11-1990

Parallel Vision Algorithms Using Sparse Array Representations

Ravi V. Shankar
Syracuse University

SU-CIS-90-37

Parallel Vision Algorithms Using Sparse Array Representations

Ravi V. Shankar

November 1990

School of Computer and Information Science
Syracuse University
Suite 4-116, Center for Science and Technology
Syracuse, NY 13244-4100

SU-CIS-90-37

Parallel Vision Algorithms Using Sparse Array Representations

Ravi V. Shankar*

November 1990

School of Computer and Information Science
Syracuse University
Suite 4-116
Center for Science and Technology
Syracuse, NY 13244-4100

(315) 443-2368

* To be presented at the International Symposium on Intelligent Robotics, Bangalore, India, January 1991.

Parallel Vision Algorithms Using Sparse Array Representations

Ravi V Shankar *
School of Computer and Information Science
Syracuse University, Syracuse, NY 13244-4100

Abstract: Sparse arrays are arrays in which the number of non-zero elements is a small fraction of the total number of array elements. This paper presents computer vision algorithms using sparse representations for arrays. The parallel architecture considered is a hypercube. The algorithms can be easily modified for other architectures like the mesh. We assume that the architecture is SIMD, i.e., all PEs work under the control of a single control unit.

1 Introduction

Consider a simple operation (such as the elementwise summation of two arrays) on two m_0N_0 x m_1N_1 x...x $m_{k-1}N_{k-1}$ arrays. If this operation is to be carried out in an N_0 x N_1 x...x N_{k-1} grid of processing elements, each processing element holds m_0 x m_1 x...x m_{k-1} elements. Hence, the operation would have to go through that many iterations.

Sparse arrays of different dimensions occur various forms in vision applications:

- Image processing : Images with a large number of points having the same intensity are sparse arrays in two dimensions.
- Pose clustering : During object recognition matching image and scene feature-pairs compute a tuple which describes the transformation between them. This tuple is used to cast votes in a high-dimensional accumulator (Number of dimensions is three when dealing with 2D object recognition, or six when dealing with 3D object recognition).
- Hough Transform : While detecting arbitrary curves or straight lines in an image, image points cast votes in a high-dimensional parameter space (Number of dimensions is two, three, or four when dealing with line detection, circle detection, and ellipse detection, respectively).

This paper is organized as follows. Section 2 describes the parallel primitives needed. The Content Access Read/Write primitives are developed as generalizations of the Random Access Read/Write primitives. Section 3 briefly describes the representations of sparse arrays in the processing elements. Section 4 describes the Neighbor Sum algorithm. The next two sections present a histogramming algorithm and its application in two vision problems - Pose Clustering and feature detection.

* To be presented at the International Symposium on Intelligent Robotics, Bangalore, India, January 1991.

2 Description of primitives used

2.1 Random Access Read(RAR)/Random Access Write(RAW)

In a RAR, some processing elements need to read data from some of the N PEs in the hypercube. The data is available in register D. Each PE has the address from which data is needed in the register P. That is, PE i needs D(P(i)). If PE i does not need data from any PE then P(i) is set to ∞. The RAR algorithm [1] has a time complexity of $O(\log N(\log\log N)^2)$. Figure 1 shows an example of a RAR.

PE index	0	1	2	3	4	5	6	7
Pointer P	6	2	.	.	1	.	3	.
After RAR	D(6)	D(2)	-	-	D(1)	-	D(3)	-

Figure 1. RAR example

In a RAW some PEs need to write their data to one of the N PEs in the hypercube. The data is available in register D. Each PE has,in register P, the address to which it needs to send its data. If PE i does not send any data P(i) is set to ∞. The RAW algorithm [1], like its RAR counterpart, has a time complexity of $O(\log N(\log\log N)^2)$. Unlike the RAW case, however, it is possible to have collisions. This happens when two PEs try to write to the same PE. When collisions are bound to occur, one of two things can be done, viz., reporting an error, or combining the colliding data values using a binary associative operator. Figure 1 shows an example of a RAW. Collisions are resolved using a binary associative operator.

