Volume 3, Issue 12, December 2013 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Study and Performance Analysis of Tree Scan Algorithm for Multi-Core Architectures

B. Muni Lavanya *
JNTUA college of Engineering,
Pulivendula, India

Abstract— In the field of trees and tree related researches, a tree scan algorithm is very important. It is also called as all-prefix-scan algorithm. The algorithm works in a prefix scan manner. The inputs given are – the array of elements and the operator which is to be applied on the elements during scan. The algorithm scans the array in such a way that, the prefix scan is done on the tree in which the operator given will be on each intermediate nodes and array elements as the leaf nodes. The general tree scan algorithm requires a total of work and depth, where n is no. of elements. By using the different parallel programming models, we may try to reduce the complexity of the algorithm and increase the performance. The comparison of the serial and parallel execution of the algorithm on different sample data is done here and analyzed. The paper highly concentrates on the performance increase of tree scan algorithm on using the parallel programming models like Open MP, MPI and Concurrent Java.

Keywords Tree Scan, Parallel Programming, OpenMp, MPI, concurrent java

I. INTRODUCTION

The tree scan algorithm that we are discussing is having multiple applications in different fields. In counting sort algorithm, which is an integer sorting algorithm, the prefix sums of a contiguous array of elements are used to calculate the position of each element in the newly created output array. The counting sort algorithm runs in a linear time for smaller values which are not greater than the total number of elements. A sorting algorithm, which is a faster one and is not restricted in the magnitude of the elements. Radix sort can be applied with the counting sort algorithm. This makes the sort more faster.[1].

II. RELATED WORK

Apart from the above listed application, the tree scan algorithm is having applications in other fields also. One such application is List ranking. This is used to transform a linked list into an array. While transforming from a linked list, we need to take care of the index of an element. This can be done by applying a prefix scan on the list or sequence and then map or assign each item to the position given by the prefix scan values. These are the main applications of tree scan algorithm. The scan result obtained by applying the prefix scan algorithm on a set of numbers given by Ω 0, Ω 1, Ω 2,… etc will provide us with a new set or sequence of elements β 0, β 1, β 2, …etc. The output elements are obtained by applying prefix scan on the input elements with a specified operation. The operation can be addition, subtraction, max () like anything.

$$\beta_0 = \Omega_0$$
$$\beta_1 = \Omega_0 \; op \; \Omega_1$$
$$\beta_2 = \Omega_0 \; op \; \Omega_1 + \Omega_2$$

OR

$$\beta_0 = \Omega_0$$
$$\beta_1 = \beta_0 \; op \; \Omega_1$$
$$\beta_2 = \beta_1 \; op \; \Omega_2$$

The ‘op’ given can be any algebraic or Boolean operation which are binary in nature.

The parallel implementation of the above discussed prefix scan or tree scan algorithm can be explained as below.[2][4][5]

1. First calculate the sums (assuming addition is the operator specified) of successive pairs of elements among which the first element will be having even index. : β0 = α0 op α1, β 1 = α2 op α3, etc.
2. Repeatedly apply the operator to get the new sequence γ0, γ 1, γ 2,.. of the old sequence β 0, β 1, β 2, ...
3. After finishing the recursion, expand the terms γ0, γ1, γ2... into two terms of the overall prefix sum: ε0 = α0, ε1 = γ 0, ε2 = γ 0 + α2, ε3 = γ 1, etc. After setting the first value, the following number εi is either copied from a location which is half way in the γ sequence, or given the value which is previously added to the corresponding value in the α sequence. Based on the size of the input array, the depth of recursion also increases. Only powers of 2, number of elements can be accepted as array. If the input elements are nth power of 2 in number, the recursion may go into a depth of . Normally, the number of steps needed for the execution is . But by using a parallel random access machine (PRAM)

the same parallel algorithm can be implemented with processes and also without any asymptotic slowdown. This is achieved by assigning different indices to each processor when there are more elements than the number of processors.[2]

III. TREE SCAN ALGORITHMS

The algorithms for tree scan and its sub functions are given here.

