PROJECT REPORT

On

PARALLEL IMPLEMENTATION OF RED - BLACK TREE

Submitted by-

P Sharath Chandra (170001033)

PVR Deepak (170001034)

3rd Year

Under the Guidance of

Dr. Surya Prakash

Department of Computer Science and Engineering

Indian Institute of Technology, Indore

Introduction:

Red - Black Tree:

It is a self-balancing Binary search tree, where each of its nodes is colored either red or black. The height of the red-black tree is O (log (n)) [n, being number of nodes], which makes it more usable to search for elements. The key difference between a red-black tree and AVL tree is in the insertion of new elements into the trees; AVL takes O(log(n)) time whereas Red - Black tree takes O (1) time.

Properties of a Red-Black Tree:

1. The root is always Black.
2. If a node is red, then both of its children are black.
3. The number of black nodes from the root of the tree to any of the leaf nodes is same.

Few applications of Red – Black Tree:

- Java: TreeMap, TreeSet
- C++ STL: map, multimap, multiset
- Linux Kernel: Completely Fair Scheduler

Few basic operations on a Red-Black Tree:

- **Construction**:
  - Given a set of initial nodes, this operation constructs the red-black tree, satisfying the properties of BST and red-black tree.
- **Searching**:
  - Given a red-black tree and an element, this operation returns whether the element is present in the tree; moreover, if it is present, it returns the index of that particular node.
- **Insertion**:
  - Given a red-black tree and an element, this operation inserts the element into the tree; thereafter solves any conflicts to restore the properties of red-black tree.
- **Deletion**:
  - Given a red-black tree and an element, this operation finds and deletes the node that contains the value of the element; thereafter restores the properties of red-black tree by certain transformations.

Algorithms:

A. Construction:

Input: n distinct elements (a1, a2, a3…an)

Case 1: when $n=2^q-1$, we can form a perfect binary tree. The property that is being exploited to construct a perfect binary tree is that the elements at every level form a A.P(Arithmetic Progression) and all nodes are colored black.

So, we define arrays C, J, D where

D[i] represent depth of node I,

(D[i] = $\lfloor \log(i) \rfloor$)

J[i] represents the number of nodes whose depth is D[i] and whose elements are smaller than x_i

(J[i] = i- $2^{D[i]}$)

C[i] represents the index of item to be stored in x_i.

Processor p_i stores $a_{CI(i)}$ in x_i.

Case 2: when $n \neq 2^q-1$, we build an empty perfect binary tree of height $\lceil logn \rceil$.

In the n^{th} level of tree, we place the nodes such that left-most side is filled and colored red.

Here, we define number L and arrays CI, CL, J, I, D where

D[i] = $\lfloor \log(i) \rfloor$,

J[i] = i- $2^{D[i],}$

CI[i] represents index of items to be stored in x_i`

CI[i] = ((2*J[i] + 1) *2^$\lfloor \log(n) \rfloor$)/$2^{D[i]}$,

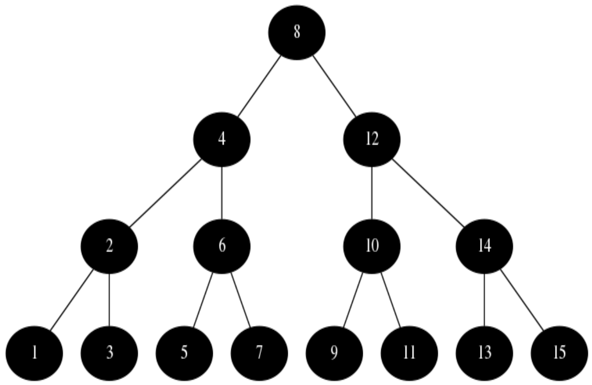
CL[i] = $\lfloor \mathrm{CI[i]}/2 \rfloor$,

L = (n+1)-2^$\lfloor \log(n) \rfloor$

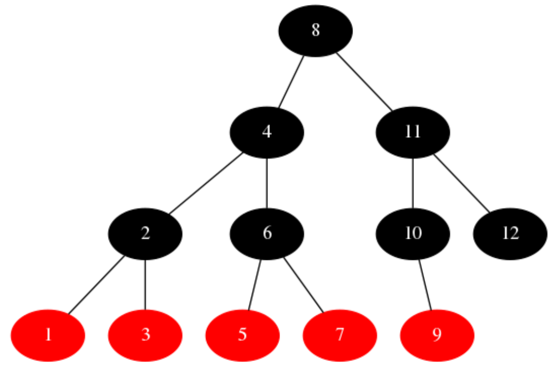
I[i] = min(CI[i],CI[i]-CL[i]+L)

Processor p_i stores $a_{I[i]}$ in x_i.

