Pattern Recognition, Vol. 26, No. 10, pp. 1511–1519, 1993
Printed in Great Britain

0031-3203/93 $6.00 + .00
Pergamon Press Ltd

PARALLEL VISION ALGORITHMS USING SPARSE ARRAY REPRESENTATIONS

RAVI V. SHANKAR and SANJAY RANKA†
School of Computer and Information Science, Syracuse University, Syracuse, NY 13244-4100, U.S.A.

(*Received* 14 *July* 1992; *in revised form* 8 *February* 1993; *received for publication* 7 *April* 1993)

Abstract—Sparse arrays are arrays in which the number of non-zero elements is a small fraction of the total number of array elements. Parallel algorithms are presented using sparse representations for arrays. It is shown that adopting such a representation not only reduces the processor/space requirement, but also provides efficient load balancing at no increase in time complexity. New parallel primitives needed to work with such a representation are defined. Sample algorithms from the areas of image processing and computer vision are presented. Alternative schemes for dealing with arrays containing large contiguous blocks of elements with identical array values are considered. The parallel architecture considered is a strict SIMD hypercube, and the applicability of the results presented to other architectures is described.

Computer vision Sparse array representations Parallel processing Hypercube algorithms

1. INTRODUCTION

Sparse arrays are arrays in which the number of non-zero elements is a small fraction of the total number of array elements. A sparse image is a sparse two-dimensional (2D) array, where the array entries denote image intensity values. Sparse arrays of different dimensionalities occur in a variety of forms in different areas. Here are some examples from the areas of image processing and computer vision:

- *Image processing.* Images with a large number of points having the same intensity (resulting, for instance, from a smooth background) are sparse arrays in two dimensions.
- *Pose clustering.* During object recognition, matching image and scene feature-pairs compute a tuple which describes the transformation that takes the scene feature to the image feature. This tuple is used to cast votes in a high-dimensional accumulator (number of dimensions in three when dealing with 2D object recognition, and six when dealing with 3D object recognition). This high-dimensional accumulator contains a signficant number of entries that are zero.
- *Hough transform.* While detecting arbitrary curves or straight lines in an image, image points cast votes in a high-dimensional parameter space. This high-dimensional parameter space is likely to be sparse. The number of dimensions is two when dealing with line detection, since a line can be described by two parameters, for instance, by its slope and its y-intercept. Circle detection needs a 3D parameter space, two parameters for position and one for radius. Detecting an ellipse in general position and orientation needs a 5D parameter space, two parameters for position, two for the major and minor axes, and one for ellipse orientation.
- *Low and intermediate level vision.* The inputs to many low/intermediate level vision algorithms such as thinning, feature-based stereo matching, surface fitting, etc. could be sparse images.

Consider a simple operation (such as the elementwise summation of two arrays) on two $m_0N_0 \times m_1N_1 \times \cdots \times m_{k-1}N_{k-1}$ arrays. If this operation is to be carried out in a set of $N_0 \times N_1 \times \cdots \times N_{k-1}$ processing elements, each processing element holds $m_0 \times m_1 \times \cdots \times m_{k-1}$ elements. Hence, the operation would have to go through that many iterations. If the input arrays are sparse, such a solution would perform very poorly. In this paper we develop a linear representation of multi-dimensional sparse arrays and present parallel primitives and algorithms that work on this representation. These primitives and algorithms provide efficient load balancing. It is shown that adopting such a sparse array representation reduces the processor/space requirement while maintaining the same asymptotic time complexity.

1.1. *Parallel architecture considered*

The algorithms described in this paper can be applied to a wide variety of parallel distributed memory architectures. The algorithms assume a strict[1] hypercube architecture for illustration purposes (a strict hypercube computer is a hypercube with weak, uniform communication, i.e. each processor is limited to sending a single data item over a single communication link at a time, and, at any given time, every processor must send data along a communication link in the same hypercube dimension). We also assume that the architecture is SIMD, i.e. all processing elements (PEs) work under the control of a single control unit. These restrictions (strict hypercube, SIMD) enable us to write algorithms and present worst-case complexity results that hold true for all other kinds of hypercubes. Further,

† Author to whom all correspondence should be addressed.

the shuffle-exchange architecture,[2] the cube-connected cycles architecture,[3] the de Bruijn architecture, and the butterfly architecture are closely related to the strict hypercube architecture (see reference (1) for details), and hence all results presented are true for these four architectures also. The algorithms could be modified for a mesh architecture, but the complexity results stated (and the conclusions based on them) are not true for the mesh.

