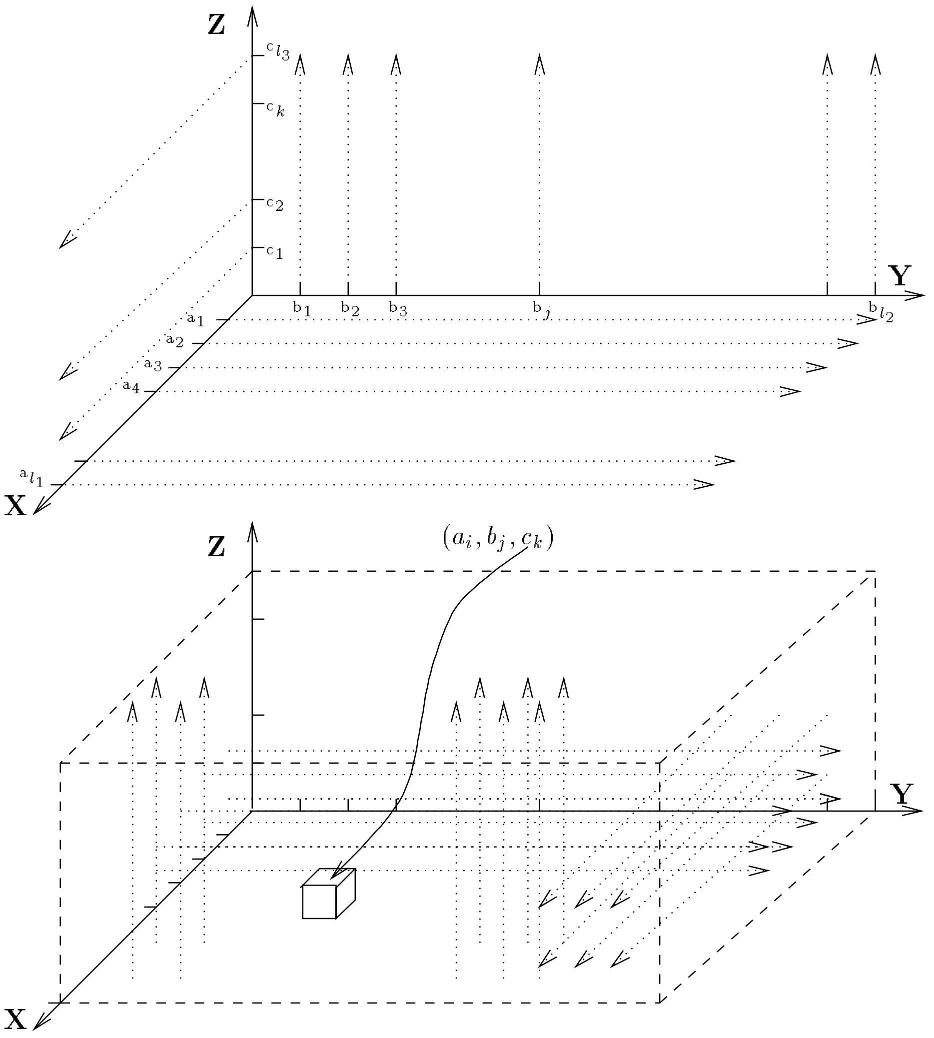

Figure 4. Triple Product: the two stages.

the data along 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 (x,z)-plane along the y-axis (see Figure 4). When completed, 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 $\mathcal{O}(l_i)$ time (for the appropriate axis), since nearest neighbors are adjacent in the hypercube, but can in fact be completed in $\mathcal{O}(\log l_i)$ time. This is because a parallel prefix operation may be used. Namely, the operation is equivalent to a collection of concurrent parallel prefix computations with "copy from the left" as the binary associative operator. The parallel prefix computation makes use of a recursive doubling scheme to spread the data rapidly along the axis. Power-of-two communication along each axis is provided by $\mathcal{O}(1)$ 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 communications cycles on a hypercube. This is true for any value of k. In the parlance of the Connection Machine Paris language, the operation is a "scan_with_copy."

3.2 Histogramming

Histogramming is defined as follows: 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 $a_i \in \{1, 2, ..., V\}$ determine a count of the number of elements equal to each possible output value, i.e., $H(k) = \#\{i \mid a_i = k\}$.

As pointed out in [9], there are three distinct approaches to parallel histogramming:

(1) Sequentially iterate, from 1 through V; for each value k, allow each a_i to mark itself if it is equal to k, and then perform a parallel sum to count the number of elements that were marked. Since parallel summing is $\mathcal{O}(\log N)$, this method has parallel complexity $\mathcal{O}(V \log N)$.

(2) Perform additive writes; each processor looks at its value a_i, and sends a message to processor a_i to increment an accumulator. There are two virtual processor sets, the initial set with N processors, one per data element a_i, and the V processors in the output, containing one accumulator per processor. The parallel complexity on a SIMD hypercube without additive writes is $\mathcal{O}(\log^2 N + \log V)$, although additive writes will typically be handled in an average-case more efficient method. In practice, the messages to increment accumulators will be combined in a probabilistic routing network, to avoid serialization at the location of the accumulators. The complexity is in all cases at least $\Omega(\log N)$, since if all messages are destined to a single processor, then the combination of the messages is equivalent to a global sum, but in practice, the complexity will depend upon V and N, the efficiency of the routing algorithm and the combining of messages in the router.

(3) Sort the data, so that $a_{\pi(i)}$ (where $\pi(\cdot)$ is a permutation) forms a nondecreasing sequence, for $i = 1, ...N$. For example, the Batcher bitonic sort algorithm operates on a hypercube machine in $\mathcal{O}(\log^2 N)$ 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 sorting process is still efficient in this configuration, although with a penalty in the proportionality constant. Next, a segmented parallel prefix sum is used to count the number of processors in each constant-data block and this information is delivered to the head processor of each block. Finally, each head processor sends the information about the cardinality of its block to processor $a_{\pi(i)}$, i.e., to the processor whose index is equal to the data item shared by the

processors of the block. Since the destinations of the messages are distinct and ordered relative to the indices of the source indices, these messages can be sent using an $\mathcal{O}(\log N + \log V)$ contention-free algorithm. The total complexity of histogramming by sorting is thus $\mathcal{O}(\log^2 N + \log V)$.

For our purposes, the histogram vector is not needed; rather, we only need knowledge of the few maximum-vote-getting values. For this purpose, in method (3) above, the final stage of sending messages can be omited, and the maximum counts among the marked processors can be determined and relayed to the front end. Thus the process of finding the maximum histogram bin can be accomplished in $\mathcal{O}(\log^2 N)$ time.

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

Using the l-th bit from the right as a key sort the data while preserving the order of equal keys:

a: $l \leftarrow 0$.

b: Mark all processors with $b_{l,i} = 0$.

c: Rank these processors: each marked processor determines its relative position among all marked processors using a parallel prefix sum (Nassimi-Sahni [11] describe a RANK algorithm). Let r_i be the rank of a processor if it is marked and t be the maximum r_i.

d: Mark all processors with $b_{l,i} = 1$.

e: Rank these processors as well; let s_i be the rank of the i-th such processor.

f: Move the a_i data: every processor with $b_{l,i} = 1$ sends its data a_i to processor $t + s_i$, while every processor with $b_{l,i} = 0$ sends its data to processor r_i — because the paths of communication are somewhat ordered, this routing can be done in $\mathcal{O}(\log N)$ time, (specifically, using the Nassimi-Sahni CONCENTRATE and DISTRIBUTE algorithms [11]). At this point, all items are stable sorted with respect to their low-order bit.

g: $l \leftarrow l + 1$.

h: If $l < \log V$, repeat steps b through g using the new bit sequence $\{b_{l,i}\}_{i=1}^N$.

Upon termination, the sequence $\{a_i\}_{i=1}^N$ will be sorted.

Figure 5. Simple Radix Sorting on a Hypercube.

In the actual implementation of geometric hashing, to be described later, we used method (2) above. However, the most efficient implementation would have been to use method (3) in conjunction with a radix sort algorithm. Lin and Kumar [8] provide a hypercube-based radix sort algorithm; however, because they sort from high-order bit to low-order bit, the algorithm is unnecessarily complicated. In Figure 5 we outline a simpler method for performing radix sorting on a hypercube; this method results in a complexity (as with Lin and Kumar's algorithm) of $\mathcal{O}(\log V \times \log N)$. An illustration of the method is given in Figure 6. The histogramming algorithm is completed as described in (3) above.