Algorithm for Main function

Input: p, where total number of elements = 2^p
Output: array applied by all-prefix scan
Method
Begin:

1. If (p<0)
   Input size=8
   Input ={2,8,3,-4,1,9,-2,7}
   Else
   For (i=0 to 2^p)
   Input input[i]
   Input size =2^p
2. Call result = Treescan(sum,0,input,input size)
3. For (i=0 to inputsize)
   Print result[i]
4. Call result = treescan(max,0,input,inputsize)
5. For (i=0 to inputsize)
   Print result[i]

End

*Description***:** The main function accepts the inputs and calls the treescan function with input parameters as arguments.

Algorithm for tree scan

Input: operator to be applied, identity value of operator, input array, input array size
Output: result of prefix scan
Method
Begin:

1. If inputsize==1
   Ret [0]=0
   return Ret
2. Count = input size/2
3. split_array(input,inputsize,odd,even)
4. For i=0 to count
   Temp[i]=op(even[i],odd[i])
5. S = treescan(op,identity,temp,count)
6. For i=0 to count
   Temp2=op(s[i],even[i])
7. z =interleave(s, temp2,count)
8. return z

End

*Description***:** The treescan function is recursively called until the limiting value of input size reaches 1. The entire array is divided into odd and even arrays and added toget a new array which applied by the recursion at each step. After recursion, it calls the interleave function for getting the prefix result

Algorithm for interleave

Input: two arrays a,b to be interleaved, size of each array(which is same)
Output: interleaved array
Method
Begin:

1. for i=0 to count
   Temp[2*i]=A[i]

Temp[2*i+1]=B[i]

2. return tmp

End

---

*Description***:** It interleaves the elements of two arrays in consecutive positions.

Algorithm for split_array

Input: source array, size of source array, destination array, position of elements to fill (odd or even)
Output: filled array
Method
Begin:

1. i=type

2. for j=0 to inputsize
dest[j]=input[i]
i=i+2

End

---

*Description***:** The filling of two arrays from source array such that one array contain all odd position elements and the other contains the even position elements.

