Parallel Processing of Linear Quadtrees for Robotics

X. Y. Zhou and W. A. Davis

Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1

ABSTRACT

Quadtrees are important data structures for the representation of two dimensional workspaces. An important advantage of a quadtree is that the structure conveniently lends itself to applications such as path planning, visual navigation, obstacle avoidance, spatial indexing, and landmark and object recognition. This paper explores a parallel adaptation scheme for linear quadtrees on an unsorted dictionary machine implementable in VLSI. Many algorithms including neighbor finding, labeling connected components, and template matching are developed. Performance analysis shows that the proposed scheme provides an improvement in efficiency. As a result, the new scheme is quite suitable for real time applications such as robotics.

Key Words: Binary tree machines, dictionary machines, pipelining, priority queues, quadtrees, linear quadtrees, robotics, search machines, systolic systems, VLSI algorithms, VLSI complexity.

1. Introduction

Two dimensional region representation has found many applications in the area of robot vision such as path planning, visual navigation, obstacle avoidance, spatial indexing, and object recognition. One of the important data structures for region representation is called a quadtree [3-5]. The quadtree representation is based on the principle of recursive decomposition of an image. Its hierarchical nature makes it relatively compact, and well-suited to operations such as complement, union and intersection, as well as determining various other region properties. Recent developments on quadtree region representations has led to the concept of linear quadtrees [3-5]. Most commonly used algorithms for neighbor finding, labeling connected components, and performing various operations such as union, intersection, rotation and translation have been proposed in the literature. Performance analysis of the various algorithms shows that linear quadtrees provide a improvement in efficiency for region representation in terms of both computational complexity and space requirements.

In real applications, however, the speed of various processing algorithms crucially affects the effectiveness of a robot vision system. In many applications sequential algorithms are too slow to enable the system to respond in real time. With the advent of multiprocessor systems, efficient parallel algorithms and data structures to meet the requirement of real time robot vision are being increasingly realized [1,6]. In this paper, a parallel adaptation for linear quadtree on an unsorted dictionary machine is presented.

2. Quadtrees and Linear Quadtrees

An image of size $M=2^n \times 2^n$ is defined by a 2^n by 2^n array of unit square pixels, where n is called the resolution of the image. An image is called a binary image if its pixels assume values of only 1 or 0. A pixel is BLACK if it has the value of 1, otherwise it is WHITE. Without loss of generality, only binary images will be considered in this paper.

A pixel array is inefficient in terms of space and time complexity. The Quadtree representation is based on the successive subdivision of the image array into four equal sized quadrants. If the array does not consist entirely of 1's or entirely of 0's, it is then subdivided into quadrants, subquadrants, etc. until blocks are obtained that consist entirely of either 0's or 1's. A quadtree is an ordered tree of degree four. The root represents the entire image, each other child node represents one of four subblocks obtained by the subdivision process. Furthermore, terminal nodes are colored BLACK or WHITE while nonterminal nodes are colored GREY. Figure 1 illustrates a quadtree representation.

The distinct feature of linear quadtrees is that they are pointerless and store only BLACK nodes. To capture the recursive decomposition process, a node in a linear quadtree is encoded by a p-tuple, $<K, r_1, \ldots, r_{p-1}>$, where, K is defined to be SH(x,y), where <x,y> are the coordinates of the left bottom pixel of the node, SH is a bit shuffling operation; $r_1, \ldots, r_{p-1}$ are properties of the node, in particular, r_1 is defined to be the resolution parameter of the node. For instance, the corresponding linear quadtree of the quadtree in Figure 1 is: {<3, 0>, <8, 1>, <14, 0>, <15, 0>, <32, 0>, <33, 0>, <35, 0>, <36, 1>, <44, 1>, <48, 1>, <56, 1>}.