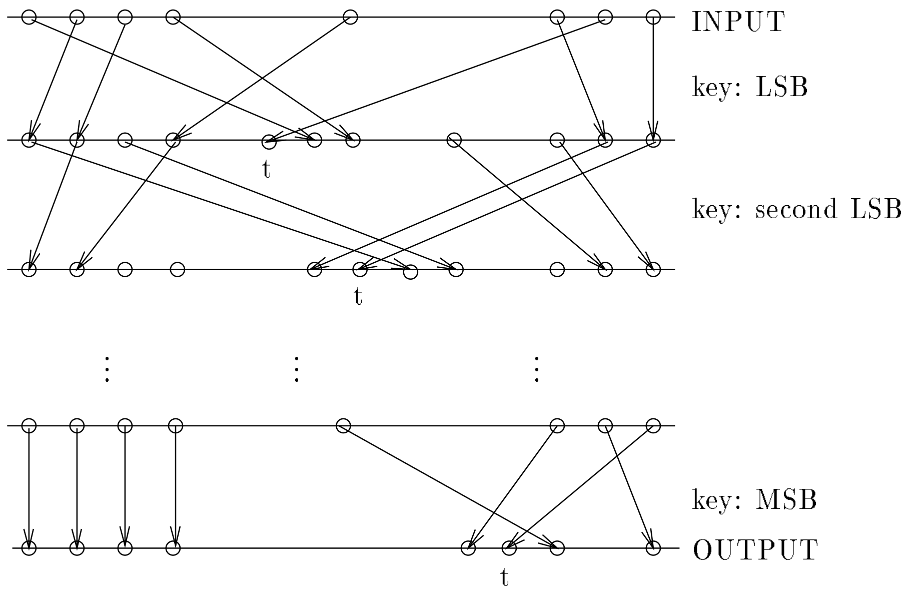

Figure 6. Radix Sorting: an illustration.

4 Data Parallelism over the Hash Table Bin Entries

In this section we describe the first of the two parallel algorithms for implementing geometric hashing. The algorithm is data parallel over the hash table bin entries and the scene points, but serial over the bases. In what follows we assume that our database contains M models; each model l has an associated set, $S_l = \{(x_{l,k}, y_{l,k})\}_{k=1}^{n}$, containing the coordinate pairs of the l-th model's n points.

4.1 Preprocessing Phase

During the preprocessing phase the hash table data structure is constructed. This phase is performed off-line. The hash table corresponding to a given database of model objects is built only once and then stored. This phase consists of two passes and is computationally expensive; the serial complexity is $\Theta(Mn^3)$ for two point bases. The proposed parallel algorithm has time complexity $\Theta(M \log n)$ if Mn^3 processors are used. With fewer processors the time complexity is higher; however, this is not of concern since the operation is performed off-line and only once. Three virtual processor sets participate in the first pass, namely the feature coordinates set V_1, the triple product set V_2, and the hash table set V_3. An additional set, V_4, the bin-entry set participates in the second pass. We assume that the number of hash table bins is less than or equal to Mn^3.

Figure 7 shows a schematic diagram of the first pass. The purpose of this pass is to determine the number of entries hashing in each bin of the hash table, when all the models in the database are considered. This pass consists of three stages.

We iterate sequentially over the models. We assume that at the beginning of the first stage, the lower order n processors of set V_1 contain the x and y coordinates of the n points in S_l for the l-th model; each processor contains the coordinates of exactly one such point. The set $(S_l \times S_l) \times S_l$ is then computed using the triple product algorithm described in Section 3.1. Each of the first n^3 processors of set V_2 now contains a triplet: $[(x_i, y_i), (x_j, y_j), (x_k, y_k)] \in (S_l \times S_l) \times S_l$. The first two points of each such triplet define an ordered basis and thus a coordinate system.

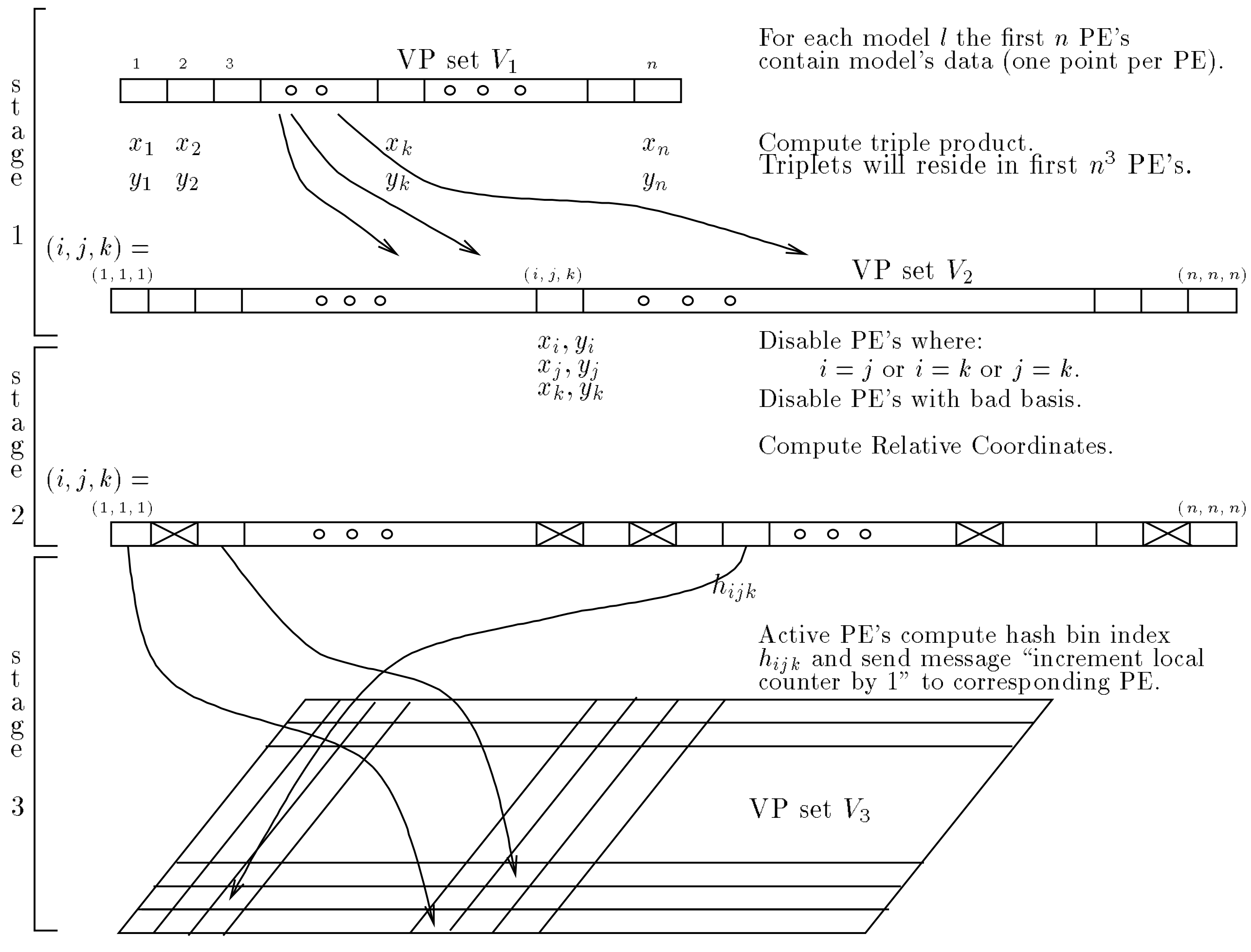

Figure 7. The first pass of a parallel algorithm that constructs the hash table data structure (Preprocessing Phase): determining the number of entries hashing in each bin of the hash table. The eventual hash table will have one processor per hash table entry.

In the second stage, each of the processors of set V_2 compute the coordinates of the third point of the triplet with respect to the coordinate system defined by the first two points of the triplet. In order to avoid numerical overflows resulting from degenerate bases, we disable those processors where $i = j$, as well as those processors where the locations of the first two coordinate pairs are extremely close. In addition, to ensure that the third point is distinct from the first two, we also disable those processors where $i = k$ or $j = k$. The precise formulas for the computation of the relative coordinates depend on the type of the allowed transformations (e.g. similarity or rigid transformation). Last, the active processors compute the coordinates of the third point in the triplet, using the appropriate function.

