International Information and Engineering Technology Association
Advancing the World of Information and Engineering

Revue d'Intelligence Artificielle
Vol. 37, No. 4, August, 2023, pp. 963-968
Journal homepage: http://iieta.org/journals/ria

# Parallelizing Depth-First Search for Pathway Finding: A Comprehensive Investigation

Vijayakumar Sangamesvarappa[1*], Vidyaathulasiraman[2]

[1] Department of Computer Science, Periyar University, Salem 636011, Tamilnadu, India
[2] Department of Computer Science, Government Arts & Science College (W), Bargur 635104, Tamilnadu, India

Corresponding Author Email: vijayviswak@gmail.com

https://doi.org/10.18280/ria.370417

**Received:** 18 March 2023
**Revised:** 25 July 2023
**Accepted:** 1 August 2023
**Available online:** 31 August 2023

***Keywords:***
*parallel processing, amdahl's law, searching algorithms, binary tree, speedup and efficiency of parallel algorithms*

## ABSTRACT

Search algorithms are integral to numerous applications in computer science. With the prevalence of multi-core processors in contemporary computing devices, the parallelization of search algorithms has surfaced as a viable strategy for achieving significant performance enhancements. This paper offers a detailed examination of the performance improvements garnered through the parallelization of search procedures, with a particular emphasis on the Depth-First Search (DFS) algorithm as it pertains to pathway discovery in binary trees. The primary aim of this study was to contrast the performance of the conventional sequential DFS approach with a novel parallel strategy designed to exploit the computational capabilities of multi-core processors. By capitalizing on the resources available in modern desktop and laptop computers, it was intended to markedly diminish the processing time necessary for examining all possible pathways in both symmetrical and asymmetrical binary trees. A meticulous experimental evaluation was conducted using a varied assortment of binary trees, spanning perfectly balanced to highly skewed structures, to ensure a thorough assessment of the effectiveness of both strategies. The primary metric employed for performance evaluation was the total processing time, a crucial consideration for time-critical applications. The experimental results confirmed the superiority of the parallelized method over the conventional sequential DFS approach. The parallel technique demonstrated significantly lower processing times for pathway discovery in all binary tree scenarios tested. These performance enhancements were particularly noticeable in larger and more complex trees, underscoring the potential of parallelization for managing computationally demanding tasks.

## 1. INTRODUCTION

Searching is an indispensable operation in computer science, especially when dealing with vast datasets. The efficient retrieval of specific items or data from such datasets can be a time-consuming endeavor. Fortunately, modern desktops and laptops are now equipped with multi-core processors, presenting an opportunity to expedite search tasks through the development of parallel algorithms. By identifying parallelizable segments within the search algorithm and distributing the workload evenly across available processors, significant reductions in processing time can be achieved [1-4].

The focus of this paper lies in addressing the challenging task of finding all paths in a tree and proposing a novel parallel algorithm tailored to harness the computational power of multi-core processors. The primary objective is to minimize the time required for this intricate search process, all without the need for additional hardware resources. To accomplish this, meticulous exploration of parallelization techniques is undertaken, aiming to strike a balance between effectively utilizing modern computing resources and efficiently exploring all paths in the tree.

One of the key challenges lies in identifying the parallelizable components within the search algorithm and ensuring proper load balancing across available processors. Through the optimization of resource utilization, the proposed parallel algorithm demonstrates its prowess in significantly reducing processing time compared to its sequential counterpart.

In this paper, we present comprehensive experimental validation to support the efficacy of the parallel approach. The results unequivocally show that the parallel algorithm outperforms the sequential alternative, thereby affirming its ability to minimize processing time without necessitating additional hardware investments.

The importance of parallelization in optimizing search operations cannot be understated. By tackling the problem of finding all paths in a tree and devising an efficient parallel algorithm, we showcase the potential of harnessing multi-core processors to achieve remarkable time reduction. As computer systems continue to evolve, the application of parallel algorithms represents a promising pathway towards enhancing search performance and addressing the challenges posed by ever-expanding datasets.

### 1.1 Finding all paths in a binary tree

From Figure 1 the possible paths from the root to terminals are $1\to2\to4$, $1\to2\to5\to6$, $1\to2\to5\to7$ and $1\to3$. The number of possible paths will be equal to Number of Terminal nodes in a tree. We know that in a binary tree a node will have maximum two child node. All the paths will start from the root it travels through left and right child up to the terminal node to find the path. Using DFS method we can find all paths of this tree. The time complexity of this sequential algorithm is O(n), where n is the total number of nodes in the binary tree.