All the leaf nodes are colored red and remaining black.



Case (i): $n=2^q-1$



Case(ii): $n\neq 2^q-1$

B. Searching

Input: Red-Black tree(T), k sorted elements (a1, a2, a3…ak)

Sequential Algorithm: A query element q is searched in Tree T, returning index of that element if it already exists in the tree or return a index of where to be placed.

Parallel Algorithm:

Since we have k elements to be searched, we assign each element to each processor using O(k) processors on CREW PRAM and run a sequential algorithm.

C. Insertion

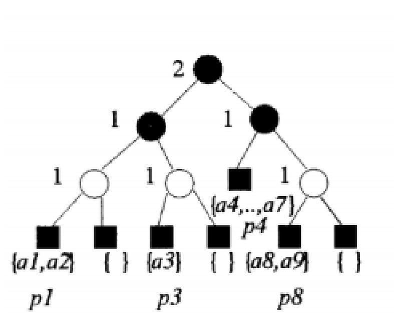
Input:

Red-Black Tree(T), k sorted elements (a1, a2, a3 … ak)

Algorithm:

Step 1: Using above search algorithm, we search for indices of the k elements.

Step 2: For each external node, we make a block consisting of the items that reached the index of the node.



After Step 2

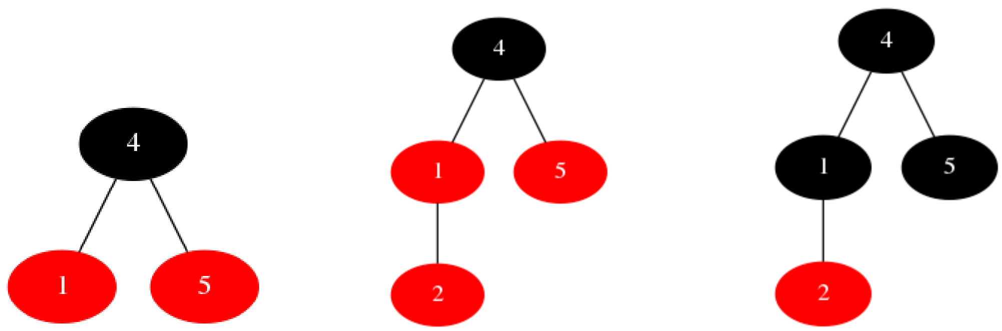
Step 3: Each block b_i is assigned to a processor p_i. For each block, insert the middle element of the block into the tree at that index.

Now divide the remaining block into two parts. If a block is from a_i to a_j, insert $a_{(i+j)/2}$ into the tree and the block is divided into two parts from i to (i+j)/2 and (i+j)/2 + 1 to j. These are made left and right children of the new node respectively. Color of each inserted node is red.