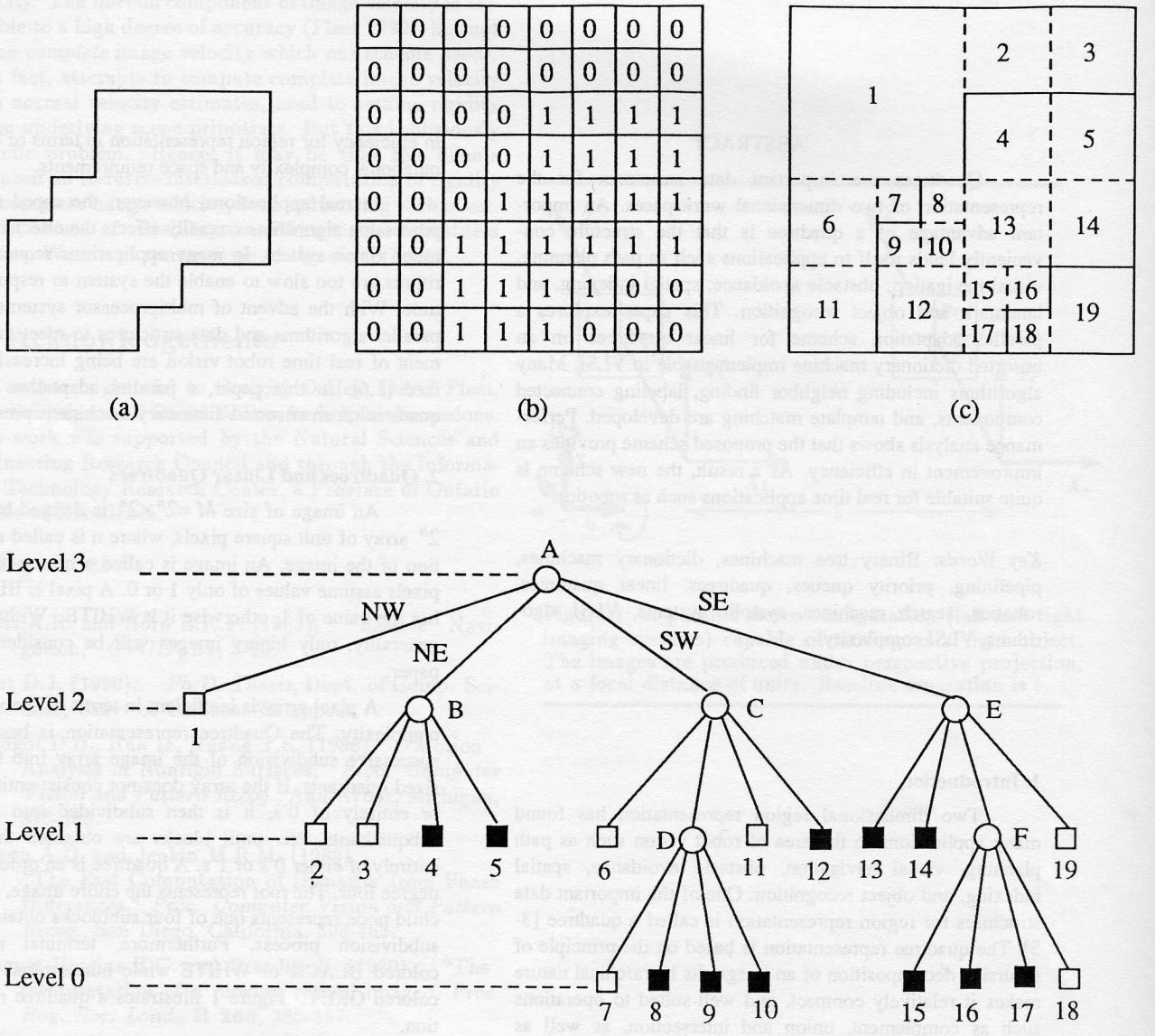

Figure 1. An example (a) region, (b) its binary array, (c) its maximal blocks, and (d) the corresponding quadtree.

3. An Unsorted VLSI Dictionary Machine

The dictionary and priority queue data structures are very important structures in applications such as sorting, searching, symbol-table, and decision-table implementations. Such a data structure consists of a set S of data elements, each of which is composed of an n-tuple $K=(k, r_1, \ldots, r_{p-1})$, where k is the key used for searching the data elements K, and $r_1, \ldots, r_{p-1}$ are the associated properties. Several researchers have explored hardware implementations of dictionary machines [1,2,9]. The hardware implementation of these structures is characterized by a strong, inherent parallelism functioning in a systolic manner, thus yielding a much higher performance than a conventional Von Neumann implementation. In the following, without loss of generality, an unsorted dictionary machine proposed in [9] will be described.