During the third stage, each of the active processors in V_2 use the coordinates of the third point to compute the index of the hash table bin (i.e., the address of the associated processor in V_3). This hash bin would traditionally contain entries of the form $(model, i, j)$, where model l is the *model*, and i,j are the local indices in the i,j,k triplet. The number of entries is updated by sending an "additive write with increment by 1" to an accumulator in the appropriate bin.

This process is repeated for each of the models in the database. When all of the models have been processed, each processor in V_3 has a count of the number of $(model, i, j)$ combinations that hash into

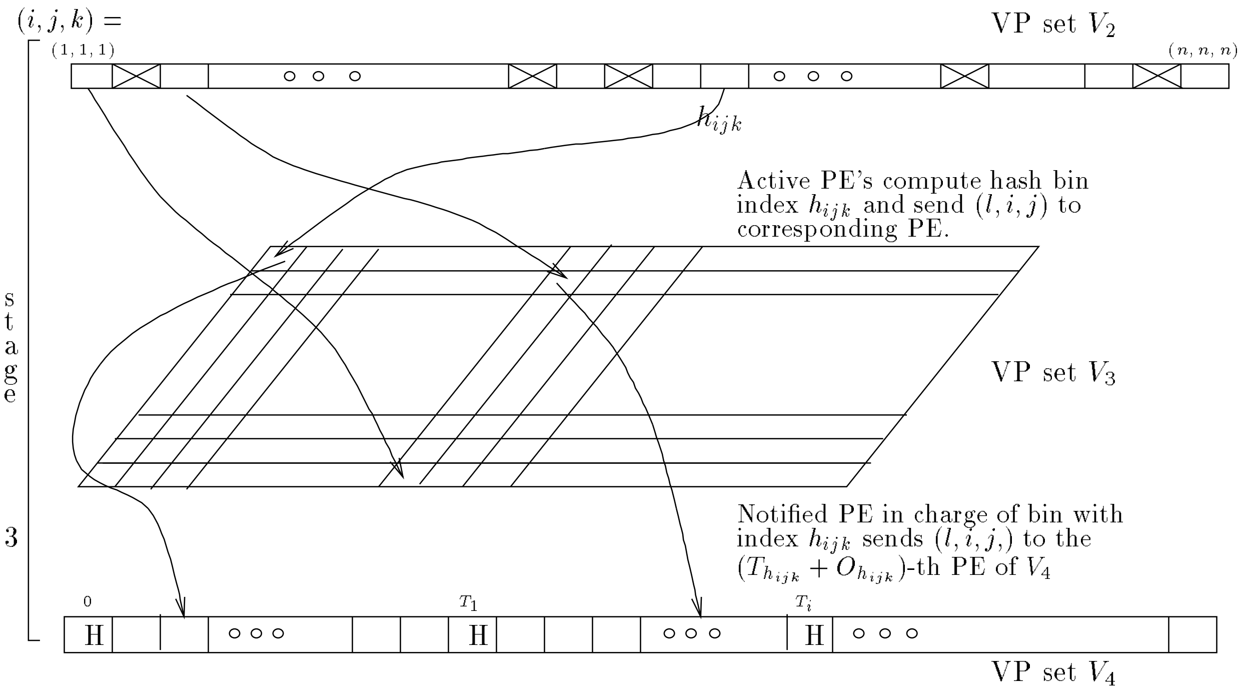

Figure 8. The second pass of a parallel algorithm that constructs the hash table data structure (Preprocessing Phase): mapping each hash table bin entry to one processor.

the corresponding hash table bin; let *number-of-entries* be that count. Using the index h of a hash table bin as a rank value, the corresponding processors of set V_3 can now order themselves. Performing a parallel prefix sum computation on the *number-of-entries* values, the h-th processor in V_3 obtains a value T_h that is the total number of entries hashing in hash table bins 0 through $h - 1$ inclusive. If all the bin entries were to be inserted in one-dimensional array, A, so that all the entries of bin h appear before any entry of bin $h + 1$, then T_h would be the starting offset for the entries of bin h.

During the second pass, each of the hash table entries are mapped onto exactly one processor from the processor set V_4; each processor of V_4 will contain one entry. The processors of V_4 that correspond to the same hash table bin can be thought of as forming a "group." The T_h-th processor of V_4 is associated with the first entry of hash bin h and is assumed to "head" all the processors of its group. Three stages comprise the second pass.

In the second pass, we again iterate sequentially over the models. The first two stages remain the same as before. During the third stage, each of the active processors in V_2 compute the index h_{ijk} (i.e., the address of the associated processor in V_3) where the tuple $(model, i, j)$ would be deposited and send a message containing $(model, i, j)$ to that bin. Each of the processors of V_3 also maintains a temporary local variable $O_{h_{ijk}}$ that is initially zero. The processors of V_3 that received a message use the previously computed $T_{h_{ijk}}$ to forward the tuple $(model, i, j)$ to processor $T_{h_{ijk}} + O_{h_{ijk}}$ of set V_4. Finally, the value of the offset within a group, $O_{h_{ijk}}$, is incremented by 1 on all of the currently active processors of V_3. A number of collisions are likely to occur when more than one processors of V_2 attempt to send distinct messages to the *same* processor in V_3. This problem can be solved by using a *Fetch-and-Add* operation on the variable $O_{h_{ijk}}$. However, in practice, the number of contending processors is expected to be small making this solution unnecessarily complicated. In our implementation a simpler protocol based on *locks* is used instead. Figure 8 shows a schematic diagram of this stage: some of the processors heading a group of entries are shown marked with an H. The above process is repeated for each of the models in the database.

The hash table is now a combination of two data structures. The first data structure (virtual processor set V_3) contains information about the offset of each group's head and the cardinality of the group (i.e. number of model/basis combinations hashing into the corresponding bin). The second data structure (virtual processor set V_4) consists of all possible hash bin entries ($Mn(n-1)(n-2)$) for the given database.

4.2 Recognition Phase

Figure 9 shows a schematic diagram of the recognition phase. Three sets of virtual processors participate in this phase, namely the feature coordinate set V_1, and the hash table set V_2 and V_3. We assume that a set of S interest points have been extracted from the scene and their coordinates reside in the local memory of the first S processors of the virtual processor set V_1. The hash table is preloaded into the local memory of the sets V_2 and V_3. Each processor from the set V_2 represents one hash bin and designates the processors of V_3 that actually contain the entries of that bin, one entry per processor of V_3. The offset of the head of the group of processors corresponding to the entries of hash bin h, as well as the cardinality of the group are maintained by the processor of V_2 in charge of bin h.

In the first stage, the front end selects a basis pair and broadcasts the coordinates of the basis points to the S processors of V_1. Recall that each processor in V_1 already holds the coordinates of an interest point. The two processors whose interest points coincide with the broadcasted basis endpoints are disabled. The remaining processors of the virtual processor set V_1 compute the relative coordinates of the corresponding interest point with respect to the broadcasted basis. They then compute the index of the hash table bin to be notified. The operations are extremely fast since they involve minimal data movement.

In the second stage, messages saying "you receive one vote" are sent by the active processors of V_1 to the appropriate hash bins. The messages are sent using additive writes and general routing; multiple votes destined for the same recipient processor combine in the routers. Each of the processors of the set V_2 maintains a local *votes* variable in order to keep track of the number of messages (i.e., votes) that the processor receives.