1.2. *Embedding grids onto hypercubes*

As an extreme case, a d-dimensional hypercube is a d-dimensional grid of size 2 in every dimension. A hypercube with N processing elements can also be viewed as a k-dimensional (binary-encoded) grid, for any k lying between 0 and $\log N$, where the node $(i_{k-1}, i_{k-2}, \ldots, i_1, i_0)$ on the grid corresponds to the node *concatenate* $(i_{k-1} i_{k-2} \cdots i_1 i_0)$ on the hypercube.

The neighbors of any node on a 2D grid can be defined as follows. The direct neighbors of the grid node with index (i_1, i_0) are the four nodes with indices $(i_1 \pm 1, i_0)$ and $(i_1, i_0 \pm 1)$. The indirect neighbors of the grid node (i_1, i_0) are the four nodes with indices $(i_1 + 1, i_0 + 1), (i_1 + 1, i_0 - 1), (i_1 - 1, i_0 + 1)$ and $(i_1 - 1, i_0 - 1)$. This definition can be generalized to deal with higher-dimensional grids.

This paper is organized as follows. Section 2 describes the parallel primitives needed for working with sparse array representations. The Content Access Read/Write primitives are developed as generalizations of the Random Access Read/Write primitives, and the Old-Neighbor Read/Write primitives as generalizations of the Neighbor Read/Write primitives. Section 3 describes ways of representing sparse arrays in the processing elements. Section 4 discusses algorithms using regular local operations on sparse arrays. The next two sections present a histogramming algorithm and its application to two vision problems—pose clustering and feature detection. Section 7 gives the algorithms for the Content Access Read/Write primitives.

2. DESCRIPTION OF PRIMITIVES USED

2.1. *Random Access Read/Random Access Write*

In a Random Access Read (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 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 ∞ (∞ is shown as a . in the figures). The RAR algorithm[2] 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)	–

Fig. 1. RAR example.

In a Random Access Write (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,[2] 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, namely, reporting an error, or combining the colliding data values using a binary associative operator. Figure 2 shows an example of a RAW. Collisions are resolved using a binary associative operator (shown as a + in the figure).

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

Content Access Read and Write are basically the RAR and RAW operations, generalized to work with sparse array representations. The generalization, however, comes at no increase in time complexity. The two primitives are defined here, and the algorithms for implementing them are presented in Section 7.

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 when data has to be read from multiple PEs.

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	7	2	.	7	1	.	3	.
After RAW	–	D(4)	D(1)	D(6)	–	–	–	D(0) + D(3)

Fig. 2. RAW example.

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)	–

Fig. 3. CAR example.

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)	–

Fig. 4. CAW example.

2.3. *Neighbor Read/Neighbor Write*

In a Neighbor Read (NR), every PE needs to read a value from a particular neighbor. Many PEs would qualify as neighbors to a given PE, depending on how the hypercube is being viewed. All PEs are restricted to read from the same neighbor. Under these conditions, the NR algorithm turns out to be a restricted case of the RAR. Since the addresses of the source PEs are not random, sorting is not required. As a result the time complexity of the NR algorithm is only $O(\log N)$.

In a Neighbor Write (NW), each PE writes to a particular neighbor. All PEs are restricted to write to the same neighbor. Since the destination PE addresses are not random, this is a restricted case of the RAW algorithm. The time complexity of the NW algorithm is $O(\log N)$. Unlike the RAW, the NW cannot have collisions.

