Range Search In Parallel Using Distributed Data Structures

Radhakrishnan Sridhar, Sitharama S. Iyengar, Subbiah Rajanarayanan

Department of Computer Science
Louisiana State University
Baton Rouge, LA 70803, USA.

ABSTRACT

The range search problem is to obtain a set of data points (tuples, records) satisfying a query which specifies a range of values on each dimension (attribute) of the data. Range search has important applications in the areas of databases and computational geometry. A parallel algorithm for range searching is developed here using the concept of distributed data structures. We use the range tree proposed by Bentley as our data structure to be distributed. We show that $O(logn)$ search time can be effected for a range search on n 3-dimensional points using $(2.\log^2 n - 10.logn + 12)$ processors and this is optimal for the range tree distribution. We present a non-trivial implementation technique on the hypercube parallel architecture with which the above time and processor bound can be achieved without any communication overhead. Our algorithm can easily be generalized for the case of d-dimensional range search.

Keywords: parallel algorithm, distributed data structures, range search, hypercube architecture, processor and time complexity

1. Introduction

Let S be a set of n d-dimensional points in R^d. A range query q is a d-range which is the cartesian product of d intervals. The output of the query is all points in S that lie within q. In the case of two dimensions the 2-range is a rectangle and for more than three dimensions the d-range is a hyperrectangle. Thus, the answer to the query q is a set of all points in S that are inside the rectangle or hyperrectangle as the case may be. Range search has several applications including databases and computational geometry [1]. The range search is equivalent to database *selection* operator on a relation.

A considerable amount of work has been devoted to the range search problem [2, 3]. Bentley [4] gives a thorough overview of various multi-dimensional and range searching problems. Several data structures and algorithms for range searching have been proposed and each has trade-offs between storage and time complexity. These structures include k-D-Tree, multidimensional trie, Super-B-tree etc. Bentley and Maurer [5] have shown the lower bound on the time complexity of range search on a set of n d-dimensional data to be $(d.logn)$. With the overlapping-ranges data structure [5] the time bound of $O(d.logn)$ can be obtained at the expense of very high storage cost which is $O(n^d)$. Most practical algorithms use a storage cost of $O(n\log^{d-1}n)$ to obtain a time bound of $O(\log^{d-1}n)$ [1]. Layered Range tree data structure [1] a variant of range tree has the above storage and time complexity; a reduction of $O(logn)$ factor in storage and time complexity of the range tree. Chazelle [6] using the concept of filtering search reduced the storage cost to $O(n.\log^{d-1}n/loglogn)$ while retaining the time complexity.

Recently, there has been a growing interest in developing parallel algorithms for problems in databases and computational geometry. This interest has been enhanced due to the availability of more feasible parallel architectures like the hypercube and the mesh of processors. Baru and Frieder [7] have developed novel algorithms for the execution of relational database operations on a hypercube parallel machine. Algorithms for the execution of the relational *join* operator on the hypercube machine were also given by Omiecinski and Tien [8]. A number of parallel algorithms for computational geometry problems can be found in [9, 10, 11, 12, 13].

More recently, Katz and Volper [14] developed a parallel algorithm for retrieving the sum of values in a region on a two dimensional grid in $O(logn)$ time with $O(n^{1/3})$ processors. In our research, we develop a parallel algorithm for the range search problem using the range-tree as our data structure. We will show that the range search on a two dimensional plane can be effected in $O(logn)$ time with $3/2.logn - 1$ processors. The retrieval of the sum of the values in a two dimensional region can also be done with the above time and processor bound.