In the last stage, every processor h from set V_2 that received one or more messages in the previous step relay the number of votes (messages) they received to the block of processors T_h through $T_{h+1} - 1$ of V_3. This operation can be done in various ways. For example one can use a modified version of Nassimi-Sahni's GENERALIZE [11]. Alternatively, processor h from set V_2 can send a message

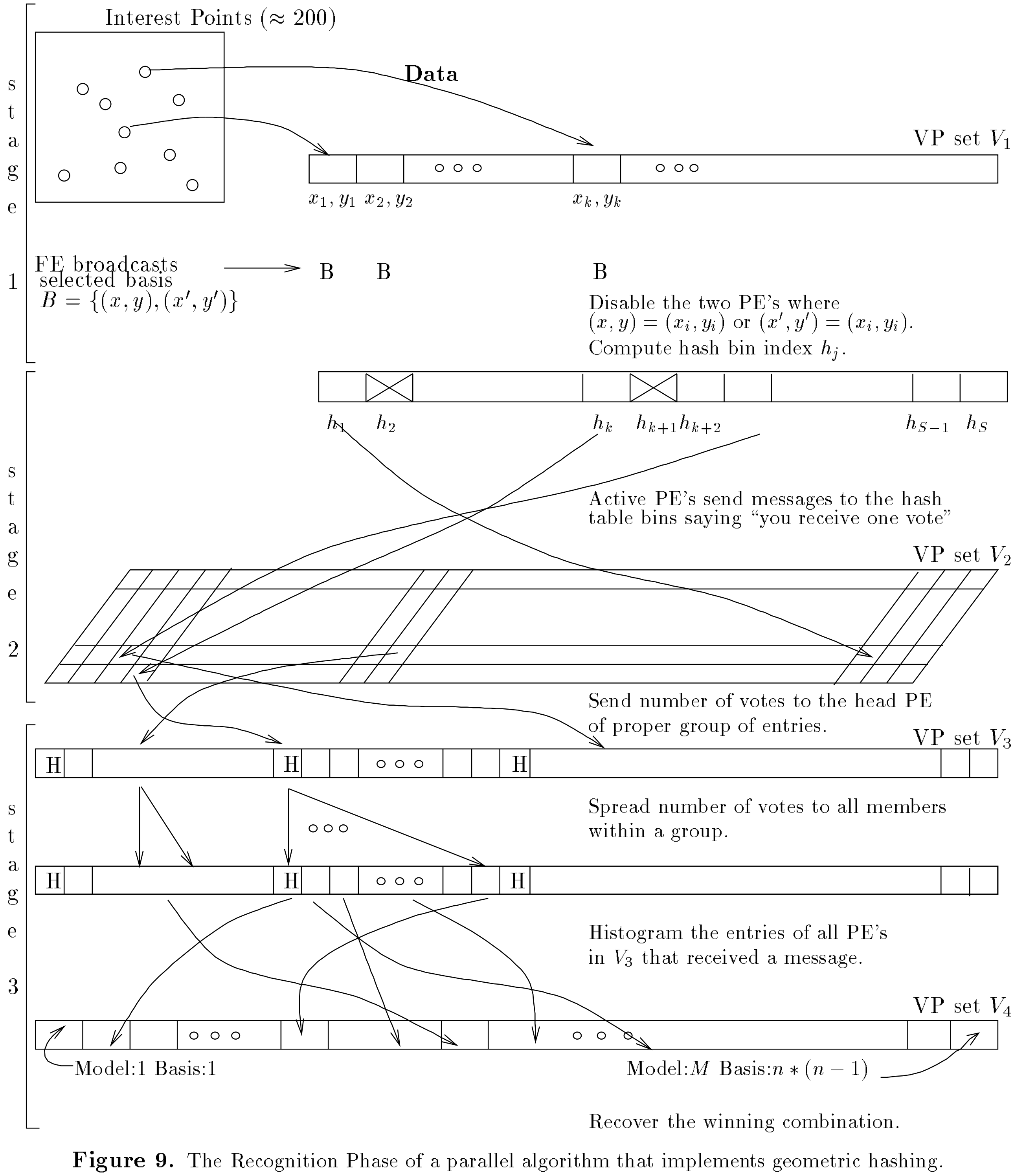

Figure 9. The Recognition Phase of a parallel algorithm that implements geometric hashing.

containing the number of votes to the 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 received value to the remaining members of its group (Figure 9). Only the processors of V_3 that correspond to entries of notified hash bins will now be active; these processors contain their own copy of the number of votes that the corresponding bin received.

At this point, we wish to histogram the entries of the active processors in the set V_3. These entries are $(\log M + 2\log n)$-bit words and the operation can be very expensive even when performed in parallel. However, as has already been noted, improved histogramming methods can be utilized, in particular, method (3) of Section 3.2 in conjuction with the radix sort algorithm of Figure 5. In our implementation, for purposes of simplicity, we chose to perform additive writes instead. Consequently, use of a fourth processor set, the histogram bin set V_4, is required. Each processor of V_4 is associated with one histogram bin representing a triplet $(model, i, j)$. The processors in the set V_3 vote for their $(model, i, j)$ entries by sending an additive write message to the appropriate histogram bin. The increment amount in these messages is the value of the *votes* variable at the processor of V_3 from where they originate. Votes destined for the same recipient combine in the routers. Last, a global-max operation of the vote tallies of each of the processors of the set V_4 recovers the winning (m, i, j) combination. The model is m and the broadcasted basis corresponds to the basis defined by the i-th and j-th points of that model; from this last correspondence we can recover the transformation the model has undergone, and verify the quality of the match.

We now consider the time complexity of the recognition phase — the only phase that is performed on-line. It is assumed that Mn^3 processors are available[1], and that the number of hash table bins is less than or equal to Mn^3, where M is the number of models in the database. Each model consists of n points.

It is easy to see that the time complexity of the recognition phase is dominated by the histogramming step. As a matter of fact, the time complexity of the remaining operations of the recognition phase is no worse than $\mathcal{O}(\log(Mn^3))$. As was previously discussed, the complexity of histogramming depends on the particular method that is used. If bitonic sorting is used to perform histogramming, the time complexity of the recognition phase is $\mathcal{O}(\log^2(SMn^3))$, where S is the number of feature points. On the other hand, if the radix sorting algorithm is used, the time complexity of the recognition phase drops to $\mathcal{O}(\log(SMn^3)\log(Mn^2))$.

5 Data Parallelism over the Model/Basis/Point Combinations

In this section we describe a second parallel algorithm for implementing geometric hashing. Again, it is assumed that the database contains M models, each one consisting of n points (coordinate pairs), and that Mn^3 processors are available. This approach is different from the one already described in that it makes use of inverse indexing. We next introduce describe the data structure that is central to the inverse indexing approach.

[1]Actually, no more than $Mn(n-1)(n-2)$ processors are needed.

5.1 The Hash-Function Table Structure

The information previously stored in the hash table is now organized as a collection of records stored in a multidimensional table indexed by $(l, basis, k)$, where k designates a point in the model l. In our case, k is an integer between 1 and n. In the table location $(l, basis, k)$ we record the properly quantized hash table coordinates of the k-th point of model l as it is computed in the coordinate frame defined by $basis$. If we let β denote the number of points needed to define a basis ($\beta = 2$ for similarity transformations, $\beta = 3$ for affine transformations), then the new table will be $(\beta + 2)$-dimensional and will contain *at most* one entry in each of its $Mn^{\beta+1}$ table locations. More specifically, the bins whose indices $(l, basis, k)$ are such that point number k occurs in the $basis$ will be empty. Furthermore, all the bins corresponding to inappropriate $basis$ combinations (such as a basis with two identical or very close points) will also be empty. Figure 10 shows the part of the hash-function table structure that

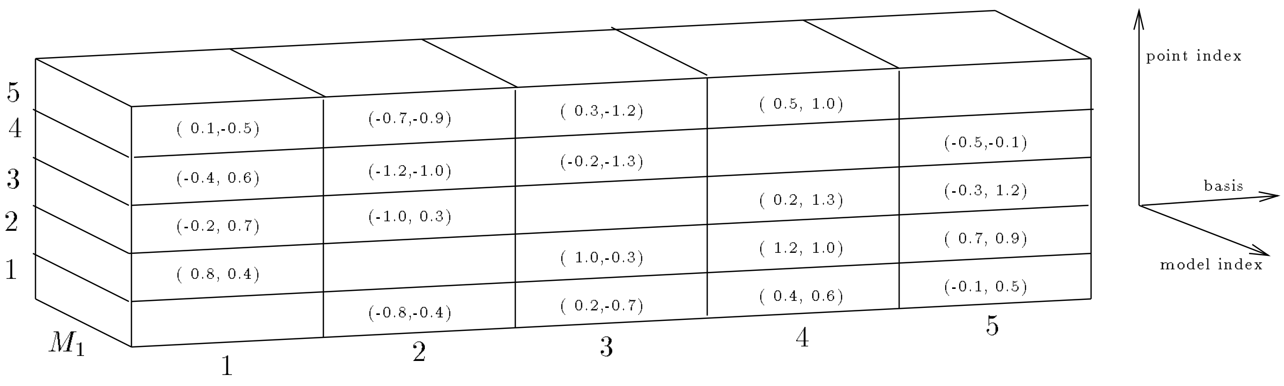

Figure 10. The hash-function table structure: the entries for model M_1. The tuples shown in each box are the coordinates of a model point in the coordinate frame defined by the corresponding (single-point) basis.

corresponds to model M_1 of Figure 1; the same degree of quantization as the one used in the creation of the hash table in Figure 2 is assumed here. In this case, the basis consists of a single point, so there are 5 possible bases, listed as 1 through 5.