The machine is structured as a pure binary tree where each node of the tree is a processing element. Each node is capable of storing an element and can communicate with its father and two sons. No relationship is maintained between the elements stored at the various nodes in the tree in terms of the values of their keys. However, the elements are stored at the lowest possible level, and the structure of the tree is dynamically balanced at each node with respect to the number of elements stored in the left and right subtrees of the node. This is achieved by maintaining a CNT register at each node which maintains the difference of the number of elements stored in two subtrees. In addition, it is assumed that input and output ports are associated with the root node. To keep track of the number of elements stored in the machine at any time, a COUNT register is maintained at the root node. A tree with 12 elements from k1 to k12, inserted in that order, is shown in Fig. 2.

As a tree architecture, the dictionary machine naturally emulates a pipelined execution of instructions: a sequence $(I_1, I_2, \ldots)$ of instructions enters the root node of the machine. Then this stream travels down the tree, level by level, and in this process, each node produces two copies (sometimes slightly modified) of this instruction for each of its two sons. At the last level nodes, the two copies of instruction are not sent further down. On the other hand, upon receiving the instruction, a processing element executes the instruction and generates a response. The response is then sent further down to last level of the tree. Moreover, at each last level node, the downward stream gets reflected to become upstream in the next instruction cycle. Finally, upstream responses are merged level by level as they propagate to the root.

A. K. Somani and V. K. Agarwal show that the machine is able to support *Insert(K)*, *Delete(K)*, *Member(K)*, *Minimum*, *Maximum*, *Near(k)*, and *No-operation* with response time $O(logN)$, where N is the number of keys present, and with pipelined interval $O(1)$[1].

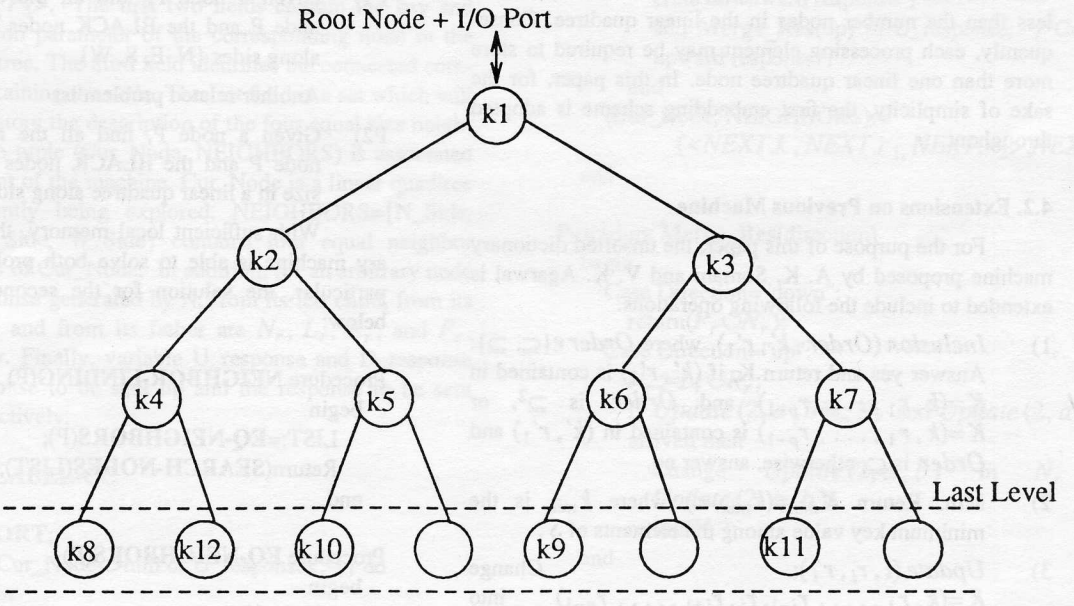

Figure 2. Unsorted Dictionary Machine with 12 Elements.

[1] Interested readers are referred to [9] for the detailed implementation.

