

STUDY OF EFFECT OF PARALLELISM ON TIME COMPLEXITIES ON VARIOUS OPERATIONS FOR TREE DATA STRUCTURE

¹Anitha Modi , ²Charvak Patel , ³Dvijesh Bhatt

^{1,2,3}Institute of Technology, Nirma University

ABSTRACT

Parallel processing is a way to extract the high amount of power from the CPUs. Traditional computers had limited amount of processors i.e. limited ability to handle more than one threads but now using GPUs we can handle thousands of threads simultaneously. If one were to build and process a data structure using multi-threading (Parallel Processing) time complexities and basic operations should change in a way that it can take benefit of number of processors. Using this concepts, time complexity of some operations can change dramatically. The paper discusses the effect on time complexities on various operations performed in a tree data structure due to introduction of parallelism.

Keywords: Parallel processing, data structures, trees, complexity, link list

I. INTRODUCTION

One of the several ways to build more powerful processors is to simply increase its clock speed. But now due to technological boundaries it has reached to a limit. An alternate to that would be to add more number of processors in a system. But the limitation of the traditional algorithm mostly written for single processor is, it might not be able to extract the computational power appropriately with larger number of processors. We need to restructure or redefine algorithms and data structure which can extract this computational power effectively. Thinking parallel is the way.

II. GOING PARALLEL WITH DATA STRUCTURES

Various operation performed on the data structure is basically a job that is supposed to be done. The job can be divided into smaller jobs. Assume that there is an operation 'J' on the data structure. Now 'J' consist of 'n' smaller jobs $J = \{J_1, J_2, \dots, J_n\}$ These jobs are mutually exclusive i.e. independent of each other and there are 'm' processors $P = \{p_1, p_2, \dots, p_m\}$ where $n \leq m$. So it's clear that by assigning the jobs on processors the job 'J' can be done. In case of lightly or heavily coupled jobs there is communication overhead that can be handled by created separate job sequences. If there are $n > m$ process then process scheduling can be introduced to schedule the jobs. This can affect the complexity of the parallel algorithm designed.

So we have to basically perform following analysis on existing data structures

1. Identify operations, divide it in sub-operations.
2. If they are independent completely then do nothing but if they are not, define how they will communicate with each other in order to do their parent operation.

3. It's not always we will find a good way to do an operation in parallel, there are many operations which can't be broken and needs to be done using single-thread only.

III. DATA STRUCTURES

Here are some data structures who has some operations which we can be redefined which if done using parallel processing will yield much less time complexity. There are many operations which will not change, we have only mentioned those which needs to be modified. The parallelism is explained using a reference multithreaded model^[1]. The conventions^[1] followed in the upcoming discussion is T_{span} : Time Complexity, *Spawn* function call mentioned will be run on the newly created thread and *Sync* used for parent thread waiting till all the child threads completes its execution. Pseudo-code^[1] and occasional C++ is used to illustrate algorithms.

3.1. Height off a Binary Tree

The height of the tree is calculated as

$$\text{Height}(T) = 1 + \max(\text{Height}(T \rightarrow \text{left}), \text{Height}(T \rightarrow \text{right}));$$

$$\text{Height}(\text{NULL}) = 0$$

Here it clear that, time complexity of tree of height for this operation is $T(h) = 2T(h-1) + c$. Time complexity of this operation is $O(2^h)$. Now here two recursive call on Right and Left sub-tree are independent of each other so we can put each on one thread so code will be like this

```
LeftH= spawn height(T->left);
rightH = height(T-> right);
sync;
return 1+ max (LeftH, rightH);
```

The two recursive calls LeftH, rightH works in parallel. The total time taken is equal to only one recursive call and hence $T_{span}(h) = T_{span}(h-1) + c$. So time complexity is $O(h)$.

3.2. Sorted Array to Balanced BST

A division of the given sorted array is done at the middle element which is made root. Further the left and right part are subdivided at the middle element which is made the root. The recursive process is repeated as long as division of element cannot be done. Finding middle element and making it root takes $O(1)$ time and time complexity is $T(n) = 2T(n/2) + c$. Here recursive calls on left and right are independent of each other so they can be executed in parallel.

Pseudo-code will be

```
BuildTree( a[n], l, r)
```

```
Begin
```

```
If l>r return NULL;
```

```
Else mid=(l+r)/2;
```

```
  Create a[mid] node
```

```
left = spawn BuildTree(a , l, mid-1);
```

```

right = BuildTree( a,mid+1,r);
sync;
End

```

The time complexity will be due to two parallel recursive calls, $T_{span}(n) = T_{span}(n/2) + c$. and hence the complexity is $O(\log(n))$.

3.3. Binary Tree to Linked List

Converting binary tree into linklist involves three steps^[4], which are stated as follows

1. Convert the root into linklist node.
2. Recursively call the left and the right subtree to form linklist
3. Join the left and right linklist.