IV. **FLOW DESIGN**

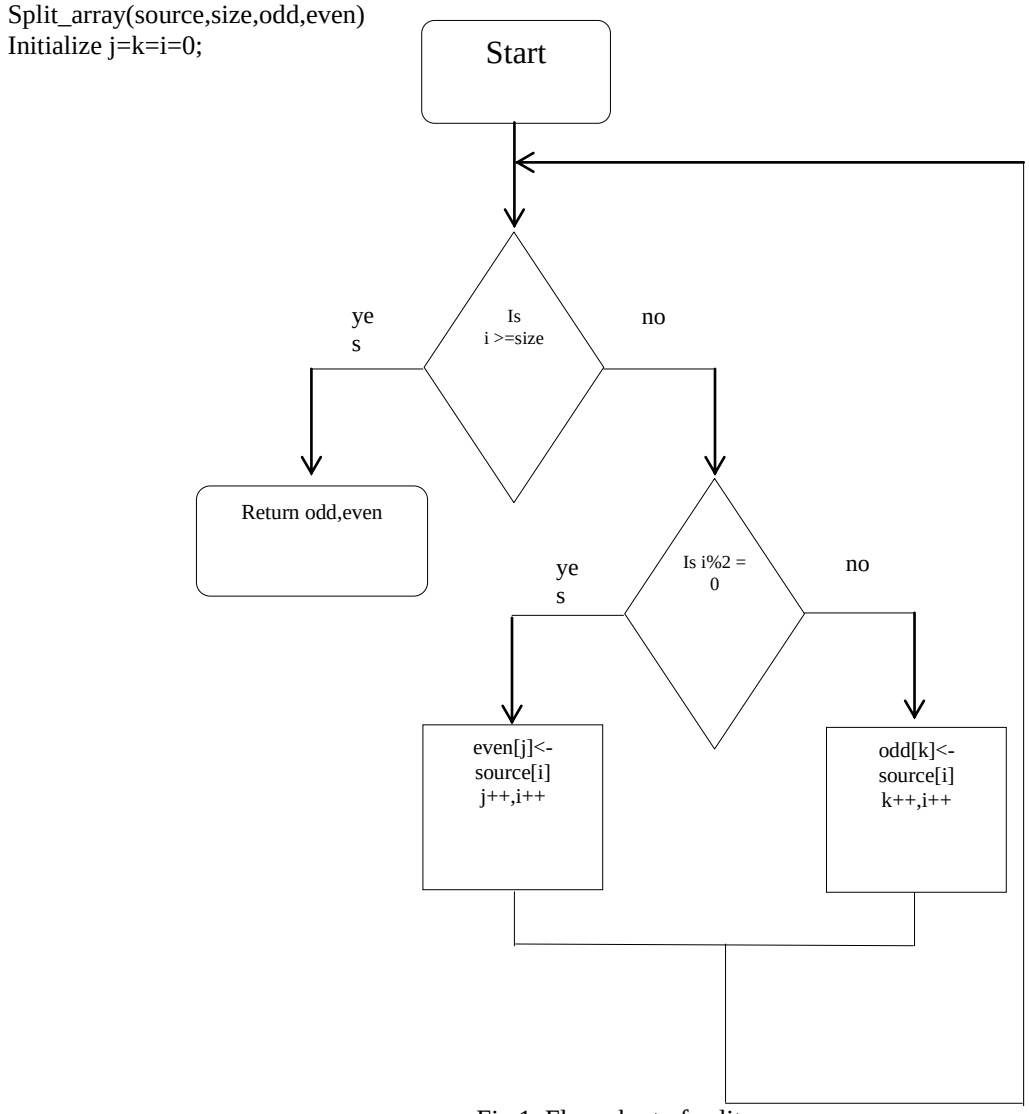

Fig 1. Flow chart of split_array

treescan (operator,identity,array,size) :