4. A Generalized Dictionary Machine for Linear Quadtrees

In this section, a scheme for embedding linear quadtrees on the above mentioned machine is presented. Then the machine is generalized to process linear quadtree operations in addition to dictionary operations. In particular, a number of fundamental operations for linear quadtrees have been emulated in the context of the new machine. Finally, a scheme for embedding linear quadtrees in the generalized machine is presented.

4.1. Embedding Linear Quadtrees on Dictionary Machine

Linear quadtrees can be embedded in the dictionary machine in two ways: complete embedding and block embedding. In the first approach, a node encoded as $<k, r_1, \ldots, r_{p-1}>$ corresponding to an element in the machine. Therefore, procedure *Insert*(K) can be employed to distribute the data items in a linear quadtree to a dictionary machine [9]. To avoid costly unshuffling operation [4], it is recommended that coordinates <x,y> be explicitly stored with each item. For a linear quadtree with N nodes, it can be shown that the time complexity for data distribution is $O(N)$. The second approach deals with the situation where the number of processing elements in the machine is less than the number nodes in the linear quadtree. Consequently, each processing element may be required to store more than one linear quadtree node. In this paper, for the sake of simplicity, the first embedding scheme is adopted throughout.

4.2. Extensions on Previous Machine

For the purpose of this paper, the unsorted dictionary machine proposed by A. K. Somani and V. K. Agarwal is extended to include the following operations:

1) *Inclusion* $(Order, k', r'_1)$, where $Order \in \{\subseteq, \supseteq\}$: Answer yes and return K, if (k', r'_1) is contained in $K=(k, r_1, \ldots, r_{p-1})$ and $Order$ is $\supseteq$[2], or $K=(k, r_1, \ldots, r_{p-1})$ is contained in (k', r'_1) and $Order$ is $\subseteq$, otherwise, answer no.

2) Min: Return $K_{min}=(k_{min}, r)$ where k_{min} is the minimum key value among the elements of S.

3) *Update* (i, r_i, r'_i): Change $K=(k, r_1, \ldots, r_{i-1}, r_i, r_{i+1}, \ldots, r_{p-1})$ into $K=(k, r_1, \ldots, r_{i-1}, r'_i, r_{i+1}, \ldots, r_{p-1})$, otherwise, do nothing.

4) Next (k): Return $K'=(k', r_1, \ldots, r_{p-1})$ such that $k'>k$ and $k'-k$ is the minimum over all elements of S. If there is no such element in set S, then answer null.

[2] In other words, the condition $r_1 \geq r'_1$ and $k \leq k' \leq k+4^{**}r_1$ is satisfied, see [4,5].

It is easy to show that the above extension and modification can be easily implemented without sacrificing the performance of the various operations.

5. Parallel Algorithms for Manipulating Linear Quadtrees

In this section, a number of linear quadtree algorithms have been emulated in the context of the generalized machine. In particular, an algorithm for neighbor finding is developed in Section 5.1. As a direct application of the algorithm in a parallel environment, a technique for labeling connected components is described in Section 5.2. Algorithms for computing various geometric properties of a region including calculating the area, centroid, and perimeter of a region are developed in Section 5.3. Finally, an algorithm for the template matching problem is presented in Section 5.4, where the importance of the proposed approach is further demonstrated.

5.1. Neighbor Finding

Neighbor finding is a cornerstone for many operations such as labeling connected components, computing perimeters, and others. The problem can be stated as follows:

P1) Given a node P, find all the adjacencies between node P and the BLACK nodes in a linear quadtree along sides {N, E, S, W}.

Another related problem is:

P2) Given a node P, find all the adjacencies between node P and the BLACK nodes of greater or equal size in a linear quadtree along sides {N, E, S, W}.