PE index	0	1	2	3	4	5	6	7
Pointer P	7	2	.	7	1	.	3	.
After RAW	-	D(4)	D(1)	D(6)	-	-	-	D(0)*D(3)

Figure 2. RAW example

2.2 Content Access Read(CAR)/Content Access Write(CAW)

CAR/CAW are generalized forms of the RAR/RAW operations. The generalization, however, comes at no increase in time complexity.

In a CAR each PE needs some piece of data, but, unlike the RAR case, it does not know exactly where to get the data from. It does, however, know the *contents* of some particular register in the source PE. The contents of that register may not be unique in each PE. Figure 3 shows an example of a CAR. We use the same names for the registers as in the RAR case. The contents of the P register are no longer pointers to other PEs. Instead they contain values from the PA (short for 'pointed at') register. Any binary associative operator can be used to combine the results in case *read-from-multiple-PEs*.

PE index	0	1	2	3	4	5	6	7
Pointer P	6	2	.	.	2	.	3	.
Pointed-at PA	7	6	6	1	1	2	3	8
After CAR	D(1)+D(2)	D(5)	-	-	D(5)	-	D(6)	-

Figure 3. CAR example

In a CAW, each PE needs to write some piece of data, but, does not have an explicit pointer to the destination PE. It does, however, know the *contents* of some particular register in the destination PE. Figure 4 shows an example of a CAW. Collisions can occur as in the RAW case. Any binary associative operator can be used to resolve collisions when more than one value is written to the same PE.

PE index	0	1	2	3	4	5	6	7
Pointer P	6	2	.	.	2	.	3	.
Pointed-at PA	7	6	6	1	1	2	3	8
After CAW	-	D(0)	D(0)	-	-	D(1)*D(4)	D(6)	-

Figure 4. CAW example

3 Representation of Sparse Arrays

In our representation for a sparse array, each processing element holds two pieces of information about a non-zero array element - its value, and its address (a concatenation of the k indices of that element). Memory requirement per processor is therefore $\log M + \log N$, where M is the range of values the array elements can take, and N is the total number of array elements.

Figure 5a shows an 8x8 32-gray level image. Figure 5b shows the compact representation of the same.

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	16	7	0	0	0	0	0
23	0	2	29	0	0	0	0
0	5	19	0	17	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 5(a) An 8x8 gray level image

address	17	18	24	26	27	33	34	36
value	16	7	23	2	29	5	19	17

Figure 5(b) Compact representation of image in (a)

4 Neighbor Sum

The *Neighbor Sum* operation is used to collect data values from neighboring array elements and sum them up. Procedure *NeighborSum* gives details. The procedure makes use of the Content Access Read algorithm. The complexity of the CAR algorithm is $O(\log N(\log\log N)^2)$ and hence the complexity of NeighborSum is $O(\log N(\log\log N)^2)$ when the dimensionality k of the data array is a constant. It should be observed that the operation described here is restricted since the sum is calculated only by the non-zero elements of the array.

```
Procedure NeighborSum(A,S,k);
{A accumulates the sum of the data in the S registers of the neighboring PEs}
{S is the sparse representation of the given data array of size m0 x m1 x .. x mk-1}
{Register Pos contains the index of each PE }
{ CAR(D,P,PA) performs a Content Access Read without collisions. D,P,PA are the data, pointer,
and pointed-at registers in the CAR algorithm}
1  begin
2    A := 0;
3    for p := 0 to k-1 do
4      begin
5        N := Pos + ∏_{i=0}^{p-1} m_i;
6        X := CAR(S,N,Pos);
7        A := A + X;
8        N := Pos - ∏_{i=0}^{p-1} m_i;
9        X := CAR(S,N,Pos);
10       A := A + X;
11     end;
12 end; of NeighborSum
```

Program 6 Computation of the sum of neighboring values

The above algorithm can be modified to perform other operations by replacing summation by maximum/minimum finding, boolean OR/AND etc.