2.4. *Old-Neighbor Read/Old-Neighbor Write*

The Old-Neighbor Read (ONR) algorithm is similar to the NR, with the exception that the original neighbor of a pixel might have migrated to a different location carrying its index along with itself. The ONR algorithm is a restricted case of the CAR. Sorting of destination addresses is not required, and there cannot be any collisions. The time complexity of the ONR algorithm is $O(\log N)$.

In an Old-Neighbor Write (ONW), each PE writes to a previous neighbor. This is a restricted CAW, and the complexity is $O(\log N)$.

3. REPRESENTATION OF SPARSE ARRAYS

3.1. *The Concentrate primitive*

In the Concentrate algorithm we start with a subset of the processing elements, each containing data in register D, and the PE's rank (that is, the number of selected PEs with lower index than self) in register R. The objective is to move the data in register D such that D(i) goes to the PE with index R(i).

The Concentrate algorithm is described in reference (4). Figure 5 illustrates the concentrate operation. The time complexity of the algorithm is $O(\log N)$ where N is the size of the given input and is equal to the number of PEs. If instead only P PEs are available ($P < N$) then the concentrate algorithm takes $O((N/P)\log P)$ time. However, in the case of special hypercube architectures like the pipelined hypercube[5] the complexity of concentrate improves to $O((N/P) + \log P)$.

3.2. *Concentrated representation of sparse arrays*

The Concentrated representation is the simplest representation for a sparse array. In this representation, 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 the total number of array elements.

Row-major indexing and shuffled row-major indexing are two ways of indexing pixels in a 2D grid. These two indexing schemes are shown in Fig. 6. Figure 7(a) shows an 8×8 32-gray level image. Figures 7(b) and (c) show the concentrated representation of the same using row-major/shuffled row-major indexing for the image pixels.

3.3. *Quadtree based representations*

A quadtree is a tree representation of a sparse image (in general, any 2D array). The root of the quadtree represents the entire image. If the portion of the image represented by any node does not have the same gray value, the node is assigned four children. Each child

D	–	D(1)	–	–	D(4)	D(5)	D(6)	–
R	.	0	.	.	1	2	3	.
D (after Concentrate)	D(1)	D(4)	D(5)	D(6)	–	–	–	–

Fig. 5. The Concentrate primitive.

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

(a)

0	1	4	5	16	17	20	21
2	3	6	7	18	19	22	23
8	9	12	13	24	25	28	29
10	11	14	15	26	27	30	31
32	33	36	37	48	49	52	53
34	35	38	39	50	51	54	55
40	41	44	45	56	57	60	61
42	43	46	47	58	59	62	63

(b)

Fig. 6. (a) Row-Major and (b) Shuffled Row-Major Indexing for an 8 × 8 image.

represents one of the four quadrants of the image portion represented by its parent. This continues recursively until all the leaf nodes represent portions of the image with the same gray value.

If a binary image contains large blocks of 1s and 0s, a quadtree representation of the image would be preferable to the concentrated representation since the processing can be done using blocks of pixels, rather than single pixels, as units. A binary image and its quadtree representation are shown in Fig. 8. Hypercube algorithms for operations on quadtrees are presented in reference (6).

Extending the quadtree representation to gray level images can be done in one of two ways. If the number of non-zero pixels in the image is very small compared to the total number of image pixels, the technique described for binary images can be used. An extra register would be needed to store the pixel value at all points associated with each block. If, instead, the image has regions with almost the same value, a region growing technique can be used to split the image into homogeneous regions, and the processing can be done using regions as units.

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

Fig. 7(a). An 8 × 8 gray level image.

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

Fig. 7(b). Concentrated representation of image in Fig. 7(a) using Row-Major Indexing.

SRM address	9	10	12	14	15	33	36	48
value	16	23	7	2	29	5	19	17

Fig. 7(c). Concentrated representation of image in Fig. 7(a) using Shuffled Row-Major Indexing.

The notion of quadtree-based representations described above is relevant only to images or 2D arrays. In the general case, when dealing with d-dimensional arrays, we can extend the idea to 2^d trees (trees in which every node has either zero or 2^d children). For instance, when $d = 3$, we talk about 8-trees or octrees.