With sufficient local memory, the proposed dictionary machine is able to solve both problems efficiently. In particular, the solution for the second problem is given below:

```
Procedure NEIGHBOR-FINDING(P)
  begin
    LIST:=EQ-NEIGHBORS(P);
    Return(SEARCH-NODES(LIST);
  end

Procedure EQ-NEIGHBORS(P)
  begin
    LIST:={};
    for k ε {N,E,S,W} do
      Add EQ-NEIGHBOR(P,k) to LIST[3];
    Return (LIST);
  end

Procedure SEARCH-NODES(NEIGHBORS)
  begin
    LIST:={};
    for <k', r'1>ε NEIGHBORS do
```

```
    add Inclusion (⊇, k', r'1) into LIST;
  return(LIST);
end
```

The algorithm first computes the set of all equal size neighbors at the root. Then instruction *Inclusion* $(\supseteq, k', r'_1)$ is broadcast to the processing elements in a pipelined fashion to explore the adjacencies between node P and BLACK nodes in the linear quadtree. The computation complexity of **EQ-NEIGHBOR** is $O(n)$ [4,5], whereas the computation complexity of **SEARCH-NODES** is $O(logN)$. Since $logN < 2n$, the time complexity of the entire algorithm is $O(n)$. Moreover, if <x,y> coordinates are explicitly stored in each node, then the time complexity of the algorithm reduces to $O(logN)$.

5.2. Labeling Connected Components

Connected component labeling is a fundamental operation in image processing. Many connected component labeling algorithms have been successfully applied on binary images as a first step towards object recognition. In this section, a novel approach for labeling all connected components using a dictionary machine is presented.

Let each processing element in the machine that consist of a local memory be represented as quadruple $<k, r_1, r_2, r_3>$. The first two fields contain the key and the resolution parameter of the corresponding node in the linear quadtree. The third field identifies the connected component containing the node. The last field is a set which will be used to store the description of the four equal size neighbors. A two tuple (Cur_Node, NEIGHBORS) is associated with the root of the machine. Cur_Node is a linear quadtree node currently being explored. NEIGHBORS={N_Side, E_Side, S_Side, W_Side} contains four equal neighbor descriptions of Cur_Node. In addition, for an arbitrary node N, the response generated by N, from its left child, from its right child, and from its father are N_r, L_r, R_r, and F_r, respectively. Finally, variable U_response and D_response is the response to be sent up and the response to be sent down respectively.

```
Procedure LABEL-CC
  begin
    INI-PORT;
    while Cur_Node<>null or U_response<>{} do
      begin
        SEARCH(Cur_Node,NEIGHBORS);
        UPDATE_ID(U_response)
      end
  end
Procedure INI-PORT
  begin
    Broadcast EQ-NEIGHBORS to all PEs;
    K.r3 = EQ-NEIGHBORS(K);   /*parallel execution*/
    K.r2 = ID-Generator;      /*parallel execution*/
    (Cur_Node, NEIGHBORS):=
          (<Min.k, Min.r1, Min.r2,>, Min.r3);
  end

Procedure UPDATE_ID(U_response)
  begin
    if U_response<>{} then return;
    for Update (2, a, b) ε U_response do
      execute Update (2, a, b);
  end

Procedure SEARCH(Var Cur_Node, Var NEIGHBORS)
  begin
    if Cur_Node=null then return;
    U_response:=D_response:={};
    for K ε SEARCH-NODES(NEIGHBORS) do
      begin
        Nr:={};
        if Cur_Node.r2 <> K.r2 then
          Nr:='Update (2, K.r2, Cur_Node.r2)';
          /*Id of node K need to be updated*/
        add Merge_Res(down) to D_response; /*Generate downward response*/
        add Merge_Res(up) to U_response; /*Generate upward response*/
      end
    (Cur_Node, NEIGHBORS):=
          (<NEXT.k, NEXT.r1, NEXT.r2>,NEXT.r3)
  end

Procedure Merge_Res(direction)
  begin
    Case Direction="down":
      return(Fr∪Nr);
    Case Direction="up":
      Nr:=Lr∪Rr;
      If Update (2, a, b)⊆Nr and Update (2, a, c) is
      arrived then
        Change Update (2, a, b) in Nr into
        Update (2, c, b);
      return(Nr);
  end
```

Procedure **INI-PORT** first broadcasts the instruction stream in the procedure **EQ-NEIGHBORS** to the processing elements in the machine, so that the computation of **EQ-NEIGHBORS** for each node can be performed in parallel. Function **ID-Generator** assigns different identifications to different nodes (i.e., the binary representation of the path from the root to the node).