5 Histogramming

Consider an array A whose elements have values in the range $[0,M)$. The histogram of A is another array H such that H[i] equals the number of elements in A that have the value i. We will visualize histogramming as a voting process in which A represents the array of voters and H represents the array of candidates. It is generally assumed that M is much smaller than the number of PEs N in the hypercube. We will assume, to the contrary, that M is much larger than N. The last assumption is not unusual in problems in intermediate-level vision where the elements of A are actually obtained as the result of a mathematical computation and could even be floating point numbers. A sample situation where this could occur is described at the end of this section. In such cases, memory requirements dictate that the array H be stored as a sparse array. Histogramming is implemented as a Content Access Write with collisions.

The histogramming problem occurs in intermediate-level computer vision as the *Pose clustering* problem. The scenario is the following: a large set of scene features is available one per processor, and a small set of model features is available in the control unit. Each processor, when given a model and scene feature pair, can compute the transformation between them in constant time. This transform is in general a k-tuple ($k = t + r$ where $t = 2$, $r = 1$ for 2D object recognition, and, $r = 3, t = 3$ for 3D object recognition). These k-tuples form the new set of voters. The voting space is a k-dimensional array called the transform-space.

The transform-space for 3D object recognition (k=6) would need 100 x 100 x 100 x 180 x 180 x 180 = 5832 billion processors, when a moderately fine resolution of 1 unit and 1 degree is used for translations and rotations respectively. The high dimensionality of the transform-space array and the limitation on the number of processors available forces us to adopt a sparse representation for the same.

Figure 7 shows how the Histogramming algorithm can be used for Pose clustering.

Do steps 1 through 5 for each model feature
Step 1 : Broadcast model feature to all PEs
Step 2 : Compute in each PE the transform between the model and scene feature to the desired precision.
Step 3 : Perform histogramming using earlier algorithm to get the sparse transform array H.
Step 4 : Compact the transform array H
Step 5 : Merge H with the transform arrays obtained in earlier iterations and compact the resulting array if necessary.

Figure 7 Steps in the Pose Clustering Algorithm

The broadcast in step 1 takes O($\log N$) time on a hypercube. The computation in step 2 is independent of the number of PEs and hence its time complexity is O(1). Step 3 takes O($\log N(\log\log N)^2$) time. The compact in step 4 takes O($\log N$) time and so does the merge in step 5. Thus the Pose clustering algorithm has a complexity of O($m * \log N(\log\log N)^2$), where m is the number of model features.

As mentioned earlier the advantage of this algorithm is that the precision with which the computed transform is represented is longer governed by the resolution with which the transform-space can be represented. It is even possible to have floating-point computed transforms, if the transform computations are sufficiently accurate.

6 Feature detection

We consider the line detection example here. Extensions to circle/ellipse detection are trivial. In the line detection case, we deal with two spaces - the 2D image-space and the 2D parameter-space. Each point (x,y) in the image-space votes for all points (ρ,θ) in the parameter-space that satisfy the equation $\rho = xcos\theta + ysin\theta$.

The feature detection case, differs from the histogramming technique seen earlier, in that each voter can now cast more than one vote. This forces us to iterate over $p-1$ parameters, where p is the dimensionality of the parameter-space (The last parameter can be determined by simple substitution). We cannot assume that the parameter-space is sparse now, since each image point can vote for many parameter-space points. However, we make use of the fact that no parameter-space point receives votes in more than one iteration to efficiently discard uninteresting points from the parameter-space.

Local maxima in the parameter-space indicate lines in the image. These local maxima can be detected using a simple modification of the neighbor sum algorithm.

7 Conclusion

This paper describes the use of sparse array representations to parallel vision algorithms. The resulting reduction in the number of processing elements required comes at no increase in time complexity. This was shown by the development of the Content Access Read/Write primitives which are generalized forms of the Random Access Read/Write primitives having the same time complexity.

8 Appendix

8.1 Other hypercube primitives

This section describes 3 basic primitives used by the CAR/CAW algorithms. Algorithms for these primitives can be found in [2].