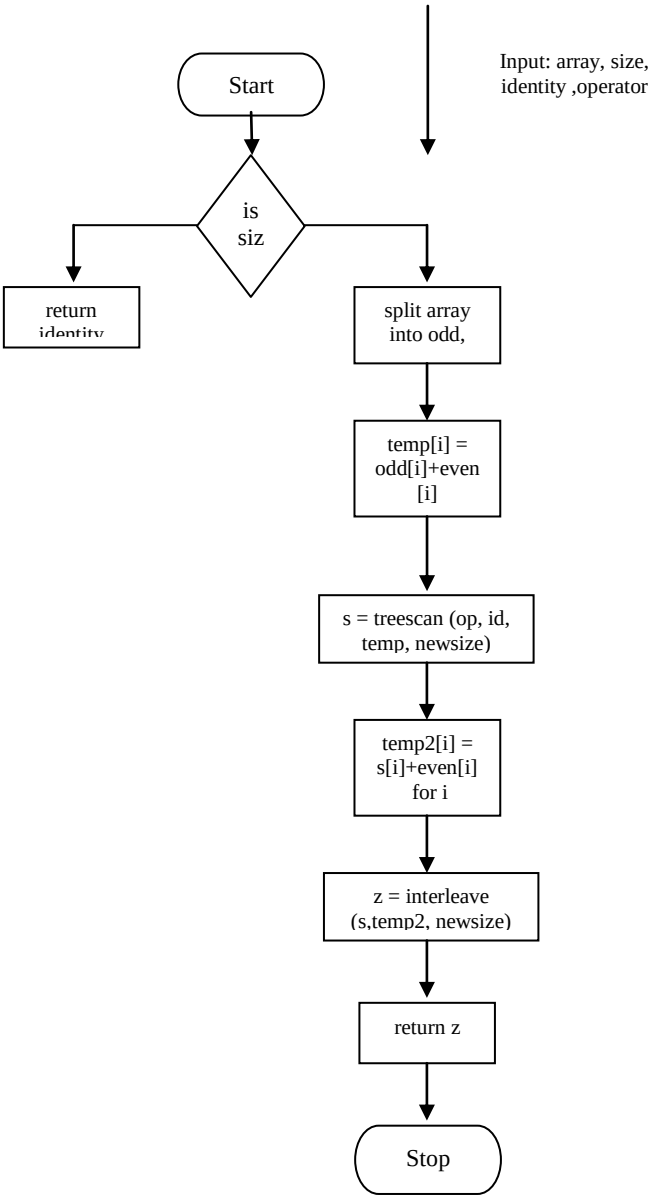

Fig 2. Flow chart of treescan

These are the flow charts for the algorithms discussed above. The data flow can be analyzed from the diagram and the region for parallelizing can be identified. The algorithm recursively calls itself until the size of the array becomes one. It return the identity value when the recursion stops. At each recursion , the size of array becomes the half.

V. IMPLEMENTATION

OpenMP: OpenMP is an industry-standard Application Program Interface which is used in parallel programming which is in fact programming on shared memories. It is a collection of compiler directives, environment variables and library files that are specific in parallel programming on shared-memory architectures. According to our algorithm, the serial code involves a recursive function and few data independent for loops, so we performed functional decomposition and little amount of data decomposition upon these loops. We have used the primitive #pragma omp parallel for to run the data independent for loops simultaneously. Functional decomposition is carried out on sum() function and max() function.

MPI: MPI, unlike OpenMP works based on message-passing between separate processes each of which carries out their own address space. MPI library specification helps the programmer to achieve these basic MPI operations on processes with the MPI standards. MPI is basically designed to perform well on parallel machines and workstation clusters. It's an uphill task to parallelize the serial program which is optimized well to perform fast. In this case we go through divide-and-conquer approach before performing any steps for parallelism. Similar to the one we did in OpenMP,

we perform functional decomposition on sum() function and max() function and a small amount of parallelism also achieved in data independent for loops. We were able to achieve a considerable amount of increase in performance using MPI primitive functions which is discussed in upcoming sections. Concurrent Java: Concurrency is the ability to run a program in parallel in such a way that the program in split into several parts and run simultaneously. As a result more the tasks are independent on the parts of a program more the throughput. In Java, the utility class concurrent which is a subclass of java.util serves this purpose. This class is available since J2SE 2.0. Here, for our case Executors frame work and Threadpools are used to perform parallelism. The fork join implementation is used so that, the filling function fillElements(), i.e., dividing the entire array into odd and even arrays can be parallelized.

VI. RESULTS

The execution time obtained for the different sample data are plotted for getting curves. The graph plotted between no.of jobs and execution time for parallel and sequential codes can be used to analyze the performance hike.