One of the keys to efficient parallel searching is the distribution of the data points to be searched. To achieve such an efficiency we use the concept of *distributed data structures*. By *distributed* data structures we mean a typically large data structure, such as a B-Tree, K-d tree, Range tree and others, that is logically a single entity but that has been distributed over several independent processor stores. This concept is not

CH2728-4/90/0000/0014$01.00 © 1990 IEEE

new and frequently arises in the area of distributed data bases. Ellis [15] developed a distributed version of Extendible Hashing for database searching. Distributed data structures of scientific calculation and processing of sets were introduced in [16, 17] respectively. One of the fundamental advantages of the concept of distributed data structure is that processors are assigned to data statically and overheads due to dynamic allocation is avoided. Also the concept of parallel processing of a single data structure have occurred in other forms such as concurrent access to a data structure and issues relating to concurrency control [18].

We will assume that the set S of n d-dimensional points is stored in a range tree (see Section 2). We will present a simple range-tree data distribution scheme and show that the search algorithm is optimal for this data structure (Section 3). Let DS(t, 1) denote the best sequential time taken to search a data structure DS with 1 processor. A parallel algorithm with p processors is optimal for a data structure DS if the search time is DS(t, 1)/p. A non-trivial range search implementation technique on a Hypercube parallel architecture is presented in Section 4. Section 5. presents an argument for the reduction on the number of processors used for range searching. Conclusions are given in Section 6.

2. The Range Tree data structure

The range tree was first introduced by Bentley after which several variants have been proposed. We will first introduce the 2-dimensional range tree. The generalization to d dimensions can be easily visualized. Let S be the set of n 2-dimensional points. First sort the n points based on the value of the x-coordinate. Imagine each point p as an interval $[x_i, x_i]$, where the first and second components are $B[p]$ (begin point) and $E[p]$ (end point). Now, the range-tree corresponding to the first dimension is a rooted binary-tree whose leaves contain the n points sorted and placed from left to right as intervals. An interior node v and its left (v_1) and right (v_2) children has an associated interval with $B[v] = B[v_1]$ and $E[v] = E[v_2]$. Now the second dimensional coordinates i.e., the y-coordinates are stored in the tree as follows. For each interval $I = (B[v], E[v])$ belonging to the node v in the tree, the y-coordinates of the points which project onto the interval I are stored as a binary-tree and the node v points to the root of the binary tree. Figures 2.a and 2.b show a set of points in the plane and its corresponding range tree, respectively. For the case where each point in the plane represents a value and the range query is to sum the values in a specified region we need to store the values S_v at each node v of the range tree as follows. Let t_v be the binary tree corresponding to the y-coordinates at node v. Let $t_v{}^b$ and $t_v{}^e$ represent the left-most and the right-most leaves of the binary tree t_v. The value S_v stored at node v is the sum of the values of the points whose x-coordinates and y-coordinates lie in the interval $(B[v], E[v])$ and $(t_v{}^b, t_v{}^e)$, respectively.

We will now state some properties of the range tree from [1].

Proposition 2.1: The number of nodes selected in the range tree during the range search on any dimension is at most $2.logn - 2$ and there are not more than two nodes selected from each level of the range tree.■

Proposition 2.2: Range searching of an n-point d-dimensional file can be effected by an algorithm in time $O((log\ n)^d)$ using the range-tree technique.■

3. Range tree distribution and parallel algorithm

The key to the success of any parallel algorithm for range searching is determined by the type of data distribution. With $O(n)$ processors effective searches can be made, but, having such large number of processors is highly impractical. In this section, with range tree as the data structure, we present a simple data distribution scheme with which $O(logn)$ search time using $(2.\log^2 n - 10.logn + 12)$ processors is effected for the case of 3-dimensional data points. The technique we describe can easily be extended to the case of d-dimensions. We will assume that the root and the leaves of the tree are at heights h ($n = 2^h$) and 1, respectively.

3.1 Estimation of processor and time-complexity

We now estimate the number of processors required to search in parallel for the case when $d = 2$ and $d = 3$.