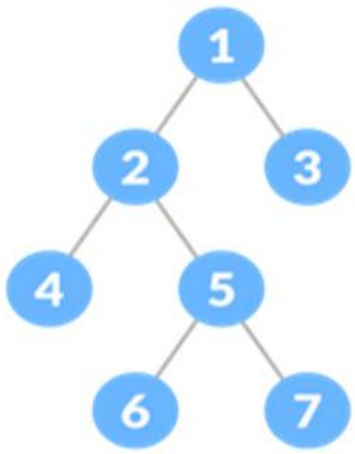


**Figure 1.** Binary tree

## 2. RELATED WORK

In recent years, search algorithms and their parallelization have been subjects of significant research interest in computer science. We review several relevant studies that have explored parallelization techniques for tree traversal and path finding, as well as investigations into load balancing and performance evaluation of parallel algorithms.

Sequential Algorithms for Tree Traversal:

Traditional sequential algorithms, such as Depth-First Search (DFS) and Breadth-First Search (BFS), have long been employed for tree traversal and path finding. Lai et al. [5] proposed an optimized recursive DFS algorithm with memory-efficient data structures for path finding in trees. Despite their widespread use, these sequential algorithms face limitations when dealing with large-scale datasets, motivating the need for parallel approaches.

Parallel Search Algorithms:

In the domain of parallel algorithms, there have been notable efforts to address tree traversal and related problems. Wodziński and Krzyżanowska [6] introduced a parallel BFS algorithm that utilizes a multi-threaded approach to achieve faster exploration of tree structures. Additionally, Uenoet al. [7] presented a parallel DFS variant employing task parallelism and synchronization techniques for graph traversal, demonstrating improved scalability on multi-core architectures.

Parallelization Techniques:

Researchers have explored various parallelization techniques that are relevant to our problem domain. Acar al. [8] discussed the challenges of load balancing in parallel graph search algorithms and introduced a work-stealing strategy for distributing tasks efficiently among processing units. Thwe and Kyi [9] proposed a hybrid parallelization method combining task-level parallelism and data-level parallelism to enhance the performance of search operations in complex data structures.

Performance Evaluation in Parallel Algorithms:

In evaluating the efficiency and scalability of parallel algorithms. Grama and Kumar [10] conducted a comprehensive study comparing parallel BFS, DFS, and A*search on multi-core processors. Their findings indicated that parallelization significantly improved the search throughput on various tree structures. However, load imbalance and communication overhead were identified as potential bottlenecks in certain cases.

Also Weiss [11] proposes the Amdhals effects in comparing the performance of parallel algorithm with the best sequential algorithm.

Applications of Parallel Algorithms in Tree Structures:

While parallel algorithms have been explored in various search domains, their specific application to tree structures has garnered interest. Riansantiet al. [12] proposed a parallel algorithm for finding all paths in a tree, using a combination of task-level and data-level parallelism. Their approach showed promising results on large-scale trees, providing insights into the potential advantages of parallelization.

The present work builds upon and extends the findings of these related studies. Our focus lies in developing a parallel algorithm specifically tailored for finding all paths in a tree, with a particular emphasis on load balancing and efficient task distribution. By addressing the challenges posed by tree structures and leveraging the computational capabilities of modern multi-core processors, we aim to achieve significant reductions in processing time while ensuring effective resource utilization.

### 2.1 Parallel approach in finding all paths in a binary tree

We can parallelize this algorithm by finding the path through left child and right child simultaneously using two processors. Two Functions called print_pathsl and print_pathsr will be executed simultaneously. The function print_pathsl will initiate to find the paths in left sub tree and the function print_pathsr will initiate to find all paths in a right sub tree. If these algorithms are executed simultaneously then our time complexity will be approximately O(n/2). This algorithm uses maximum two processors because we can split the tree as Left Sub tree and Right sub tree. But in our laptop or desktop more than two processors are available [9].

For implementing these functions we are using two single dimensional arrays called pathl and pathr which are used by the functions print_pathsl and print_pathsr respectively.

### 2.2 Parallel algorithm for finding all paths in a binary tree

Step 1: Declare a node with data, Left child address and Right child Address.

Step 2: Create a tree using the function

struct node* newnode (int data).

Step 3: Call the functions

Void print_pathsl (struct node*node) and Void print_pathsr (struct node*node) in parallel to process the left sub tree and right sub tree simultaniouly.

Step 4: Both the functions calls the Void print_paths_recur (struct node*node, int path [], intpath_len) recursively to print all the paths in left sub tree and right sub tree using the function void print_array (intints [], intlen).