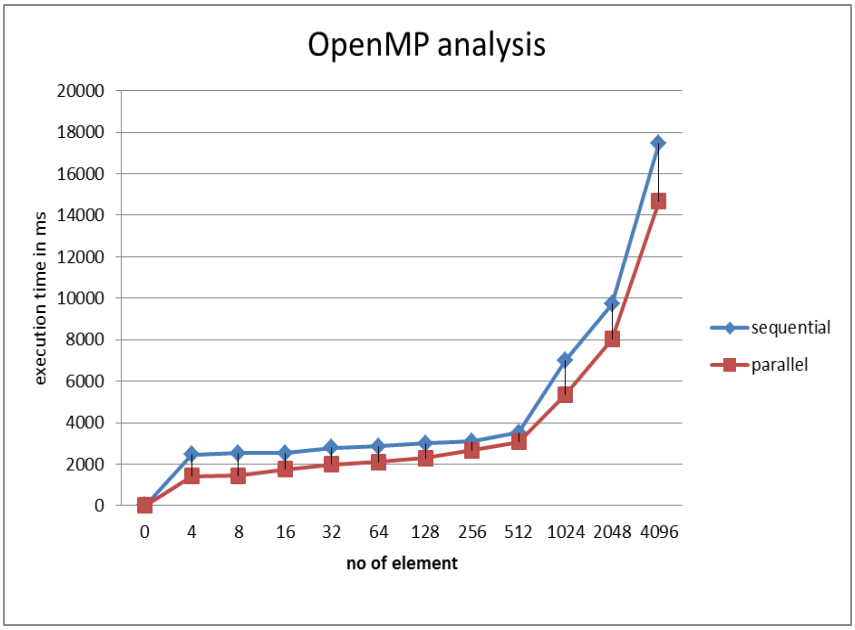

Fig.3 Open MP performance analysis- size of array Vs exec. time in milliseconds

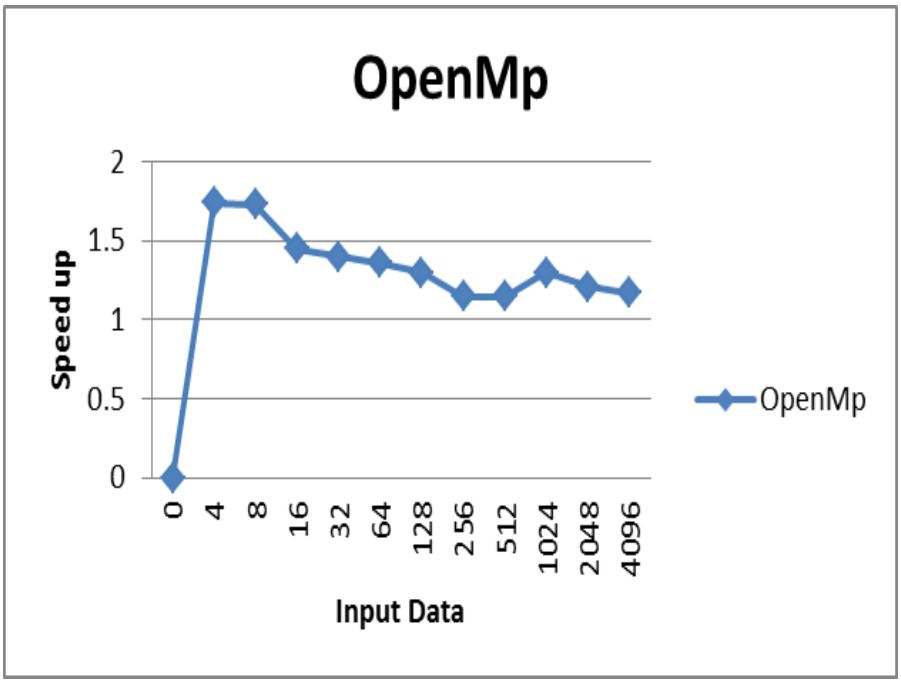

Fig 4.OpenMP speed up analysis

The execution time obtained for sample data are plotted and got a graph like below. The parallel program shows more efficiency than serial.