After initialization, the algorithm processes the linear quadtree nodes in ascending key order and all operations are

[3] Procedure **EQ-NEIGHBOR**(P, S) gives the description of the neighbor of P of equal size in a direction specified by side S, see [4,5].

pipelined. For each node, Cur_Node, being processed, the adjacencies between Cur_Node and the nodes in the linear quadtree of greater or equal size are explored. Depending on the configuration of the region under consideration, an adjacent node Q may have assigned a label different from that of Cur_Node, in which case all the nodes having the same label as Q are requested to change labels so that they will be assigned the same label as that of Cur_Node. Therefore, a response: "*Update* $(2, K.r_2, Cur\_Node.r_2)$" is generated. The request is then sent down so that it is reflected on the boundary and moves up to the root. To perform the instruction "*Update* $(2, K.r_2, Cur\_Node.r_2)$", every processing element in the machine shares the responsibility of merging responses. In addition, the root node regulates the traffic.

To be specific, procedure **Merge_Res** is designed for merging responses. By merging here, a union of two responses is meant. Due to the fact that there can be no more than three neighbors whose size is equal to or greater than that of Cur_Node, the message length of a response is bounded by a constant. To ensure the correctness of updating in a parallel environment, the algorithm checks the newly arrived update instruction and modifies the corresponding components of U_response as necessary.

Clearly, after a certain period of time, there will be two streams entering the root node: the stream formed by the two_tuple <Cur_Node, NEIGHBORS>, and the stream formed by U_responses. The program executes the two instruction streams alternately.

The dominant step of the algorithm is the while loop. Since both NEIGHBORS and U_response are bounded by a constant, each iteration of the loop takes $O(logN)$ time, with a pipelined interval $O(1)$. Therefore, the time complexity of the whole algorithm is $O(N)$, and compares favorably to the best known sequential algorithm which requires a computation time of $O(nN)$ [4].

5.3. Computing Various Geometry Properties of a Region

This section contains a set of algorithms for calculating various geometric properties of a region. In particular, methods for computing the area and centroid of a region will be given in Section 5.3.1. A technique for computing the perimeter of a region is described in the subsequent section.

5.3.1. Computing the Area and Centroid of a Region

The area of a block is defined to be the total number of BLACK pixels comprising the region. The following procedure **AREA** computes the area of a region.

```
Procedure AREA
  begin
    response := 4**K.r1; /*execute parallelly*/
  end
```

A processing element is invoked upon receiving the instruction **AREA** broadcast from the root. The response of the instruction is defined to be the area of the node in the linear quadtree. The responses are then merged level by level by summation.

The centroid of a region is defined to be a pixel (x', y') such that

$$(x', y') = (X/m, Y/m),$$

where m is the area of the region, and $(X, Y) = (\sum x_i, \sum y_i)$ is the summation of the coordinates of the BLACK pixels. According to the result in [4], the values of X, Y, and m can be computed as follows:

```
Procedure CENTROID
  begin
    L:=2**K.r1;
    S:=L*L*(L+1)/2;
    <sumx, sumy, area> :=
          < OD(K.k)*S, EV(K.k)*S, 4**K.r1>;
    response := <sumx, sumy, area>;
  end
```

where <**OD**($K.k$), **EV**($K.k$)>, which is the <x,y> coordinates of $<K, S>$, can be obtained by unshuffling the parameter K.

The procedure is the same as that of **AREA**, except the coordinates of all BLACK pixels are also accumulated. Furthermore, the responses are merged by means of a vector summation.

It is not difficult to see that the time complexity of **AREA** and **CENTROID** is $O(logN)$ and $O(n)$, respectively. If, however, the <x,y> coordinates for each item is explicitly stored, then the time complexity for **CENTROID** reduces to $O(logN)$.