5.2 Preprocessing Phase

During the preprocessing phase the hash-function table is constructed. The database contains M models each one having n points. This phase is performed off-line, only once. For the cases of rigid and similarity transformations, the basis tuples are formed using pairs of points and consequently, the hash-function data structure is a four-dimensional table. Three sets of virtual processors participate in this phase, namely the feature coordinates set V_1, the triple product set V_2, and the hash-function table set V_3. Each processor of set V_3 is associated with exactly one table location. This phase consists of three stages, and has parallel time complexity $\Theta(M \log n)$, given Mn^3 processors.

The initial two stages remain the same as in the first algorithm (See Figure 7). We sequentially iterate over the models. The set S_l of the l-th model's point coordinates is initially located in the low-order n processors. The set $(S_l \times S_l) \times S_l$ is then computed; each of the processors containing a triplet $[(x_i, y_i), (x_j, y_j), (x_k, y_k)]$ computes the coordinates of (x_k, y_k) in the coordinate frame the other two points of the triplet define.

During the third and final stage, each of the active processors from the virtual processor set V_2 use the values l, i, j and k to compute the table location in the four-dimensional table where the hash-function value should be stored. A message containing the computed relative coordinates, properly

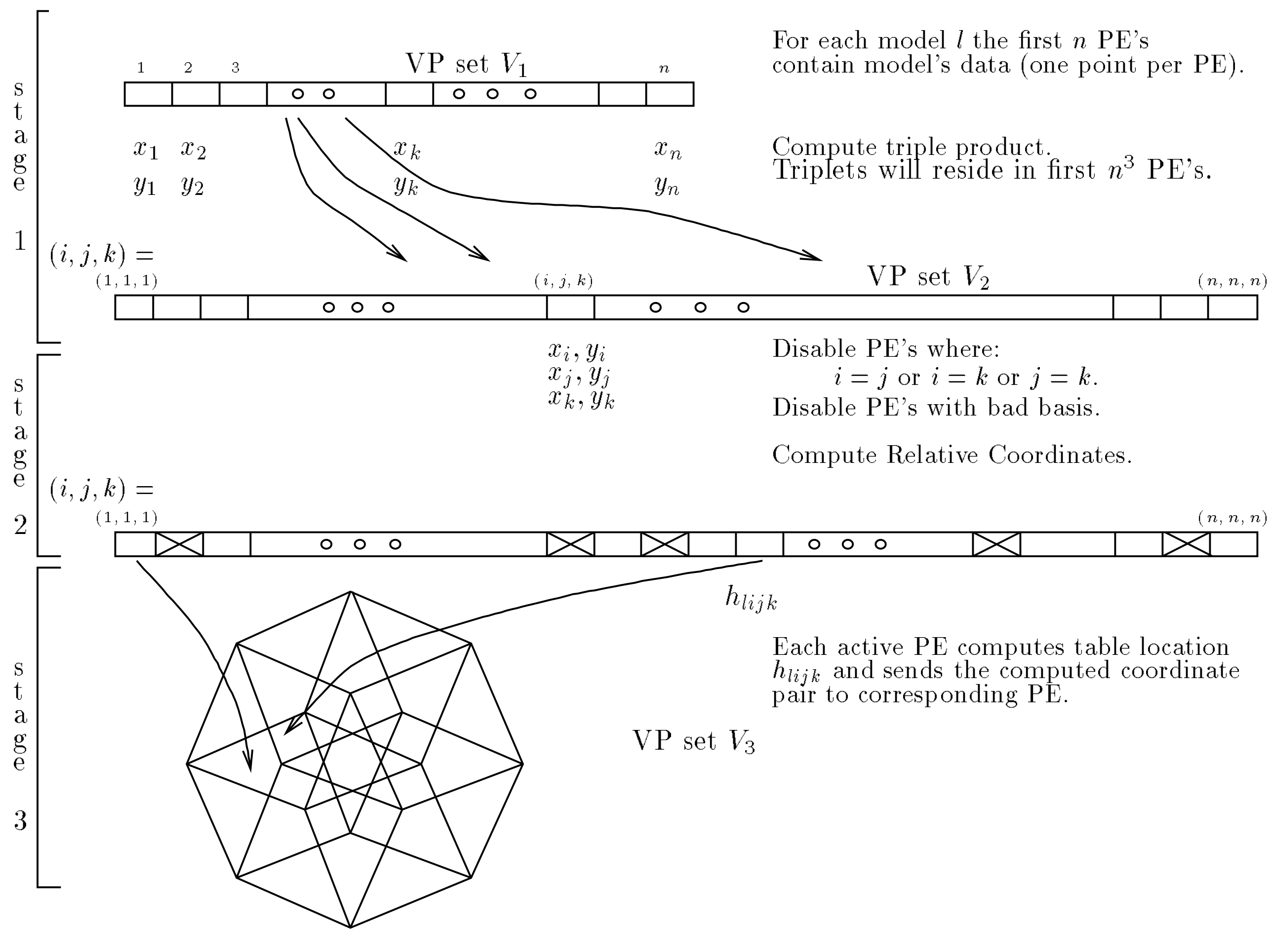

Figure 11. The Preprocessing Phase of a parallel algorithm that constructs the table to be used in the Recognition Phase. The table will have one processor per table model/basis/point combination.

quantized, is then sent to the appropriate processor in the set V_3. No collisions can occur during this step.

The above process is repeated for each of the models in the database. When all of the models have been processed the created hash-function table is stored.

5.3 Recognition Phase

Figure 12 shows a schematic diagram of the recognition algorithm. Two sets of virtual processors participate in this phase, namely V_1, the feature coordinate set, and V_2 the hash-function table set. We again assume that a set of S (S is typically 200) interest points have been extracted from the scene and their coordinates reside in the local memory of the first S processors of virtual processor set V_1. The table is preloaded into the local memory of V_2. In what follows, and for clarity purposes, we assume that $S = n^2$ and that M,n are both powers of 2.