In Proposition 2.1 we note that at most $2.logn - 2$ range tree nodes are selected for any range query on a single dimension. This tells us that with $2.logn - 2$ processors we can search the next dimension in parallel. Now, the time-complexity is given by the following simple equation:

$$Q(1, n) = O(logn)$$
$$Q(2, n) = Q(1, n) + O(logn) = O(logn)$$

Here $Q(1, n)$ is the time taken to search the range tree in dimension 1. Let us say that another $2.logn - 2$ processors are available at each of the selected nodes during the processing of the dimension i. The next dimension $i+1$ can also be processed in parallel. Generalizing this scheme to d-dimensions we can see that the time-complexity is now given by the equation:

$$Q(1, n) = O(logn)$$
$$Q(d, n) = Q(d-1, n) + O(logn) = O(d.logn)$$

The total number of processors ($P(d, n)$) required to search a range tree storing n d-dimensional points and achieve the above time-complexity is given by the equation:

$$P(1,n) = 1$$
$$P(2,n) = 2logn - 2$$
$$P(d,n) = P(d-1,n)\,(2logn - 2) = O(\log^{d-1}n)$$

A simple observation that at most 2 nodes are selected at each of the heights from $h-2$ to 1 (Proposition 2.1) helps to reduce the above loose processor bound to a great extent. The number of data points belonging to dimension i stored at node v at height r in the $i-1$-dimension range tree is 2^r. Let $t(v)$ be the range tree corresponding to these points. The number of processors required to do a parallel search on $i+1$-dimensional points stored in $t(v)$ is $2.log(2^r) - 2$. We now present the estimation on the number of processors for $d = 3$. From arguments above we have,

$$P(3,n) = 2.[2.log(2^{h-2}) - 2 + 2.log(2^{h-3}) - 2 + \cdots + 2.log(2^{h-(h-1)}) - 2]$$
$$P(3,n) = 2.\log^2 n - 10.logn + 12$$

In the above processor estimation we have not included processors needed to search a tree stored in the node v at height $h-1$. It is not necessary to have additional processors for node v, since, if node v is selected during the search none of the nodes in the subtree rooted at v will be selected. Note that with at most $2.logn - 4$ processors the node v can be processed. There are $\log^2 n - 5.logn + 6$ processors assigned to the nodes of the subtree rooted at v and they are sufficient to process the tree belonging to node v. We now give a set of equations with which we can estimate the number of processors needed to search d-dimensional data.

Let T_0 denote a complete binary tree on n-nodes with height h ($2^h = n$). The root of T_0 be at height h and leaves are at height 1. Let T_i denote a complete binary subtree on 2^{h-i} nodes and a, a_1, a_2, etc. are integer constants greater than zero. We define a function E as follows:

$$E(m, [a_1.T_i + a_2.T_j + ...]) = E(m, a_1.T_i) + E(m, a_2.T_j) + \cdots$$

For $i > 0$, $E(m, a.T_i) =$

$$\begin{cases} a.T_i & \text{if } i < 2.m - 4 \\ a.T_i + 2.T_{i+2} + 2.T_{i+3} + \cdots + 2.T_{h-1} & \textit{otherwise} \end{cases}$$

$$E(1,T_0) = 1$$
$$E(2,T_0) = T_0$$
$$E(d,T_0) = E(d, E(d-1,T_0))$$

The number of processors required is obtained by applying the following function F to every term $a.T_i$ in the resultant equation.

$$F(a.T_i) = a.(2.log(2^{h-i}) - 2) = 2.a.(h - i - 1)$$

3.2 Distribution of data among processors

In the case of shared memory model data contained in the range-tree need not be distributed among processors and idle processors are allocated dynamically to the selected nodes of the tree. The dynamic assignment of processors to nodes is an overhead to the system as it has to maintain a list of idle processors. Assuming that the selected nodes during the processing of dimension i is I, the time taken to assign idle processors to the selected nodes is $O(I)$. Other obvious benefits of data distribution which include recovery and data reconstruction motivates the need for static assignment of processors to the nodes of the tree. In the previous subsection we have determined the upper bound on the number of processors required to do a parallel range search in $O(logn)$ time for the case of 2- and 3-dimensional data points. We now show how the processors are actually assigned and give the search strategy for the above cases. The case of d-dimension is a natural extension of the approach presented here.