5.3.2 Computing the Perimeter of a Region

Perimeter computation is another basic operation in image processing. The perimeter computation algorithm using a dictionary machine will be specified by the following procedure **PERIMETER**. Procedure **PERIMETER** is a variation of **LABEL-CC**: the algorithm traverses the linear quadtree in ascending key order. For each node P in the linear quadtree being visited, the length of each of its four sides is first included in the value of the perimeter. Then the neighbor nodes of P which have size greater or equal to P need to be considered. For each adjacent node Q that is BLACK, twice the length of the common side is deducted from the value of the perimeter. This reflects the fact that the segment between P and Q does not belong to the boundary of the region. It is assumed that procedure **PERIMETER** follows the conventions made in **LABEL-CC** unless otherwise specified.

```
Procedure PERIMETER
begin
  INI-PORT;
  perimeter:=0;  /*perimeter is associated with the root*/
  while Cur_Node<>null do
    begin
      perimeter := perimeter + 4*(2**Cur_Node.r1);
      for K ε SEARCH-NODES(NEIGHBORS) do
      perimeter := perimeter -2*(2**Cur_Node.r1);
      (Cur_Node, NEIGHBORS):=
            (<NEXT.k, NEXT.r1>, NEXT.r3);
    end
end
```

Similar to **LABEL-CC**, the time complexity of **PERIMETER** is bounded by $O(N)$.

5.4. Two Dimensional Template Matching

Image template matching is a basic image processing operation. It is often used for edge and object detection; filtering; and image registration [3]. An $R \times R$ image matrix $I[0..R-1, 0..R-1]$ is searched with an $S \times S$ template $T[0..S-1, 0..S-1]$. The output is an $R \times R$ matrix $C_{ij} = \sum_{s=0}^{S-1} \sum_{t=0}^{S-1} I[(i+s),(j+t)]*T[s,t]$. In the following, without loss of generality, the subimage $I[i..i+S-1, j..j+S-1]$ is referred to as window W[i,j]. Because of the fundamental nature of this problem and because of its complexity ($O(R^2T^2)$ on a single processor computer) much attention has been devoted to the development of efficient multicomputer parallel algorithms [7,8]. This section introduces a new approach for template matching, using the proposed dictionary machine.

The dictionary machine is composed of $R \times R$ processing elements each of which responsible for processing a window. Assume that the image data is encoded as a linear quadtree, whereas the template is encoded as a sequence of BLACK pixels, with each pixel encoded as <k,0>, where k is the location code of the pixel (the zero value of the second parameter means a block of unit size). Initially, the image data is distributed to the machine in such way that blocks that overlap the window $W[i,j]$, where $i, j = 0,...,R$, are associated with processing element PE(i,j). Moreover, the key associated with PE(i,j) is defined to be SH(i,j). In other words, the odd bits of the key are used to identify the x coordinate of the window and the even bits of the key are used to specify the y coordinate of the window. For this purpose, it is assumed that PE(i,j) is equipped with a queue Q1(i,j) to stored the subimage data. Lastly, the template data is associated with PE(i,j) as a queue Q2(i,j). During the execution of the algorithm, the template data is broadcasted to processing element P(i,j) from the root.

In what follows is a parallel template matching algorithm. The important aspect of this algorithm is that pixels in the template are broadcast to the machine in pipelined fashion. For each node in the machine visited, the intersection test is performed.

```
Procedure Template_Match
  begin
    while Q<>null do
      begin
        Broadcast head(Q) to every elements in the
        machine;
        Q:=tail(Q);
      end;
    With P(i,j) do   /*parallel execution*/
      while Q1<>null and Q2<>null do
        begin
          Cur_Pixel := Shift(head(Q2),i,j);
          Case Cur_Pixel=head(Q1):
            C[i,j] := C[i,j] + 1;
            Q1 := tail(Q1); Q2 := tail(Q2);
          Case Cur_Pixel is contained in head(Q1):
            C[i,j] := C[i,j] + 1;
            Q2 := tail(Q2);
          Case otherwise:
            Q1 := tail(Q1); Q2 := tail(Q2);
        end
  end
```

Procedure **Shift** does the translation of a pixel in the template from one location to another. According to [4], the coordinates of the current pixel, Cur_Pixel, after translate is calculated as (**OD**(Cur_Pixel.k)+i, **EV**(Cur_Pixel.k)+j). Therefore, procedure **Shift** is as follows:

```
Procedure Shift(Pixel,i,j)
  begin
    <x,y>:=<OD(Pixel.k)+i, EV(Pixel.k)+j>;
    Return(SH(x,y));
  end
```