4. REGULAR LOCAL OPERATIONS

Regular local operations are characteristic of many image-processing and vision algorithms. Some of these involve the computation of the result of applying a binary associative operator like +, max, min, or boolean OR/AND to neighboring elements. Other operations such as convolving, template matching, smoothing, and sobel edge detection involve the computation of a

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	1	0	0	0	1	0
1	1	0	0	0	0	0	0
0	0	1	1	1	1	1	1
0	0	1	1	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1

Fig. 8(a). An 8 × 8 binary image.

SRM index	8	12	28	36	48
Level	2	3	3	2	1
value	1	1	1	1	1

Fig. 8(b). A quadtree-based representation of the image in Fig. 8(a).

sum of products using a weighted window. These algorithms can be written as repeated calls to the Neighbor Read/Write primitives when using the original (non-sparse) array representations, or to the Old-Neighbor Read/Write primitives when using sparse array representations.

For instance, the NeighborSum algorithm is needed to collect data values from neighboring array elements and sum them up. Procedure NeighborSum in Fig. 9 gives details. The procedure makes use of the ONR algorithm. An outer for loop iterates once for each dimension in the original array. Row-major indexing is used to index elements in the original array. Therefore, the addresses of a particular element's neighbors in dimension p are obtained by adding/subtracting $m_0 \times m_1 \times \cdots \times m_{p-1}$ to the address of that element. The complexity of the ONR algorithm is $O(\log N)$ and hence the complexity of NeighborSum is $O(\log N)$ when the dimensionality d 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.

5. HISTOGRAMMING BASED ALGORITHMS

5.1. *Histogramming using sort and count*

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. Histogramming can be done by first sorting the contents of array A and then counting the number of elements with the same values using a segmented prefix scan.[7] The time taken by the sorting dominates the time complexity of this algorithm which is $O(\log N(\log\log N)^2)$. An alternate way of implementing histogramming is by using a radix sort. This has a complexity of $O(\log M * \log N)$ on an N PE hypercube where M is the number of gray levels.

It is generally assumed that the number of candidates M is much smaller than the number of voters N. We will assume, to the contrary, that M could be much larger than N, i.e. the number of candidates could be higher than the number of voters. Although the last assumption sounds ridiculous in real-life situations, it is not unusual in problems where the elements of A are obtained as high precision numbers (or floating-point numbers) following a mathematical computation. A sample situation where this could occur is in Pose clustering in intermediate-level vision. Pose clustering is described later in this section. When the number of possible values that the elements of array A can take is very high, memory limitations dictate that the array H be stored as a sparse array. Histogramming can still be implemented as a sort and count algorithm.

5.2. *Pose clustering*

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, $r = 3, t = 3$

```
Procedure NeighborSum(A, S, d);
{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 × m1 × ··· × m(d-1)}
{Register Pos contains the original index of the pixel held by each PE}
{ONR(D, P, PA) performs an Old-Neighbor Read. D, P, PA are the data, the neighbor address, and the old
address registers}
1  begin
2    A:= 0;
3    for p:= 0 to d − 1 do
4      begin
5        N:= Pos + ∏(i=0 to p−1) m_i;
6        X:= ONR(S, N, Pos);
7        A:= A + X;
8        N:= Pos − ∏(i=1 to p−1) m_i;
9        X:= ONR(S, N, Pos);
10       A:= A + X;
11     end;
12 end; {of NeighborSum}
```

Fig. 9. Computation of the sum of neighboring values.

Do steps 1–5 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, after discarding model and scene feature pairs which do not match.
- Step 3. Perform histogramming to get the sparse transform array H.
- Step. 4. Concentrate the transform array H.
- Step 5. Merge H with the transform arrays obtained in earlier iterations and concentrate the resulting array if necessary.

Fig. 10. Steps in the Pose Clustering Algorithm.

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.