The process is shown in the following figure:

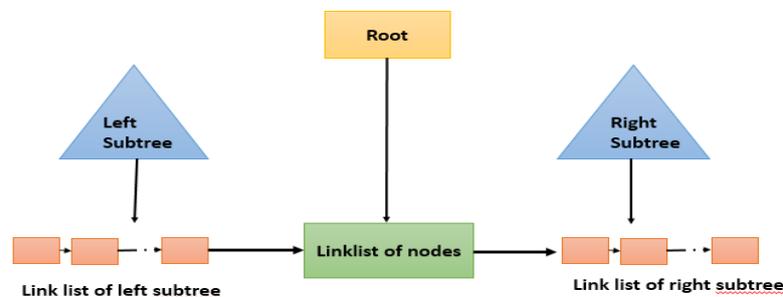


Figure:1 Converting Binary Tree to Linked List

Converting left sub-tree and right sub-tree to linked list is independent operations and can be done in parallel. Pseudo-code will be as below.

```

TreeToLL( root)
Begin
if( root == NULL) return NULL
else
createlinklist node for root key
leftLL = spawn TreeToLL(root->left)
rightLL = TreeToll(root->right)
sync;
connectleftLL,root key ,rightLL
End

```

The time complexity without multithreading would be $T(n) = 2T(n/2) + c$. Which is $O(n)$ and with multithreading $T_{span}(n) = T_{span}(n/2) + c$. Which is $O(\log(n))$.

3.4. BST To the Sorted Array

Due to dependency there are certain changes in that needs to be incorporated for parallelism. Following is the algorithm to convert binary search tree into sorted array.

```

TreetoSortedArray( Root, array)
Begin
  If root is NULL return
Else
  Index=TreetoSortedArray(Root_left, array)
  Array[index++] = root_key
  Index=TreetoSortedArray(Root_right, array)
  Array[index++] = root_key
End

```

From the above algorithm it is observed that left and right sub-parts are not independent. There is a shared variable *index* which is needed by the sub-parts. To introduce parallelism we need to change the structure of the binary tree. In addition now Tree-Node has one extra field called *weight* which is number of nodes in the tree rooted at that node which can be accommodated in the data structure during creation or can be updated in the entire tree in $O(h)$ time. The following figure: 2 shows the modified data structure to support parallelism.

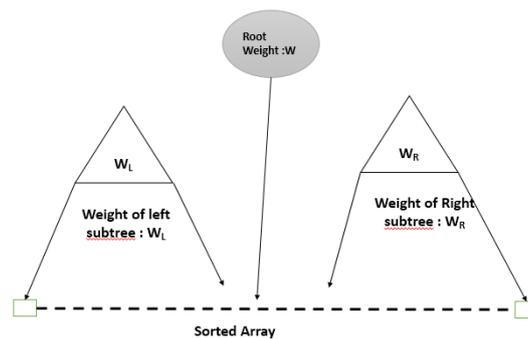


Figure: 2 BST To Sorted Array

Without introducing parallelism, time complexity would be $O(n)$ because we need to fill weight of each node following this recurrence relation $T(n) = 2T(n/2) + c$. But by introducing parallelism the recurrence relation is $T_{span}(n) = T(n/2) + c$, which gives time complexity of $O(\log(n))$. The position of the current node in the sorted array can be computed as

Position(current-Node) = 1 + number of nodes in the left-subtree + start index of array.

Recursive relation for this will be $T_{span}(n) = T_{span}(n/2) + c$; Which gives us time complexity of $O(\log(n))$.

3.5. Building Max or Min Heap

BuildMaxHeap(A^[1])^[4] process builds heap form an array in $O(n)$ time but using parallel processing we can build heap in $O(\log(n))$ time. The steps for traditional process is to start from the non-leaf node, apply Max-Heapify^[1] and go to next non-left node and follow the procedure till root.

This procedure gives us time complexity of $O(n)^{[1]}$. This process is iterative and hard to make it parallel so we will define the recursive approach. It's more like post-order processing

```
Procedure BuildHeap(array,length , node)
Begin
    If Node > length or node < 0 return
    Spawn BuildHeap( array , length left(node))
    Sync
    Max-Heapify(node)
End
```

Time complexity of the above algorithm is $T_{span}(h) = T_{span}(h - 1) + T(h)$. Time complexity will be summation from $h = 0$ to $h = \log(n)$ of $O(h)$. Which will result in $O(\log(n))$.

IV. CONCLUSION

In the above sections we have observed that by introducing parallelism in the tree data structures, the operations which were of order $O(n)$ reduced to $O(h)$. We also had to change the way we think and construct trees. Trees are recursive data structure which gives us opportunities to perform operations in parallel. By introducing modified trees, new ways can be established so that parallel processors can take more advantage of it.

REFERENCES

- [1] Introduction to Algorithms 3rd edition by Thomas H. Corman, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
- [2] Intro to parallel processing with CUDA
- [3] http://www.nvidia.com/object/cuda_home_new.html
- [4] Brass, Peter. Advanced data structures. Vol. 1. Cambridge: Cambridge University Press, 2008.
- [5] Allig, Christoph, and Jörg Conradt. "Parallel Data Structures."