8.1.1 Merge

Merging of two sorted arrays can be done on the hypercube using the bitonic merge algorithm. The merge algorithm takes time O(log N) where N is the number of PEs.

8.1.2 Segmented Scans

In the segmented prefix scan algorithm a 1-bit register S is used to indicate the start of a new segment when set to 1. Data is available in register D. A binary associative operator + is specified. The objective is to obtain in PE i the quantity D(j) + D(j+1) + ... D(i) where j satisfies the following properties (i) j <= i (ii) S(j) = 1 and (iii) for all k satisfying $j < k <= i$ and S(k)=0.

The segmented prefix scan algorithm is a modified form of the prefix scan algorithm. Figure 8 illustrates the scan primitive. The time complexity of the algorithm is O(log N).

PE index	0	1	2	3	4	5	6	7
Data	7	3	4	2	0	1	5	1
After prefix +-scan	7	10	14	16	16	17	22	23
Segment register S	1	1	0	1	0	0	0	1
After segmented +-scan	7	3	7	2	2	3	8	1

Figure 8. Prefix/Segmented prefix scan example

8.1.3 Sort

Bitonic sort can be used to sort an array on the hypercube. The bitonic sort algorithm takes time O(log^2 N). A recent result [3] gives a deterministic hypercube sorting algorithm that has a complexity of O(log N(log log N)^2).

8.2 Algorithms for CAR/CAW

The CAR algorithm is described through an example (figure 8). Here the number of PEs $N = 8$, while P(0:7) = (6 2 ∞ ∞ 2 ∞ 3 ∞), PA(0:7) = (7 6 6 1 1 2 3 8). As mentioned earlier PE i needs to fetch data from all PEs in which register PA has the same value as P(i). P(i) = ∞ iff PE i is to receive no data. The data to be read is available in register D. We begin by sorting the contents of the P and i registers using P as the key and i to resolve ties. Next we sort the contents of the PA and data registers using PA as the key and i to resolve ties. After tagging the P and PA registers (a tag of 1 indicates a P value while a tag of 0 indicates a PA value), the contents of the P and PA registers are merged together in register M. This results in alternate segments of values from P and PA registers, which we will refer to as P segments and PA segments. These segments are further divided such that the M values are identical throughout each segment. A segmented scan on the data values is now done in the PA segments alone. The binary associative operator specified for collisions is used as the scan operator. The M value in the last PE in each PA segment is compared with the M value in the first PE in the P segment adjacent to it. If the values are identical, the result of the scan in the last PE in the PA segment is

copied to the first PE in the adjacent P segment. A segmented copy scan in the P segments makes the result available to the rest of the PEs in the P segments. A sort (with the index $i2$ as the key) in all registers with a tag of 1 gives the required result.

The CAW algorithm is illustrated in figure 9. The algorithm is similar to the previous one. The P registers are now tagged 1 and the PA registers 0. The data and P registers are now sorted together with PA as the key, while the i and PA registers are sorted with P as the key. In both cases i is used to resolve collisions. The merge, segment creation, binary associative operator scan, and copy scan proceed exactly as before. A final sort gives the required result.

References

[1] D. Nassimi and S. Sahni, 'Data Broadcasting in SIMD Computers', *IEEE Transactions on Computers*, Vol C-30, No.2, 1981, pp.101-107.

[2] S. Ranka and S. Sahni, *Hypercube algorithms for Image Processing and Pattern Recognition*, Springer Verlag, 1990.