Step 4 : Rebalancing

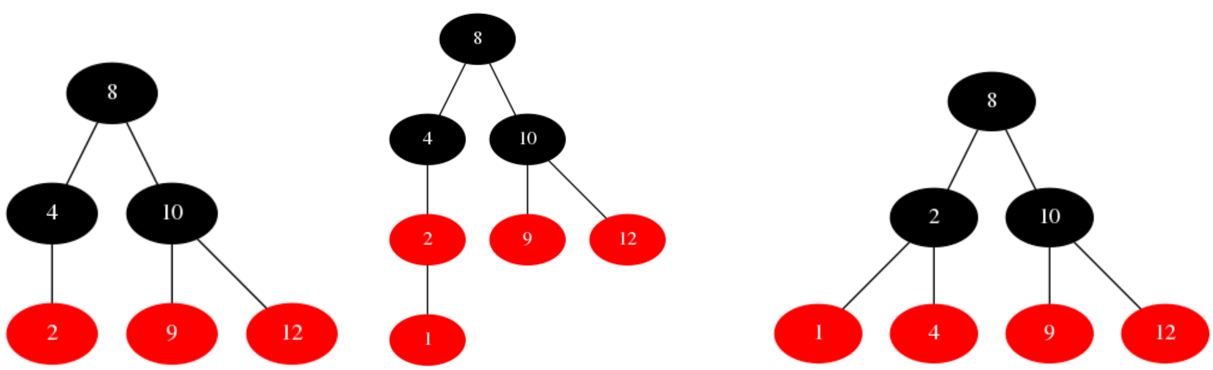
After inserting one element from each block, resulting tree may not satisfy the properties of red-black tree. So, the following re-arrangements are performed at each inserted node if there is a conflict at that node: -

a) Promotion/Demotion:
Example: Addition of node '2'



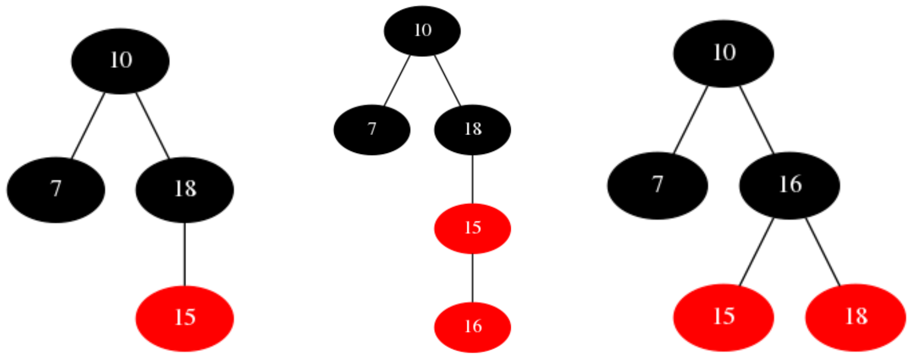
Initial intermediate final

b) Single rotation



Initial intermediate final

c) Double rotation



Initial intermediate final

Now step 3 and step 4 are iterated until all the nodes are inserted and no conflict exists.

Complexity:

'n' is number of nodes in the tree

'k' is the number of elements to be searched from/inserted into tree.

A. Construction:

Sequential Time Complexity: O(n)

Parallel Time Complexity: O(1) using O(n) processors (CREW PRAM)
Work Complexity: n*O(1) = O(n); So work optimal parallel algorithm

B. Search:

Sequential Time Complexity: k*O(log(n)) = O(k*log(n))

Parallel Time Complexity: O(log(n)) using O(k) processors (CREW PRAM)

Work Complexity: O(k*log(n)); So work optimal parallel algorithm

C. Insertion:

Sequential Time Complexity: k*O(log(n)) = O(k*log(n))

Parallel Time Complexity:

Step 1: O(log(n)) using O(k) processors

Step 2: O(1) using O(k) processors

Step 3: O(1) time

Step 4: Atomic operations (promotion, demotion, single rotation, double rotation) takes O(1) time

Step 3 and step 4 and totally iterates O(log(k)).

Total time Complexity: O(log(n) + log(k))