We will call an assignment of processors to the nodes of the range-tree *proper* if the number of processors used in the assignment is less than or equal to the number of processors estimated in Section 3.1 to achieve a time-complexity of $O(d.logn)$, for a range search in d-dimensions. We will now present a proper assignment scheme for the case of 2-dimensions first. Let T be a 1-dimensional range tree of height h. Starting with the leaves at height 1 to height $h-3$, we will allocate 2 processors to each of the heights, since, at most two nodes are selected from each of those heights by Proposition 2.1. If processors p_i and p_j are allocated at height r ($1 \le r \le h-3$), then starting from the left assign nodes at height r p_i and p_j, alternatively. This assignment would guarantee that the two selected nodes would be in different processors. Now, let p_i and p_j be allocated to height $h-2$. The first and the second pairs of nodes at height $h-2$ from the left are assigned p_i and p_j, respectively. The two nodes at height $h-1$ are assigned the same processor that are assigned to their children. The root of T is assigned any of the processor assigned to its immediate child at height $h-1$. The total number of processors used in the assignment is $(2.logn - 2)$. To search the third dimension, the tree corresponding to a node v is assigned new set of processors the same way as described in the case of 1-dimensional range tree. For two nodes v_1 and v_2, their trees are assigned with the same set of processors if processor assigned to v_1 is the same as the processor assigned to v_2. Thus, the above assignment scheme uses exactly the same number of processors as estimated in Section 3.1. Figure 3.a gives the assignment of processors for the tree in Figure 2.b.

The search strategy is very simple. Each processor assigned to a node v at height r is responsible for giving the search message to the appropriate processor at height $r-1$.

The search message is sent from a processor v at height r to a processor at height $r-1$ if the query interval does not completely contain the interval $(B[v], E[v])$. If the interval containment is satisfied no more search message is issued from v and the tree at v is searched next. We know that each processor is assigned more than one tree node. The node interval to be chosen for comparison with the query interval and the processor to which the search message has to be sent are all done by the processor with the help of simple array indexes. We skip the details here.

Theorem 1: The range search on a 2-dimensional and 3-dimensional sets of n points can be done in $O(logn)$ time with $(2.logn - 2)$ and $(2.\log^2 n - 10.logn + 12)$ processors, respectively. The sum of values in the range can also be done for the case of 2-dimension and 3-dimension in $O(logn)$ time with the above processor bounds.

Proof: The sum of values in a range can be retrieved using the values S_v stored at each node of the range tree (see Section 2.). The rest of the result follows from the discussion above.■

We would like to end this section with the note that there can be more than one proper assignment scheme. Figure 3.b gives another proper processor assignment scheme for the tree in Figure 2.b.

4. Range searching on the Hypercube machine.

We will now proceed to give details on how the nodes of the tree can be mapped on to the hypercube for efficient searching. First we will present the 2-dimensional case. A good mapping is one which minimizes the communication time in the hypercube. Consider the processor assignment discussed in the previous section. A mapping which takes the i th processor and maps it to the i th hypercube node would require a total communication time of $O(logn.loglogn)$, since we require $O(logn)$ processors for range searching in 2-dimensions and $O(logn)$ is the height of the range-tree corresponding to the first dimension. Hence the total search time for range-searching in two dimensions using a range-tree on a hypercube would be $O(logn.loglogn)$. We now present a mapping which would reduce the total search time to $O(logn)$ on the hypercube.

Consider the assignment of processors as discussed in the previous section to the nodes of the range tree corresponding to dimension 1. A processor at height r (p_i) after checking its range will send the search message to another processor at height $r-1$ (p_j). If p_i and p_j are adjacent to each other in the hypercube the communication time is a constant, otherwise, it can be as high as $O(loglogn)$ the diameter of the hypercube. We now present a mapping (embedding) technique which gives constant-time communication time between processors in adjacent levels of the range tree.