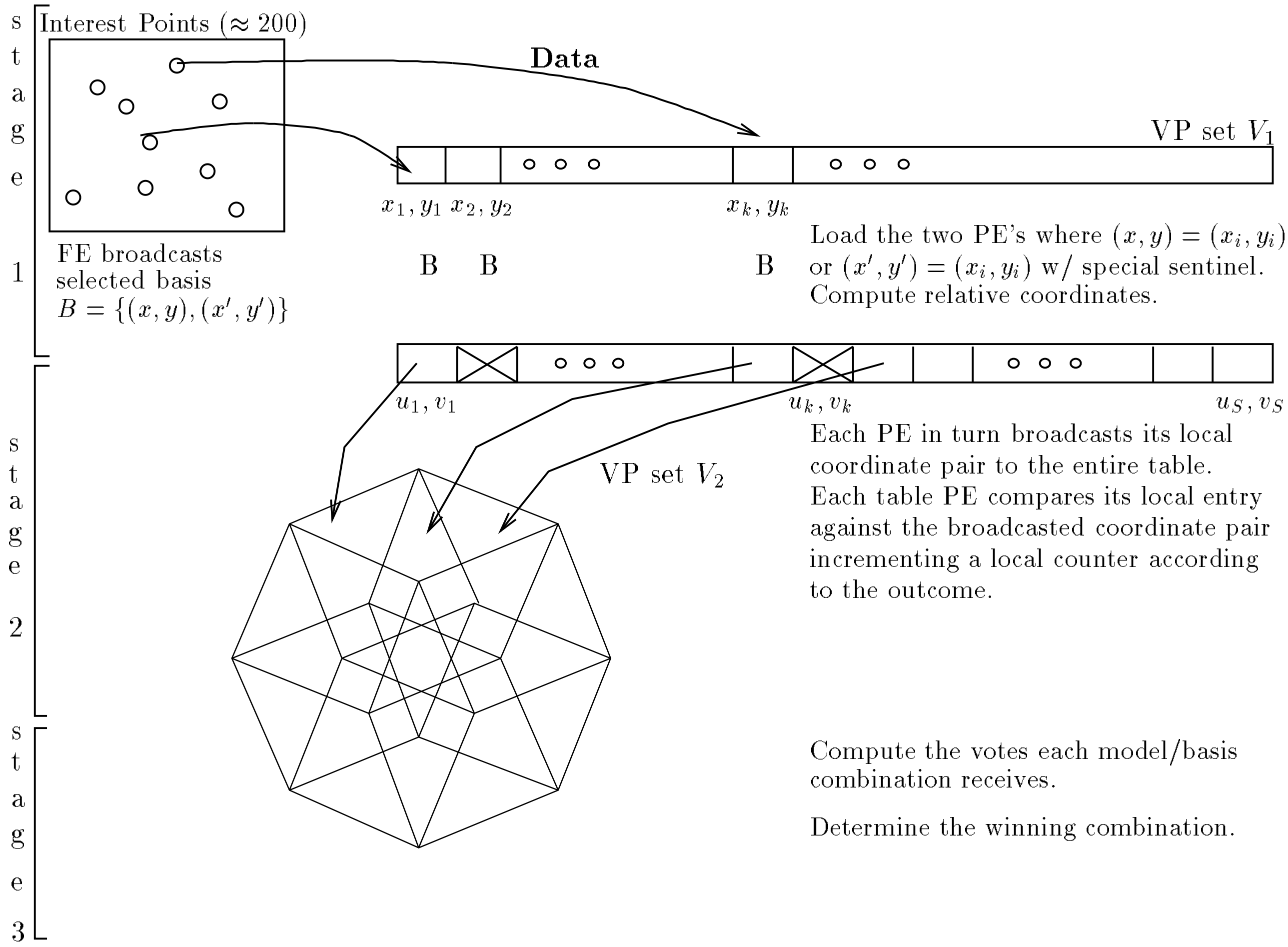

Figure 12. The Recognition Phase of a parallel algorithm that implements geometric hashing.

We operate sequentially through the basis sets and the scene points. First, the front end selects a basis pair and broadcasts the coordinates of the points in the basis to the first 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 use the broadcasted basis to compute the relative coordinates of the locally stored scene point in the coordinate frame of the 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 broadcasted to all the processors of the set V_2. Each broadcasted coordinate has the form (u, v), and gives a location in the hash table where a vote should be tallied. Each processor in V_2, indexed by (m, i, j, k), where (i, j) gives a basis pair, contains a hash location (coordinate pair) which the processor can compare against (u, v). If the two locations are sufficiently close together, then the table location (m, i, j, k) records a hit indicating a vote for model m and basis (i, j). This vote originates from the particular point (among the $S-2$) whose coordinates in the frame of the selected basis are being broadcasted. The tallying of votes continues by accumulating hits in each hash-function table location; each of the $S-2$ broadcasts generates either one or no hits at any table location.

When the tallying is completed, a third stage is invoked; using a parallel prefix sum operation we add the votes over k among locations (m, i, j, k). The result is the total number of votes that the model m and the basis (i, j) obtains for the given scene and basis selection. Finally, a global-max among the processors associated with the locations holding the sum of votes is used to determine the winning model/basis combination. A final verification step may be added to determine the quality of the match.

Strictly speaking, each of the $S-2$ broadcasts will require $\mathcal{O}(\log(Mn^3))$ 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, and assuming that $S <= Mn^3$. As a matter of fact, and 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 $\mathcal{O}(\log(n^2))$. This slice of data can then be spread to the rest of V_2, in parallel slices, requiring no more than $\mathcal{O}(\log Mn)$ 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. Two of the processors of each slice are empty. The processors within a slice can now exchange their data in such a way that the entire list of computed coordinate pairs becomes available to every single one of them. This can be simply achieved by a recursive doubling procedure which communicates data between pairs of processors, and forms lists of coordinate pairs. It must be noted though that the entries of those lists will not necessarily appear in the same order in each processor. This recursive doubling procedure can be completed in $\mathcal{O}(S)$ time.

Let us now consider the time complexity of the recognition phase — the only phase that is performed on-line. It is assumed that Mn^3 processors are available[2]. M is the number of models in the database. Each model consists of n points. S is the number of point features in the input scene.

It is clear that the time complexity of the recognition phase is dominated by the third stage. Indeed, the time complexity of the first stage is $\mathcal{O}(\log S)$. The second stage consists of a number of parallel prefix operations and the stage's time complexity is no worse than $\mathcal{O}(\log(Mn) + \log(n^2))$. The time complexity of the last stage, as was already described, is $\mathcal{O}(S + \log n + \log(Mn))$. This is also the complexity of recognition phase.

6 Implementation Details

In this section we present in detail the actual implementations of the two algorithms already described. Both algorithms are implemented on a 8K-processor Connection Machine using synthetically generated databases.

[2]Actually, no more than $Mn(n-1)(n-2)$ processors are needed

6.1 Data Parallelism over the Hash Table Bin Entries

The algorithm suggested in Section 4 assumes the availability of a total of Mn^3 processors. For the databases that we intend to use, e.g. 1024 models each having 16 points, this translates to at least 2^{22} processors, i.e. many more processors that those currently provided by the available SIMD architectures. One could use the virtual processor facility provided by the software interface to accommodate the processor set V_3; however, the involved overhead would be prohibitive. As a matter of fact, the virtual processor ratio for set V_3 would equal 512 on the 8K-processor Connection Machine.

In order to implement the algorithm, all the $(model, i, j)$ entries that hash into the same bin are stored contiguously in a single physical processor memory rather than associated with distinct physical processors. More specifically, each hash table bin is mapped onto a physical processor; this processor maintains locally a list of all the $(model, i, j)$ entries hashing into the corresponding hash table bin. Clearly, for a given database, some of the local lists could be empty. The collisions that are likely to occur during the preprocessing phase, when more than one processors of set V_2 attempts to deposit the tuple $(model, i, j)$ at the same hash table bin (processor of V_3) are resolved using a simple algorithm based on "locks."

During recognition, the processors corresponding to hash table bins that received one or more votes, scan their local lists and cast a vote for each entry $(model, i, j)$ encountered. The time needed for the list traversal is clearly dominated by the longest such list. As we have already indicated, the current implementation uses method (2) of Section 3.2 in order to histogram the entries of those hash table bins that receive a message. This histogramming process currently accounts for 99% of the execution time of the recognition phase. Clearly, efficiencies in histogramming will very much improve the performance of the implementation. In particular, the use of our radix sort based method of Section 3.2 is expected to reduce processing times, at least for our instances of 18-bit words, and typical sequences of 256K items to be histogrammed. Further improvements in efficiency can be achieved by requantization of the hash space, and use of symmetries.

For a given database of models, the typical hash bin occupancies for uniformly quantized hash bins are non-uniform (Figure 13). Requantization of the hash space leads to hash-bin frequency equalization, thus reducing execution time, and is equivalent to using an appropriate remapping function. For example, in the case where the model points are distributed according to a Gaussian distribution of mean zero and standard deviation σ, the remapping functions are:

$$\text{Rigid Transformations}: \quad \mathbf{h}(u,v) = \left(1 - e^{-\frac{u^2+v^2}{3\sigma^2}},\ \operatorname{atan2}(v,u)\right),\ \textit{and} \tag{1}$$

$$\text{Similarity Transformations}: \quad \mathbf{h}(u,v) = \left(1 - \frac{3}{u^2+v^2+3},\ \operatorname{atan2}(v,u)\right). \tag{2}$$

where atan2(v, u) is the function giving an angular radian measure of the vector formed by (u, v) from the positive horizontal axis. Figure 13 shows the result of applying such a remapping function in the case where the models can undergo similarity transformations.

Recalling from the discussion of Section 2 that the coordinate frame's origin is located at the midpoint between the two points defining the basis, it is easy to see that certain symmetries arise in the storage pattern of entries in the hash table. If $\mathbf{e}_1$ and $\mathbf{e}_2$ form a basis pair of model l, and the coordinates of point $\mathbf{p}$ in model l are (u, v), then there will an entry of the form $(l, (\mathbf{e}_1, \mathbf{e}_2))$ at location (u, v) in the hash table (or at the rehashed position corresponding to (u, v)). When the basis pair ($\mathbf{e}_2$,

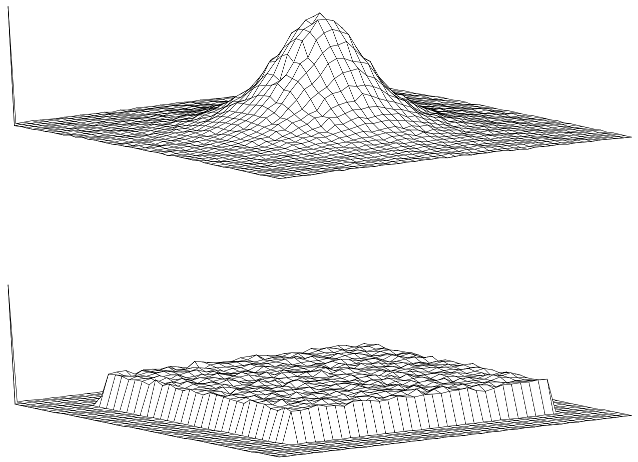

Figure 13. Hash table equalization for similarity transformations. Points are Gaussian distributed over $\mathcal{R}^2$.