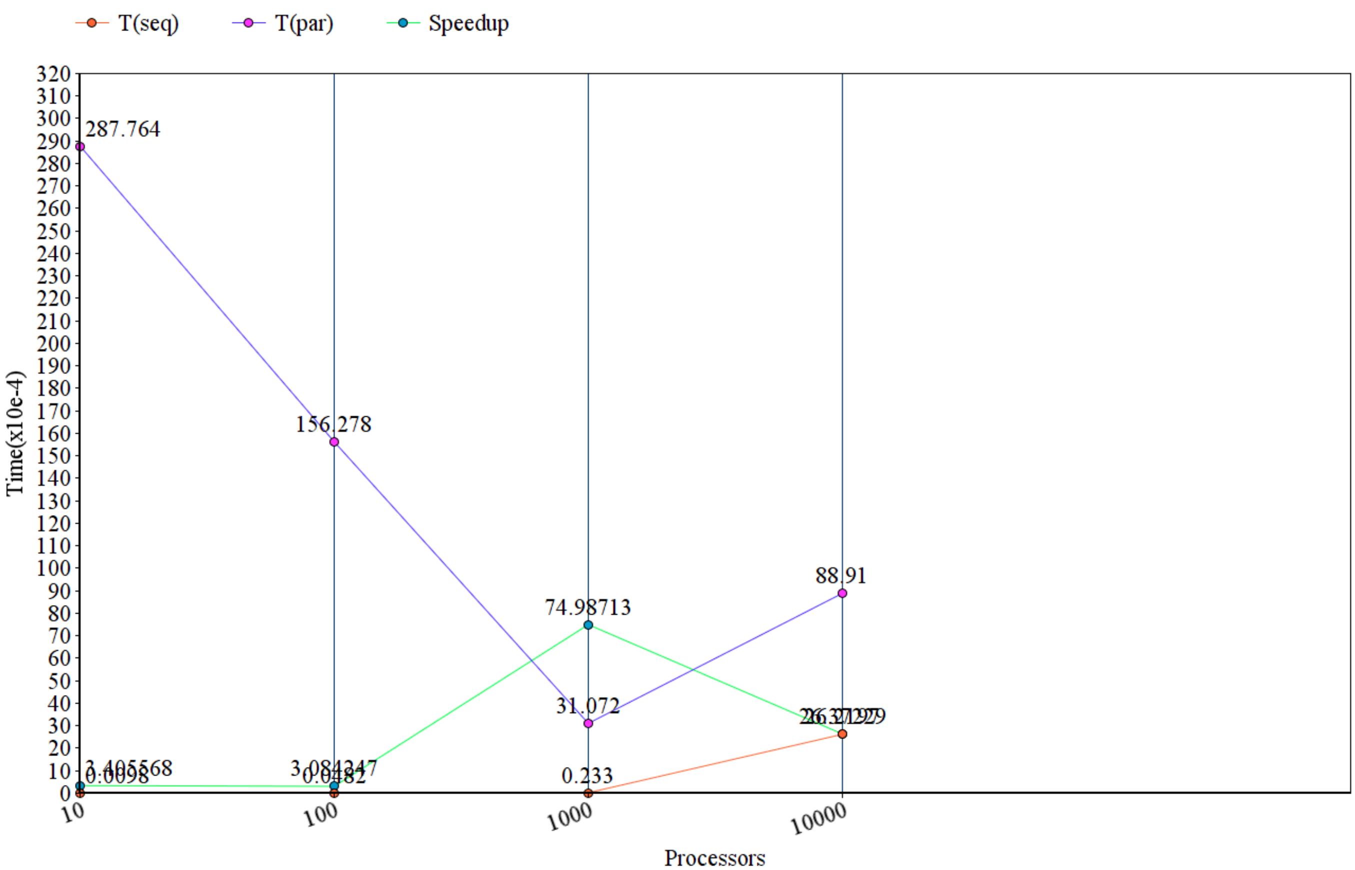
Work Complexity: O(k*(log(n)+log(k))); So Not work optimal.

Results:

A.) Construction:

No. of processors	Sequential Time	Parallel Time	Speedup
10	0.00000098	0.0287764	0.003405568
100	0.00000482	0.0156278	0.003084247
1000	0.0000233	0.0031072	0.07498713
10000	0.0000234476	0.008891	0.2637229
100000	0.00262197	0.0248426	0.105543301
500000	0.00618674	---	---