It can be seen that a processor p_i at height r is adjacent to two processors at heights $r-1$ and $r+1$. Based on the processor assignment discussed earlier and the adjacency relationship between processors we form a graph G called the *processor assignment graph.*

A processor assignment graph G consists of $(2.logn - 2)$ nodes and is connected as follows. The graph G consists of 4 chains c_1, c_2, c_3, and c_4. The chains c_1 and c_2 contains odd and even numbered processors respectively (will be referred as odd and even numbered nodes). Two odd or even numbered nodes are adjacent in their respective chains iff they belong to adjacent levels of the range tree. The chain c_3 (c_4) formed when an edge is drawn from every node a in c_1 (c_2) to every node b in c_2 (c_1), whenever a and b are in adjacent tree levels (see Figure 4.a).

We will show in Proposition 4.1 that the graph G cannot be embedded in the hypercube with dilation 1 (i.e., all adjacent nodes in G will not be adjacent when embedded in the hypercube). For dilation two embedding we require that the dimension of the cube be $O(logn)$, i.e., with a cube containing $2^{O(logn)}$ nodes. In this case the expansion, (i.e., the ratio of the number of hypercube nodes to the number of graph nodes) is exponential. The processor assignment graph can be embedded onto a binary tree with dilation 3. The binary tree can then be embedded onto an hypercube with an expansion of one and a dilation of 3 [19]. Thus, the processor assignment graph can be embedded onto an optimal hypercube with dilation 6.

Proposition 4.1: The processor adjacency graph G cannot be embedded on a hypercube with dilation one.

Proof: In a hypercube 2 nodes a and b can be adjacent at most to the two same set of two nodes c and d. In G two nodes a and b can be adjacent to the same 4 set of nodes. This implies either a should be adjacent to b or vice versa. Now, the dilation is 2.■

Lemma 4.2: The processor adjacency graph G can be embedded on to an optimal hypercube with dilation 6.

Proof: First we will show the processor adjacency graph G can be embedded onto a binary tree with dilation 3. Let $c_j(i)$ refer to the i th element in the j th chain. Make $c_1(2)$ the root r of a binary-tree T. Make $c_2(2)$ the right child of r and $c_1(1)$ and $c_2(1)$ the left and right children of $c_2(2)$. We use the following segment to construct the rest of the tree T.

1. $i = 3$;
2. Make $c_1(i)$ the left child of r;
3. $p = c_1(i)$;
4. Make $c_2(i)$ the right child of p;
5. Make $c_1(i+1)$ the left child of p;

6. If not all nodes in c_1 has been processed increment i and GOTO step 3;

Clearly, the above construction would obtain an embedding of G on to a binary tree with dilation three. Figure 4. gives the graph G and its corresponding binary-tree representation. The above segment of code guarantees that two adjacent nodes in G are at most three distance apart in T. Now, using known algorithms [19] we can embed T onto an optimal hypercube with dilation 3. Thus, G can be embedded onto an optimal hypercube with dilation 6.■

Theorem 2: Theorem 1. holds in the case when the processors are arranged as an hypercube architecture.

Proof: From Lemma 4.2 it is clear that the communication time for the search message to travel from one level to the adjacent one in the range tree is a constant. In the case of 2-dimensions after embedding G in a hypercube with $(2.logn - 2)$ nodes, we can easily see that the range search can be done in $O(logn)$ time. In the case of 3-dimensional range search the processor bound can be achieved with several hypercube machines of different sizes as follows. With the availability of two cubes of size $(2.logn - 4)$, two cubes of size $(2.logn - 5)$ and so on, the 3-dimensional range search can be done using a total of $(2.\log^2 n - 10.logn + 12)$ processors. This is done by embedding each of one of the subtrees optimally onto their respective hypercubes.■

It would be interesting to see if given the availability of single hypercube can the search be carried efficiently. It turns out that it is possible and for a 3-dimensional range search using an hypercube with $O(\log^2 n)$ processors.