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

Figure 10 shows how the Histogramming algorithm can be used for Pose clustering. 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 $O(\log N \times (\log \log N)^2)$ time. The concentrate 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 $O(m * \log N \times (\log \log N)^2)$, 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 no 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.

Continuing our analogy with the voting procedure, Pose clustering merely allows each voter to cast multiple votes, unlike the earlier histogramming case in which each voter had just one vote. This complicated the problem since it is not possible to decide on a winner until the entire voting is over. However, by establishing criteria that a voter has a satisfy to cast each vote, ineligible voters can be barred from casting some votes.

6. FEATURE DETECTION USING THE HOUGH TRANSFORM

The Hough transform is a robust technique that is used to detect analytic curves in an image. We consider the detection of lines here. The idea can be extended to circle/ellipse detection.

In the line detection case, we deal with two spaces—the 2D image-space and the 2D parameter-space. To detect lines in an image, an edge detector is first used to find edge points. Each edge point (x, y) in the image-space votes for all points (ρ, θ) in the parameter-space that represent the parameters of lines passing through the edge point (x, y). These are precisely the parameter-space points that satisfy the equation $\rho = x \cos \theta + y \sin \theta$. Local maxima in the parameter-space, which accumulates the votes, indicate lines in the image. These local maxima can be detected using a simple modification of the neighbor sum algorithm.

An iteration over one of the two parameters (say θ) is necessary during voting. We can no longer assume that the parameter-space is sparse, since each image point can vote for many parameter-space points. In each iteration, only parameter-space points with a certain value of θ can receive votes. The feature detection case differs from the simple histogramming seen earlier

Do steps 1–5 for each value of θ

Step 1. Broadcast θ to all PEs.

Step 2. Compute, in each PE representing an edge point (x, y), the value of ρ that satisfies the equation $\rho = x \cos \theta + y \sin \theta$.

Step 3. Perform histogramming using a sort and count algorithm to get the parameter-space array V showing the number of votes received.

Step 4. Concentrate the parameter-space array V after removing parameter-space points that received too few votes.

Step 5. Merge V with the parameter-space arrays obtained in earlier iterations, and concentrate the resulting array if necessary.

Step 6. Perform local maxima detection in the parameter-space array V to detect lines in the image.

Fig. 11. Steps in the Line Detection Algorithm.

in that each voter can now cast more than one vote. It also differs from the Pose clustering algorithm because no candidate can receive votes in more than one iteration. By fixing the minimum number of votes a candidate needs, many of the candidates can be eliminated even before the voting process gets completed.

The line detection algorithm is outlined in Fig. 11. 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 concentrate in step 4 takes $O(\log N)$ time and so does the merge in step 5 and the local maxima detection in step 6. Thus the line detection algorithm has a complexity of $O(m*\log N(\log\log N)^2)$, where m is the number of values θ was quantized into.

The number of dimensions in the parameter-space increases with the complexity of the feature we are trying to detect. The number of dimensions is three and five when dealing with circle detection and ellipse detection, respectively. When the parameter-space is p-dimensional, the algorithm is forced to iterate over $p-1$ parameters (the last parameter can be determined by substitution).

7. ALGORITHMS FOR CAR/CAW

The CAR algorithm is described through an example (Fig. 12). Here the number of PEs $N = 8$ (the PE index i runs from 0 to 7), while $P(0:7) = (6\,2\,\infty\,\infty\,2\,\infty\,3\,\infty)$, $PA(0:7) = (7\ 6\ 6\ 1\ 1\ 2\ 3\ 8)$. As mentioned earlier PE i needs to fetch data from all PEs in which register PA has the same value as $P(i)$. $P(i) = \infty$ 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. The sorted contents are shown in the rows marked P1 and $i1$. Next we sort the contents of the PA and data registers using PA as the key and i to resolve ties. The sorted contents are shown in the rows marked PA1 and data. 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. The row labeled $i2$