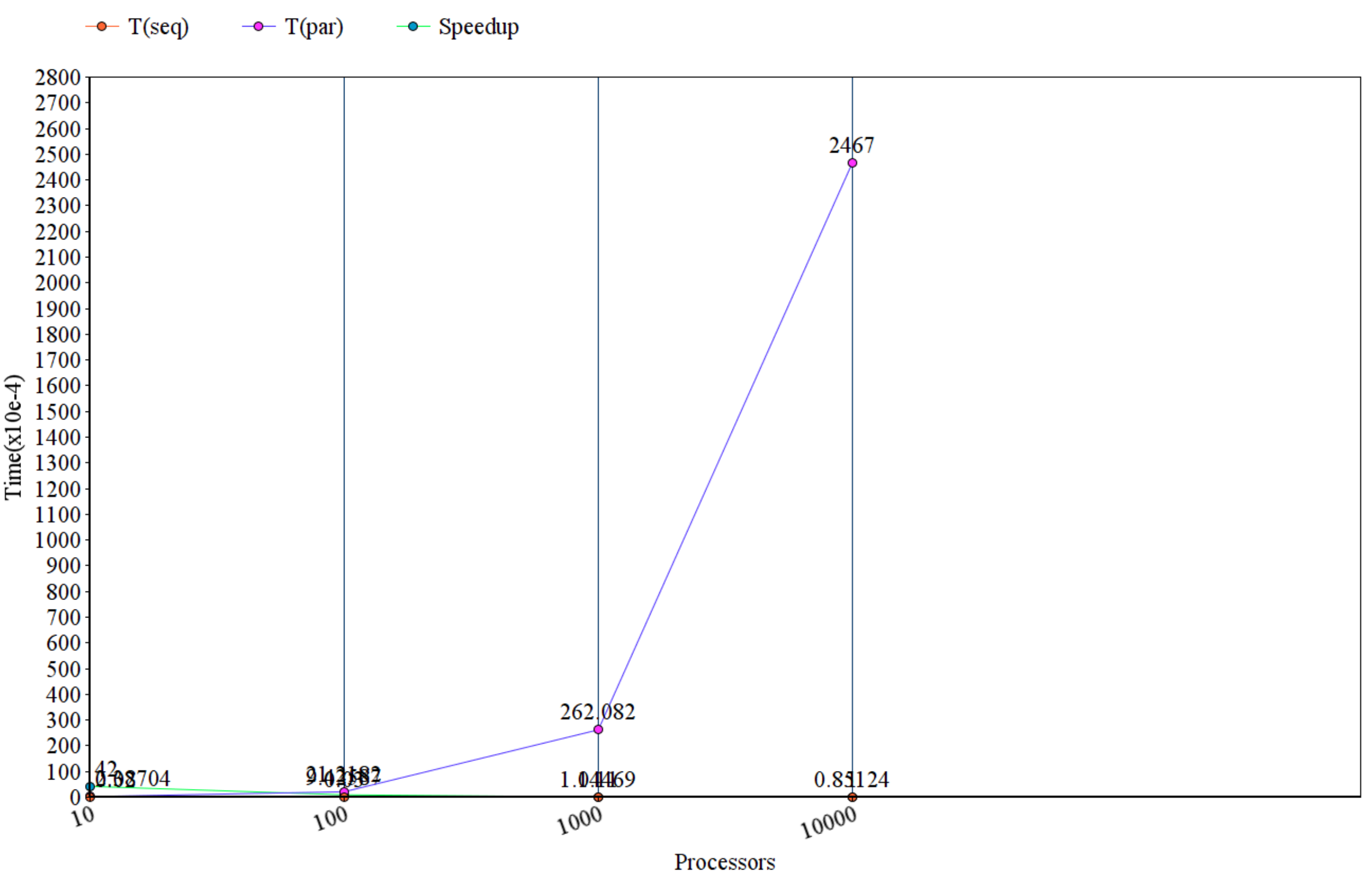
Graph between number of processors and Sequential Time, Parallel Time, Speedup:



B.) Search:

No. of processors	Sequential Time	Parallel Time	Speedup
100	0.00003	0.00212182	0.000942587
1000	0.00011	0.0262082	0.00114469
10000	0.001	0.2467	0.0085124