$\mathbf{e}_1$) is used, the coordinates of $\mathbf{p}$ will be $(-u, -v)$. Thus in the location $(-u, -v)$ there will be an entry of the form $(l, (\mathbf{e}_2, \mathbf{e}_1))$. Due to the symmetry of the rehashing functions, the location of this entry, even when rehashing is used, will be symmetrically related to the entry of the form $(l, (\mathbf{e}_1, \mathbf{e}_2))$ caused by $\mathbf{p}$ when $(\mathbf{e}_1, \mathbf{e}_2)$ is used as a basis. The result is that for every entry in the hash table (before remapping) above the $v = 0$ axis, there is an equivalent entry in the bottom half plane below the $v = 0$ axis, with the only change that the basis pair has been reversed. Thus, there is no need to store the bottom half of the hash table! Instead, whenever a hash occurs during the recognition phase to the lower half plane, the entries can be generated from the entries in the upper half plane. Since the entries of any given hash bin can now be spread over two bins (processors), the corresponding lists of entries will be half as long.

A final possibility is to perform a *folding* of the hash space. For example, when a hash occurs to the lower half-plane, rather than generating the entries from the list of entries in the symmetrically opposite position of the upper half-plane, we can instead register a vote for the entire bin in the upper half-plane. This will confuse entries of the form $(l, (\mathbf{e}_1, \mathbf{e}_2))$ with entries of the form $(l, (\mathbf{e}_2, \mathbf{e}_1))$ — thus our basis pairs are now basis sets, and $(\mathbf{e}_1, \mathbf{e}_2)$ is the same as $(\mathbf{e}_2, \mathbf{e}_1)$. Although this means that a particular $(model, basis)$ may receive more votes than it deserves, in practice we have encountered no difficulties with this method.

6.2 Data Parallelism over the Model/Basis/Point Combinations

The algorithm suggested in Section 5 assumes the availability of a total of Mn^3 processors. As we already mentioned, for the databases that we intend to use this translates to a number of processors much higher than provided by currently available SIMD machines. Use of the virtual processor facility provided by the software interface, in order to accommodate the employed table is again dismissed due to the extremely high involved overhead.

In order to implement our algorithm, we have chosen to store the entries of the processors in charge of table locations $(l, i, j, k)_{k=1}^{n}$ contiguously in the memory of the processor corresponding to location $(l, i, j, 1)$. This in effect collapses the 4th dimension of the table while at the same time reducing the virtual processor ratio of the corresponding processor set: the new virtual processor ratio is 32 on the 8K-processor Connection Machine. Observe that during the construction of the table, $n-2$ processors attempt to deposit a distinct tuple (coordinate pair) at the same bin. Use of "locks" provides the necessary serialization.

During recognition, the S processors of V_1 rank themselves using either the index of the locally available scene feature or their own address: either approach to performing the ranking involves no data movement and is therefore extremely fast. After having computed the relative coordinates of the local feature in the frame determined by the broadcasted basis, each of the S processors in turn spread their computed coordinates to all of the processors of the table using a parallel prefix computation. After a coordinate tuple is received, each of the table processors, scans its entire local list of coordinate pairs: if the broadcasted tuple matches an entry of the local list, the processor increments the value of a local counter. After all S processors have taken their turn, the values of the local counters contain precisely the number of votes the corresponding $(model, i, j,)$ combination has received for the broadcasted basis. A global-max operation on the values of those counters recovers the winning combination.

Observe that in the current implementation a total of $S(n-2)$ comparisons are performed by each table processor. This is highly inefficient since $S + (n-2)$ comparisons suffice. In fact, the lists of coordinate pairs maintained by each table processor can be sorted during the preprocessing phase. The processors of the set V_1 can rank themselves using the relative coordinates they have computed and use that ranking to determine their turn in spreading the local coordinate pair. When all of the computed coordinate pairs have been communicated to the table processors, each such processor proceeds to determine how many times the entries of its local list occur in the set of computed coordinate tuples: using indirect addressing, only $S+(n-2)$ comparisons are needed. In this alternative the ranking of the S processors of V_1 does involve data movement, hence it is more costly than before. However, we believe that the savings incurred by the decreased number of comparisons will more than counterbalance the cost of ranking.

Finally, observe that the entries of the hash-function table locations (l, i, j, k) and (l, j, i, k) differ only in their signs. We can thus dispose of half of the table! In terms of the actual implementation described above, this results in a reduction of the virtual processor ratio of the corresponding processor set by 2, and thus in an analogous speedup. Of course, it introduces the need for additional bookkeeping but since the required operations involve no data movement this bookkeeping can be performed without any appreciable overhead.

7 Performance Results

The model databases that we use for our experiments contain 1024 models. Each of the models consists of 16 points that are generated randomly. These random points are distributed either uniformly over the unit disc, or according to a Gaussian distribution of mean zero and standard deviation 1. The database models are allowed to undergo either a similarity or a rigid transformation. All of the experiments are carried on an 8K-processor Connection Machine system. Figure 14 shows some typical models (dot patterns).

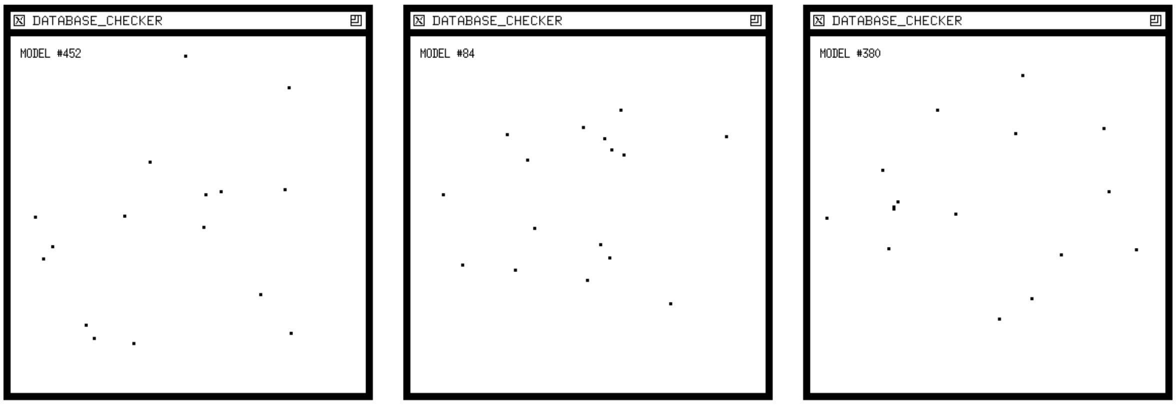

Figure 14. Typical models.

All four combinations were studied:

(1) Rigid Transformations and Gaussian distributed model points;

(2) Similarity Transformations and Gaussian distributed model points;

(3) Rigid Transformation and uniform distribution of the model points over the unit disc; and,

(4) Similarity Transformation and uniform distribution of the model points over the unit disc.

The re-quantization of hash space variants for (1), (2), and (3) above were also studied.

The scenes we use in our experiments are also generated synthetically. In order to create a scene, one of the models is selected from the database; after an arbitrary rotation and translation (and possibly scaling), the model is embedded in a scene of randomly generated points. The total number of points in the final scene is approximately 200; 16 of these points belong to the model. Figures 15 and 16 show typical scenes. Noise is added to the positions of the points (through round-off error). In both implementations, the front end randomly selects a pair of scene points (probe) as the basis to be used during the recognition phase.