5. Processor reduction

In this section we will show that $(3/2.logn - 1)$ processors are sufficient to effect a range search in $O(logn)$ time for a set of points in the plane and thus saving $((logn)/2 - 1)$ processors. The approach can be generalized to d-dimensions easily. The processor reduction is illustrated in the following example. Let T_{256} be a range tree with 256 leaves. Time taken by a single processor to process the tree T_{256} is in the worst case 8 units of time. Two T_{16} trees can be processed sequentially by a single processor in 8 units of time. This means that with two processors, the tree T_{256}, and two T_{16} trees can be processed in 8 units of time instead of using three processors and still requiring 8 units of time. We will generalize the above idea and estimate for a two dimensional range tree of height h. The range tree $T_{2^{h-2}}$ requires $h - 2$ units of time. Since there are two such trees at height $h - 2$, we will allocate two processors. For similar reasons we have to allocate two processors for each of the heights from $h - 2$ to $(h-2)/2 - 1$. For heights from $(h-2)/2$ to 1 we allocate a single processor. The total number of processors allocated to the entire tree is now $(3/2.logn - 1)$. Finding a proper processor assignment scheme with reduced number of processors is easy.

The estimation on the number of processors mentioned in Section 3. can further be reduced as the processors allocated during the processing of dimension i can be used to process dimension $i + 1$.

6. Conclusion

The problem of range search was solved in parallel using the range tree data structure. The nodes of the range tree were distributed among the processors in such a way that the search can be carried efficiently in parallel. It can be easily shown that our our algorithm is optimal for the chosen data structure in the case of arbitrary dimension $d = O(1)$, from Proposition 1.1. Based on the assignment of processors to the nodes of the range tree a processor assignment graph was created. The processor assignment graph was embedded onto an optimal hypercube for the execution of the range search without any communication overhead. Finally, a processor reduction argument was presented.

Acknowledgements: The authors would like to thank Dr. Zheng, Dr. Kraft, and Mr. Salil Menon for their comments on a earlier version of this paper. We are deeply indebted to the anonymous refrees whose comments greatly enhanced the presentation of our work.

References

[1] Preparata, F.P. and Shamos, M.I., "Computational Geometry : An Introduction," *Springer-Verlag, New York*, 1985.

[2] Bentley, J.L. and Friedman, J.H., "Data Structures for Range Searching," *ACM Surveys*, vol. 11, no. 4, December 1979.

[3] Iyengar S.S., Rao N.S.V., Kashyap R.L., and Vaishvani V.K., "Multidimensional Data Structures: Review and Outlook," *Advances in Computers*, vol. 27, pp. 70-119, 1988.

[4] Bentley, J.L., "Multidimensional Divide-and-Conquer," *CACM*, vol. 23, no. 4, pp. 214-229, April 1980.

[5] Bentley, J.L. and Maurer, H.A., "Efficient Worst-Case Data Structures for Range Searching," *Acta Informatica*, vol. 13, pp. 155-168, 1980.

[6] Chazelle, B., "Filtering Search: A New Approach To Query-Answering," *Siam J. Comput.*, vol. 15, no. 3, pp. 703-724, August 1986.

[7] Baru, C. K. and Frieder, O., "Database operations on Cube-Connected Multicomputer System," *IEEE Trans. on Computers*, vol. 38, no. 6, pp. 920-927, June 1989.

[8] Omiecinski, E. and Tien, E., "A Hash-Based Join Algorithm for a Cube-Connected Parallel Computer," *IPL*, vol. 30, pp. 269-275, March 1989.

[9] Atallah, M.J., Cole, R., and Goodrich, M.T., "Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms," *Siam J. Comput.*, May 1989.

[10] Dehne, F. and Stojmenovic, I., "An $O(\sqrt{n})$ Time Algorithm for the ECDF Searching Problem for Arbitrary Dimensions on a Mesh-Of-Processors," *IPL*, vol. 28, 1988.