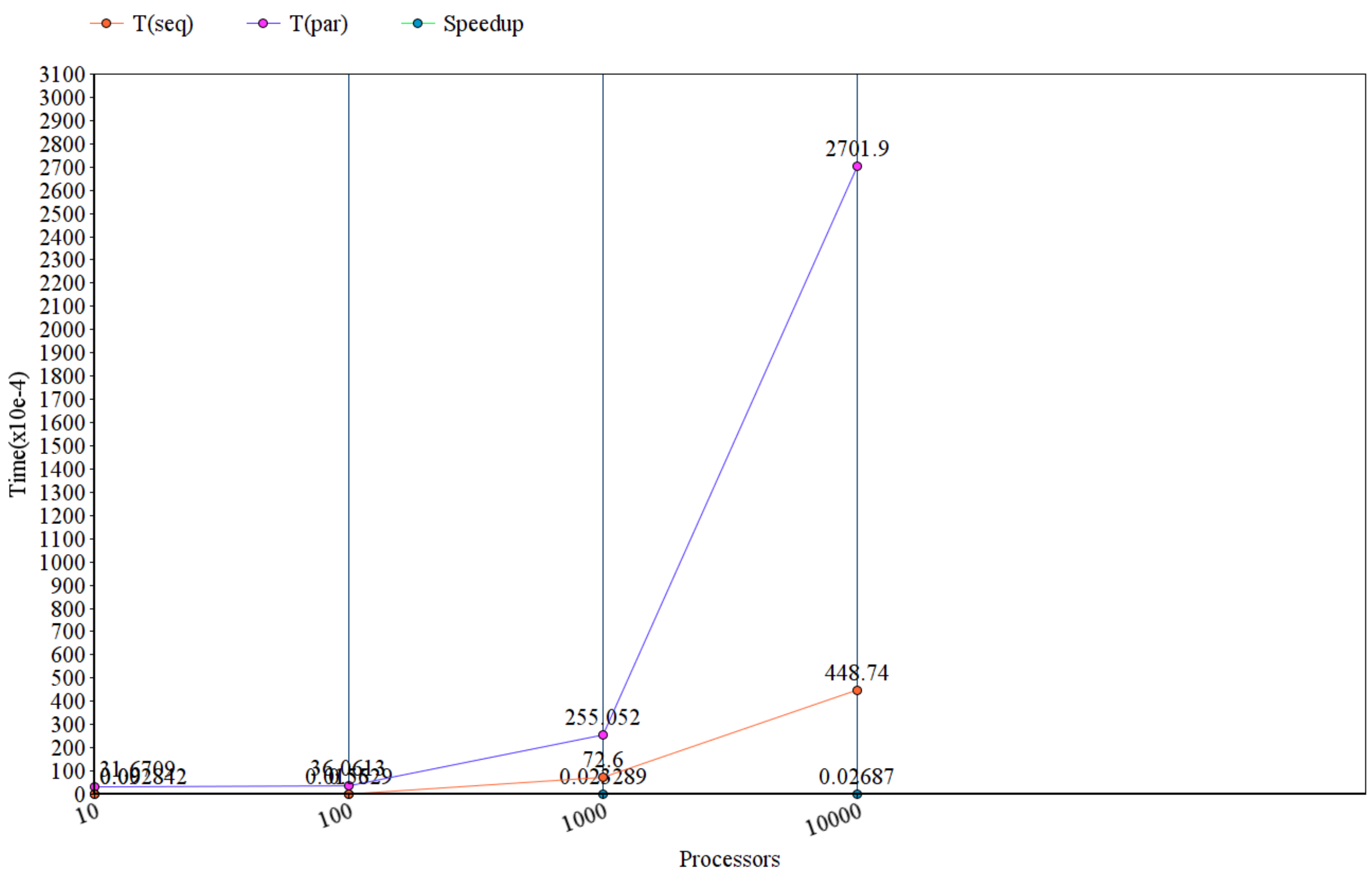
Graph between number of processors and Sequential Time, Parallel Time, Speedup:



c.) **Insertion**:

No. of processors	Sequential Time	Parallel Time	Speedup
100	0.00056	0.00360613	0.1552911
1000	0.00594	0.0255052	0.232893
10000	0.00726	0.27019	0.026889

Graph between number of processors and Sequential Time, Parallel Time, Speedup



References:

[Parallel algorithms for red–black trees - Heejin Park, Kunsoo Park.]()