The time complexity of the algorithm can be calculated as follows: the cost of broadcasting is that of $O(logR)$, with pipelined interval $O(1)$. Therefore, the time complexity of the first while loop is $O(logR + \#Q)$, where $\#Q$ is the number of BLACK pixels in the template. On the other hand, the second while loop can be repeated for at most $\#Q+\#Q\,1-1$ times, where $\#Q\,1 = \max\{\#Q\,1(i,j)\}$, for $i,j=0,...R-1$. Since the time complexity of **Shift** is that of $O(n)$ [4], the time complexity of the second while loop is $O(n(\#Q+\#Q\,1))$[4]. Therefore, the time complexity of the whole algorithm is $O(logR + n(\#Q+\#Q\,1))$.

Again, further investigation reveals that the factor n associated with the second term can be dropped if coordinates <x,y> are explicitly stored with each data item. In other words, the optimal worst case performance of the

[4] Note: $O(\#Q+\#Q\,1)=O(S^2)$.

algorithm can be further reduced to $O(logR + \#Q + \#Q\ 1)$, whereas the optimal algorithm on a hypercube interconnection network is that of $O(logR + S^2)$.

6. Conclusion

An adaptation scheme of a linear quadtree on an unsorted VLSI dictionary machine has been proposed for robotics applications. Various operations on images have been developed to support the scheme. Analysis of the algorithms demonstrates that that manipulation of a linear quadtree using a dictionary machine is quite efficient. In particular, the algorithm for labeling connected components, as well as algorithms for computing various geometric properties of a region, are superior to sequential counterparts. For example, linear and logarithmic cost for various operations are desired features of the new scheme. The major advantage of the template matching algorithm is its simplicity and efficiency. Such merit is achieved using the linear quadtree encoding scheme, which takes advantage of the aggregative property of images. Consequently, with the same hardware cost (in terms of the number of PEs and the space requirement), the newly developed algorithm outperforms previous algorithms that use a hypercube.

Finally, it is important to point out that embedding linear quadtrees on an unsorted dictionary machine instead of a sorted machine implies less global constraint on the overall structure. Therefore, region transformation such as translation and rotation, and region operation such as union and intersection can be performed more efficiently. The detailed discussion and the corresponding algorithms are left to another paper.

References

[1] M. J. Atallah and S. R. Kosaraju, "A Generalized Dictionary Machine for VLSI", *IEEE Trans. on Computers*, Vol. C-34, February 1985, pp. 151-155.

[2] J. H. Chang et. al, "Systolic Tree Implementation of Data Structures", *IEEE Trans. on Computers*, Vol. 37, June 1988, pp. 727-735.

[3] R.T. Chin and C. R. Dyer, "Model-Based Recognition in Robot Vision", *Computing Surveys*, Vol. 18, No. 1, March 1986, pp. 67-108.

[4] W. A. Davis and Xiaoning Wang, "A New Approach to Linear Quadtrees", TR 84-9, The University of Alberta, Edmonton, Alberta, Canada, November 1984, and Proceedings of Graphics Interface'85, Montreal, May 1985, pp. 195-202.

[5] W. A. Davis and Xiaoning Wang, "Connected Component Labeling Using Modified Linear Quadtrees", Proceedings Graph Interface'86, Vancouver, May 1986, pp. 235-240.

[6] S. K. Nandy et. al, "Linear Quadtree Algorithms on the Hypercube", *Proceedings of International Conference on Parallel Processing*, St. Charles, Illinois, August 1988, pp. 227-229.

[7] S. Ranka and S. Sahni, "Image Template Matching on SIMD Hypercube Multicomputers", *Proceedings of International Conference on Parallel Processing*, St. Charless, Illinois, August 1988, pp. 84-91.

[8] S. Ranka and S. Sahni, "Image Template Matching on MIMD Hypercube Multicomputers", *Proceedings of International Conference on Parallel Processing*, St. Charles, Illinois, August 1988, pp. 92-99.

[9] A. K. Somani and V. K. Agarwal, "An Efficient Unsorted VLSI Dictionary Machine", *IEEE Trans. on Computers*, Vol. C-34, September 1985, pp. 841-851.