[11] Miller, R. and Stout, Q. F., "Mesh Computer Algorithms for Line Segments and Simple Polygons," *Int'l Conf. on Parallel Processing*, pp. 282-285, 1987.

[12] Karlsson, R.G. and Overmars, M.H., "Normalized Divide-and-Conquer: A Scaling Technique for Solving Multi-Dimensional Problems," *IPL*, vol. 26, pp. 307-312, 1987/88.

[13] Stojmenovic, I., "Computational Geometry on a Hypercube," *Int'l Conf. on Parallel Processing*, pp. 100-103, 1987.

[14] Katz, M.D. and Volper, D.J., "Geometric Retrieval in Parallel," *Jou. of Parallel and Distributed Computing*, vol. 5, pp. 92-102, 1988.

[15] Ellis, C.S., "Distributed Data Strucutures: A Case Study," *Int'l Conf. on Parallel and Distributed Computing*, 1985.

[16] Scott, L.R., Boyle, J.M., and Bagheri, B., "Distributed Data Structures for Scientific Computation," *Departments of Computer Science and Mathematics, Pennsylvania State University*, 1987.

[17] Mu, Z. and Chen, M.C., "Communication-Efficient Distributed Data Structures on Hypercube Machines," *Department of Computer Science, Yale University*, 1986.

[18] Lehman, P. and Yao, S.B., "Efficient locking for concurrent operations on B-trees," *ACM TODS*, vol. 6, no. 5, pp. 650-670, December 1981.

[19] Monien, B. and Sudborough, I.H., "Simulating Binary Trees on Hypercubes," *3rd AWOC*, 1988.

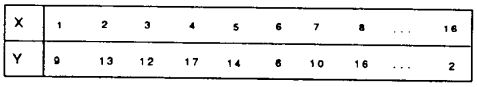

Figure 2.a - A set of points in the plane.

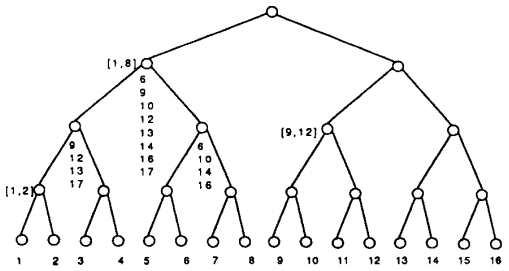

Figure 2.b - A 2-dimensional Range Tree. Some of the intervals are marked and the Y-values are shown as a sorted array.

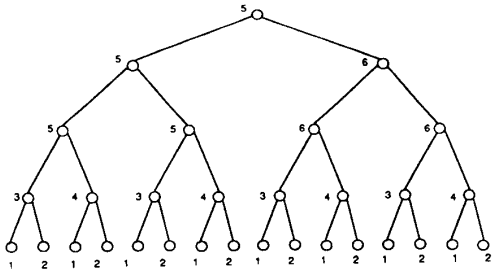

Figure 3.a - Processor assignment scheme

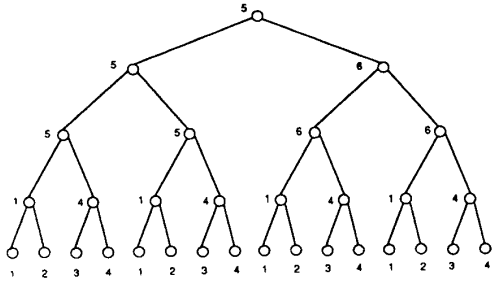

Figure 3.b - Another processor assignment scheme

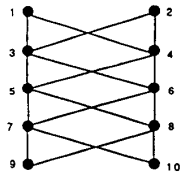

Figure 4.a - A processor assignment graph

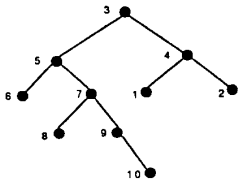

Figure 4.b - The binary tree embedding of the processor assignment graph in Figure 4.a.