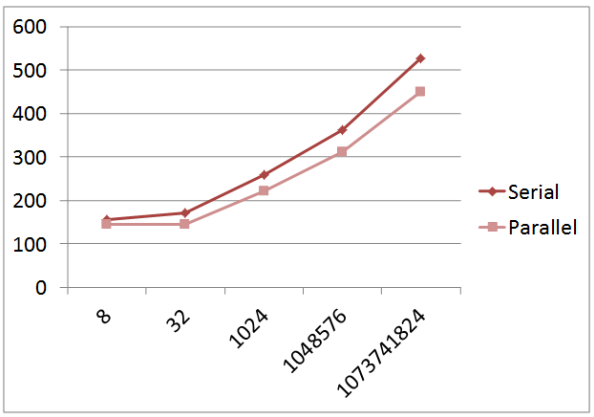

Fig 5.size of array Vs execution time in milliseconds

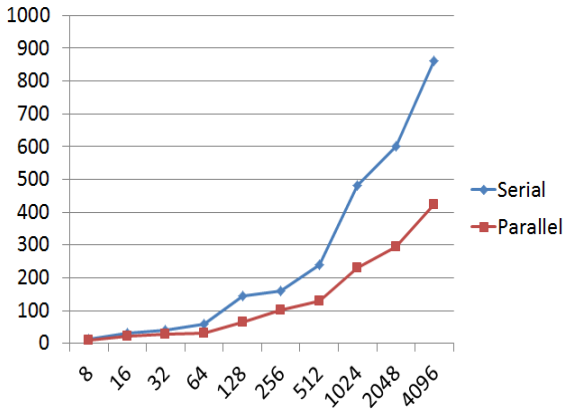

Fig 7. Concurrent Java – serial and parallel comparison

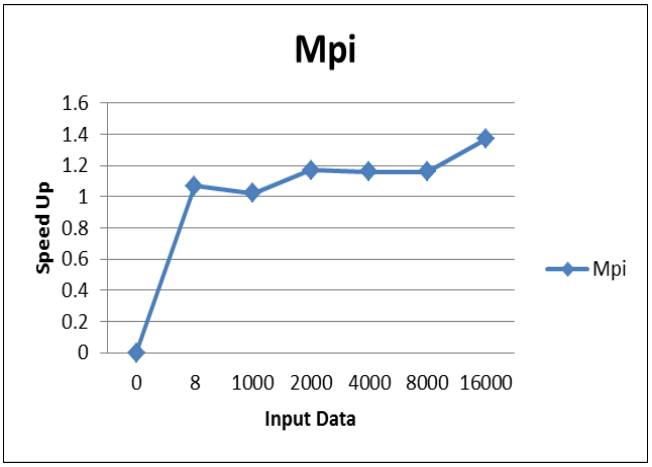

Fig 6. MPI speed up analysis

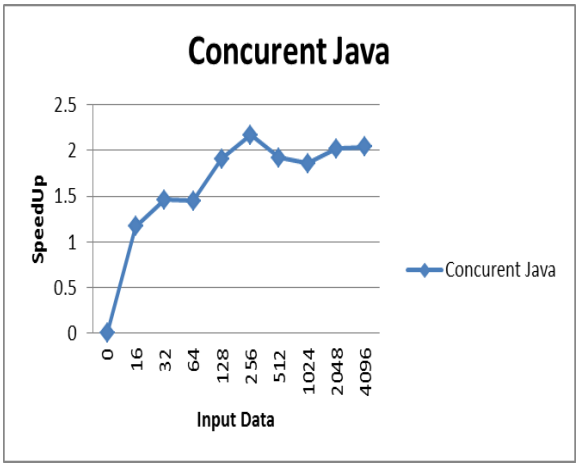

Fig 8. Cocurrent Java speed up analysis

The analysis of concurrent java parallel program with the sequential program execution times are shown in figure 8.

VII. CONCLUSIONS

The present study led us to the conclusion that, using the parallel programming models will efficiently reduce the total execution time of programs. The parallelized programs can be executed without any errors or side effects but with improved performance.The parallel programming model that we've used –OpenMP , MPI and Concurrent java are very powerful options in parallel programming.

REFERENCES

[1]. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001), "8.2 Counting Sort", *Introduction to Algorithms* (2nd ed.), MIT Press and McGraw-Hill, pp. 168–170, ISBN 0-262-03293-7.

[2]. Ladner, R. E.; Fischer, M. J. (1980), "Parallel Prefix Computation", *Journal of the ACM* 27 (4): 831–838, doi:10.1145/322217.322232, MR0594702.

[3]. Tarjan, Robert E.; Vishkin, Uzi (1985), "An efficient parallel biconnectivity algorithm", *SIAM Journal on Computing* 14 (4): 862–874, doi:10.1137/0214061.

[4]. Ofman, Yu. (1962), (in Russian), *Doklady Akademii Nauk SSSR* 145 (1): 48–51, MR0168423. English translation, "On the algorithmic complexity of discrete functions", *Soviet Physics Doklady* 7: 589–591 1963.

[5]. Khrapchenko, V. M. (1967), "Asymptotic Estimation of Addition Time of a Parallel Adder" (in Russian), *Problemy Kibernet.* 19: 107–122. English translation in *Syst. Theory Res.* 19; 105–122, 1970.