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)	-

Fig. 12. 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)    -
```

Fig. 13. CAW (Content Access Write) example.

shows the original PE indices after the merge. 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 Fig. 13. The algorithm is similar to the previous one. The P registers are now tagged 0 and the PA registers 1. The data and P registers are now sorted together with P as the key, while the i and PA registers are sorted with PA 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.

8. CONCLUSION

This paper described the use of sparse array representations in parallel computations. Sample computations from the areas of vision and image processing were presented. The resulting reduction in the number of processing elements and space required came 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 that work with sparse array representations. Two other primitives, Old-Neighbor Read/Write, were also presented to simplify neighbor communication when using sparse array representations.

Acknowledgements—The authors would like to thank the referees for their comments on this paper. The work of Sanjay Ranka was supported in part by NSF under CCR-9110812.

REFERENCES

1. R. Cypher, Efficient communication in massively parallel computers, Ph.D. Thesis, Department of Computer Science, University of Washington (1989).
2. D. Nassimi and S. Sahni, Data broadcasting in SIMD computers, *IEEE Trans. Comput.* **C-30**(2), 101–107 (1981).
3. F. P. Preparata and J. Vuillemin, The cube-connected cycles: a versatile network for parallel computation, *CACM* **24**(5), 300–309 (1981).
4. S. Ranka and S. Sahni, *Hypercube Algorithms for Image Processing and Pattern Recognition.* Springer, Berlin (1990).
5. J. Jaja and K. W. Ryu, Load balancing and routing on the hypercube and related networks, *J. Parallel Distributed Computing* **14**(4), 431–435 (1992).
6. R. Shankar and S. Ranka, Hypercube algorithms for operations on quadtrees, 6th Distributed Memory Computing Conference, April–May (1991).
7. J. J. Little, G. Blelloch and T. A. Cass, Algorithmic techniques for computer vision on a fine-grained parallel machine, *IEEE Trans. Pattern Analysis Mach. Intell.* **11**(3), 244–257 (1989).
8. G. Plaxton and R. Cypher, Deterministic sorting in nearly logarithmic time, *Proc. ACM Symp. on Theory of Computing*, pp. 193–203 (1990).

APPENDIX

A.1. *Other hypercube primitives*

This section describes three basic primitives used by the CAR/CAW algorithms. Algorithms for these primitives can be found in reference (4).

A.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.

A.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 $D(j) + D(j+1) + \cdots + D(i)$ where j satisfies the following properties (i) $j \leq i$, (ii) $S(j) = 1$ and (iii) for all k satisfying $j < k \leq i$ and $S(k) = 0$.

The segmented prefix scan algorithm is a modified form of the prefix scan algorithm. Figure A1 illustrates the scan primitive. The time complexity of the algorithm is $O(\log N)$.

A.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[8] gives a deterministic hypercube sorting algorithm that has a complexity of $O(\log N(\log\log N)^2)$.

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

Fig. A1. Prefix/Segmented prefix scan example.

About the Author—RAVI V. SHANKAR is a doctoral candidate in computer science at Syracuse University. He received his B.E. in computer science and engineering from Anna University, Madras, India, in 1987. He was nominated Syracuse University Fellow for the academic years 1987–90. He was a Visiting Scientist at the Siemens Center for Research and Development in Munich, Germany, from July 1991 to December 1991. His research interests include computer vision, image processing, and parallel computing.

About the Author—SANJAY RANKA received his B.Tech. in computer science and engineering from the Indian Institute of Technology, Kanpur, in 1985 and Ph.D. in computer and information science from the University of Minnesota, Minneapolis, in 1988. Currently, he is an Assistant Professor in the School of Computer Science at Syracuse University. He spent the summer of 1991 as an Academic Visitor at IBM T. J. Watson Research Center. His current areas of interest are parallel algorithms, models of parallel computation, compilers and software environments for parallel machines, and neural networks. Professor Ranka has co-authored over 70 journal and conference papers, three book chapters and a book on *Hypercube Algorithms for Pattern Recognition and Image Processing.* He is currently a subject area editor (Algorithms and Scientific Computing) for *Journal of Parallel and Distributed Computing.* He was also a co-guest editor of a special issue (February 1992) of *IEEE Computer* on parallel processing for computer vision and image understanding.