For the first implementation, the computation proceeds as outlined in Section 6. For cases (1) through (4) above, the time needed for a single probe during the recognition phase is approximately 3.4 seconds; approximately 7 milliseconds are spent on the computational part of the algorithm and the remainder of the time is devoted to the histogramming process. When we make use of the rehashing functions and of the appropriate uniform hash table, the probe time reduces to 1.2 seconds with 8K processors. It must be stressed here that none of the experiments made use of the hash table symmetries;

exploitation of those symmetries will speed up the recognition by a factor of 2, bringing the time required for a single probe down to 0.60 seconds on an 8K-processor Connection Machine. This is because only half of the hash table must be stored in the Connection Machine, and thus the length of the lists of entries maintained by each processor will be halved. Use of the folding (thereby using basis sets instead of basis pairs) will incur even greater computational savings.

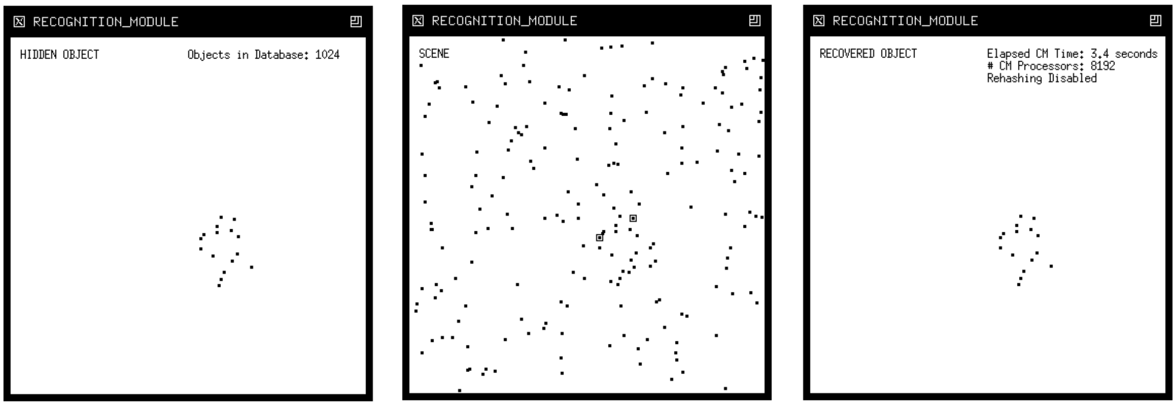

Figure 15. A Recognition Example.

On a 64K machine, the entries of a hash bin can be shared among 8 processors resulting in an 8-fold speedup. Furthermore, additional speedup is introduced by the fact that the virtual processor ratio of the histogram bin set (set V_4) is now 4 (as opposed to 32 with 8K processors): the send-with-additive-write operations will now be much faster. Extrapolations indicate that a 64K machine, using the symmetries and rehashings, but without taking advantage of the foldings, would process the entire computation, including the histogramming, in approximately 70 milliseconds. Additional speedup can be obtained by making use of the folding of the hash table, thereby voting for basis sets as opposed to basis pairs, or by doing partial histogramming.

We have also experimented with the following modification to the $(model, basis)$ items. For databases of 1024 models, the model numbers require 10 bits of information; grouping models into 32 classes, a class of models may be specified with only 5 high-order bits of the model number. Instead of voting for the $(model, basis)$ pair (18-bit data items), we vote for the $(model - class, basis)$ combination, which requires 5 fewer bits. Although this potentially confounds 32 models into a single model, in our experiments, very few items received additional votes, and thus the resulting vote tallies were changed very little. From the $(model - class, basis)$ receiving the most votes, it is very easy to determine the corresponding best model. This modification resulted in probe times times of 0.7 seconds for the recognition phase on an 8K machine (compared to the previous 1.2 second result). On a larger machine, improvements will be similar.

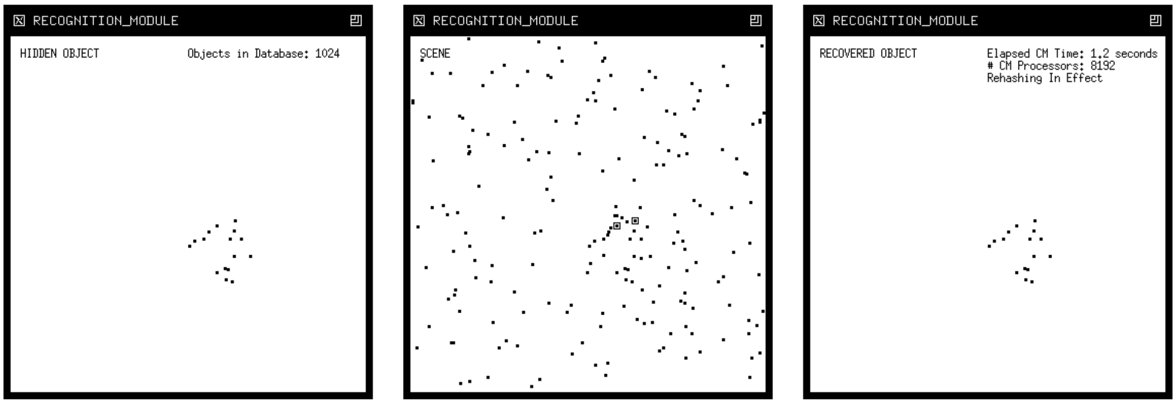

Figure 16. A Recognition Example.

Such model-binning is equivalent to performing an iterated histogramming of the model and basis pairs. At an opposite extreme, the radix sorting method of histogramming should prove to be more efficient than the implementations described here for the range of sizes of the database used in our experiments.

In the second implementation, the computation proceeds as outlined in Section 6. We tested the implementation with a large number of scenes. For cases (1) through (4) above, the implementation processes the scenes at a rate of 5.0 milliseconds per scene point, i.e. for a given probe the implementation requires approximately 1.0 second for a two hundred point scene, on an 8K-processor machine. We expect that with the incorporation of indirect addressing the resulting implementation will outperform that of the first algorithm.

On a 64K-processor machine, the tabular structure will be spread over more processors, thus decreasing the virtual processor ratio of the hash table set by a factor of 8 (i.e. from 32 down to 4), and resulting in similar speedups.

In Figures 15 and 16 we present two recognition examples; both points of the basis belong to the model embedded in the scene.

References

[1] **Bourdon, O. and G. Medioni**. "Object recognition using geometric hashing on the Connection Machine". In *Proceedings of the International Conference on Patern Recognition*, pages 596–600, Atlantic City, June 1990.

[2] **Chin, R. and C. Dyer**. "Model-based recognition in robot vision". *ACM Computing Surveys*, 18(1):67–108, March 1986.

[3] **Hummel, R. and H. Wolfson**. "Affine invariant matching". In *Proceedings of the DARPA Image Understanding Workshop*, April 1988.

[4] **Huttenlocher, D. P. and S. Ullman**. "Object recognition by affine invariant matching". In *Proceedings of the DARPA Image Understanding Workshop*, April 1988.

[5] **Lamdan, Y. and H. Wolfson**. "Geometric hashing: A general and efficient Model-Based recognition scheme". In *Proceedings of the 2nd International Conference on Computer Vision*, pages 238–249, 1988.

[6] **Lamdan, Y. and H. Wolfson**. "On the error analysis of geometric hashing". Technical Report 213, Robotics Lab., New York University, October 1989.

[7] **Lamdan, Y. and J. Schwartz and H. Wolfson**. "Object recognition by affine invariant matching". In *Proceedings of the IEEE Computer Vision and Pattern Recognition Conference*, pages 335–344, Ann Arbor, Michigan, June 1988.

[8] **Lin, W-M. and V.K. Prasanna Kumar**. "Efficient histogramming on hypercube SIMD machines". *Computer Vision, Graphics, and Image Processing*, 49:104–120, 1990.

[9] **Little, J., G. E. Blelloch, and T. A. Cass**. "Algorithmic techniques for computer vision on a Fine-Grained parallel machine". *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(3), March 1989.

[10] **Lowe, D. G.** *Perceptual Organization and Visual Recognition.* Kluwer Academic Publishers, 1985.

[11] **Nassimi, D. and S. Sahni**. "Data broadcasting in SIMD computers". *IEEE Transactions on Computers*, C-30:101–107, 1981.

[12] **Press, W. H., B. Flannery, S. A. Teukolsky and W. T. Vetterling**. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 1988.

[13] **Stenstrom, J. R.. and A. J. Heller**. "Verification of recognition and alignment hypothesis by means of edge verification statistics". In *Proceedings of the DARPA Image Understanding Workshop*, pages 957–966, April 1988.