[3] G. Plaxton and R. Cypher, 'Deterministic Sorting in nearly logarithmic time', *Proc. ACM Sympsium on Theory of Computing*, 1990, pp.193-203.

i	0	1	2	3	4	5	6	7
P	6	2	x	x	2	x	3	x
PA	7	6	6	1	1	2	3	8
P1	2	2	3	6	x	x	x	x
i1	1	4	6	0	2	3	5	7
tag	1	1	1	1	1	1	1	1
PA1	1	1	2	3	6	6	7	8
data	D(3)	D(4)	D(5)	D(6)	D(1)	D(2)	D(0)	D(7)
tag	0	0	0	0	0	0	0	0

After merge:

M	1	1	2	2	2	3	3	6	6	6	7	8	x	x	x	x
tag	0	0	0	1	1	0	1	0	0	1	0	0	1	1	1	1
data	D(3)	D(4)	D(5)	–	–	D(6)	–	D(1)	D(2)	–	D(0)	D(7)	–	–	–	–
i2	–	–	–	1	4	–	6	–	–	0	–	–	2	3	5	7

After segmented +–scan in PEs with tag 0 (+ could be any binary associative operator):

M	1	1	2	2	2	3	3	6	6	6	7	8	x	x	x	x
tag	0	0	0	1	1	0	1	0	0	1	0	0	1	1	1	1
scan1	D(3)	D(3) +D(4)	D(5)	–	–	D(6)	–	D(1)	D(1) +D(2)	–	D(0)	D(7)	–	–	–	–
i2	–	–	–	1	4	–	6	–	–	0	–	–	2	3	5	7

After segmented copy–scan in PEs with tag 1:

M	1	1	2	2	2	3	3	6	6	6	7	8	x	x	x	x
tag	0	0	0	1	1	0	1	0	0	1	0	0	1	1	1	1
scan1	D(3)	D(3) +D(4)	D(5)	–	–	D(6)	–	D(1)	D(1) +D(2)	–	D(0)	D(7)	–	–	–	–
scan2	–	–	–	D(5)	D(5)	–	D(6)	–	–	D(1) +D(2)	–	–	–	–	–	–
i2	–	–	–	1	4	–	6	–	–	0	–	–	2	3	5	7

After sort:

i3	0	1	2	3	4	5	6	7
car	D(1)+D(2)	D(5)	–	–	D(5)	–	D(6)	–

Figure 9. CAR (Content Access Read) example

i	0	1	2	3	4	5	6	7
P	6	2	x	x	2	x	3	x
PA	7	6	6	1	1	2	3	8

P1	2	2	3	6	x	x	x	x
data	D(1)	D(4)	D(6)	D(0)	x	x	x	x
tag	0	0	0	0	0	0	0	0

PA1	1	1	2	3	6	6	7	8
i1	3	4	5	6	1	2	0	7
tag	1	1	1	1	1	1	1	1

After merge:

M	1	1	2	2	2	3	3	6	6	6	7	8	x	x	x	x
tag	1	1	0	0	1	0	1	0	1	1	1	1	0	0	0	0
data	–	–	D(1)	D(4)	–	D(6)	–	D(0)	–	–	–	–	x	x	x	x
i2	3	4	–	–	5	–	6	–	1	2	0	7	–	–	–	–

After segmented +–scan in PEs with tag 0 (+ could be any binary associative operator):

M	1	1	2	2	2	3	3	6	6	6	7	8	x	x	x	x
tag	1	1	0	0	1	0	1	0	1	1	1	1	0	0	0	0
scan1	–	–	D(1)	D(1) +D(4)	–	D(6)	–	D(0)	–	–	–	–	x	x	x	x
i2	3	4	–	–	5	–	6	–	1	2	0	7	–	–	–	–

After segmented copy–scan in PEs with tag 1:

M	1	1	2	2	2	3	3	6	6	6	7	8	x	x	x	x
tag	1	1	0	0	1	0	1	0	1	1	1	1	0	0	0	0
scan1	–	–	D(1)	D(1) +D(4)	–	D(6)	–	D(0)	–	–	–	–	x	x	x	x
scan2	–	–	–	–	D(1) +D(4)	–	D(6)	–	D(0)	D(0)	–	–	–	–	–	–
i2	3	4	–	–	5	–	6	–	1	2	0	7	–	–	–	–

After sort:

i3	0	1	2	3	4	5	6	7
caw	–	D(0)	D(0)	–	–	D(1)+D(4)	D(6)	–

Figure 10. CAW (Content Access Write) example