**A. Declaration of a node as in Step 1.**

struct node {int data; struct node*left; struct node*right;};

**B. Function to Create a tree as in Step 2.**

struct node*newnode (int data) {struct node*node=(struct node*) malloc (sizeof (struct node)); node->data=data; node->left=NULL; node->right=NULL; return (node);}

**C. Function to store all the paths in left sub tree from the**

**root node to all terminal nodes in an array as in the Step 3.**

Void print_pathsl (struct node*node) {intpathl []; pathl [0]=node->data; print_paths_recur (node->left, pathl, 1);}

**D. Function to store all the paths in right sub tree from the root node to all terminal nodes in an array as in the Step 3.**

Void print_pathsr (struct node*node) {intpathr []; pathr [0]=node->data; print_paths_recur (node->right, pathr, 1);}

**E. A recursive function which findsall the path from given node as in Step 4.**

Void print_paths_recur (struct node*node, int path [], intpath_len) {if (node==NULL) return; path [path_len]=node->data; path_len++; if (node->left==NULL && node->right==NULL) {printf("\n"); print_array (path, path_len);} else {print_paths_recur (node->left, path, path_len); //recursively calls the left node of the tree print_paths_recur (node->right, path, path_len); //recursively calls the right node of the tree}}

**F. Function to print all the paths as in Step 4.**

Void print_array (intints [], intlen) {int i; for (i=0; i<len; i++) {printf ("->%d", ints [i]);} printf ("\n");}

In the main program create a tree by calling the function newnode. After creating the tree call the functions in parallel environment.

print_pathsl (root);
print_pathsr (root);

## 3. EXPERIMENTAL RESULTS

### 3.1 Example 1: Given a symmetric binary tree

A Symmetric binary tree is shown in Figure 2.

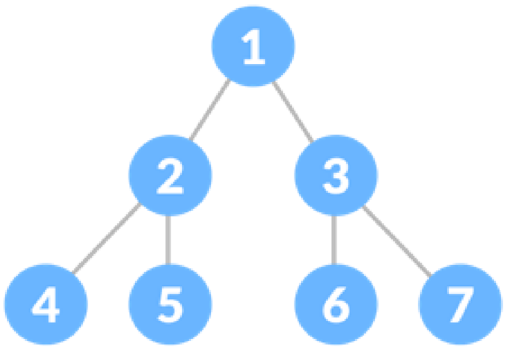


**Figure 2.** A symmetric binary tree

**Table 1.** Parallel execution of a balanced binary tree

| Steps | Processor 1 | | | | Processor 2 | | | | Paths |
|---|---|---|---|---|---|---|---|---|---|
| | Pathl | Node | Path_len | Terminal? | Pathr | Node | Path_len | Terminal? | |
| 0 | 1 | 1 | 1 | | 1 | 3 | 1 | | |
| 1 | 1 2 | 2 | 2 | | 1 3 | 3 | 2 | | |
| 2 | 1 2 | 4, 5 | 2 | | 1 3 | 6, 7 | 2 | | |
| 3 | 1 2 4 | 4 | 3 | | 1 3 6 | 6 | 3 | | |
| 4 | 1 2 4 | 4 | 3 | Y | 1 3 6 | 6 | 3 | Y | 1, 2, 4 & 1, 3, 6 |
| 5 | 1 2 4 | 5 | 2 | | 1 2 6 | 7 | 2 | | |
| 6 | 1 2 5 | 5 | 3 | Y | 1 2 7 | 7 | 3 | Y | |
| 7 | 1 2 5 | 5 | | | 1 2 7 | 7 | | | 1, 2, 5 & 1, 3, 7 |

**Table 2.** Parallel execution of an unbalanced binary tree

| Steps | Processor 1 | | | | Processor 2 | | | | Path |
|---|---|---|---|---|---|---|---|---|---|
| | Pathl | Node | Path_len | Terminal? | Pathr | Node | Path_len | Terminal? | |
| 1 | 40 | 2 | 1 | | 40 | 3 | 1 | | |
| 2 | 40 20 | 20 | 2 | | 40 60 | 60 | 2 | | |
| 3 | 40 20 | 10, 30 | 2 | | 40 60 | null, 80 | 2 | | |
| 4 | 40 20 10 | 10 | 3 | | 40 60 80 | 80 | 3 | | |
| 5 | 40 20 10 | null | 3 | | 40 60 80 | 80 | 3 | | |
| 6 | 40 20 10 | null | 3 | Y | 40 60 80 | null, 90 | 3 | | 40, 20, 10 |
| 7 | 40 20 10 | 30 | 2 | | 40 60 80 90 | null | 4 | | |
| 8 | 40 20 30 | 30 | 3 | | 40 60 80 90 | null | 4 | Y | 40, 60, 80, 90 |
| 9 | 40 20 30 | null | 3 | | | | | | |
| 10 | 40 20 30 | null | 3 | Y | | | | | 40, 20, 30 |

From Table 1 we can see that four possible paths of Figure 2 is 1→2→4, 1→3→6, 1→2→5 and 1→3→7 processed simultaneously by both the processors and terminated at the same time. In a **Symmetric** tree number of nodes and number of paths will be equal. Hence the load balancing in the symmetric tree will be perfect and time taken for left binary tree and right binary tree is same.

### 3.2 Example 2: Given an asymmetrical binary tree

In an asymmetrical binary tree, the nodes are arranged in such a way that one subtree is denser and deeper than the other. This imbalance can lead to varying path lengths from the root node to the leaf nodes in each subtree [13-15]. As a result, the time complexity of certain operations, such as searching for a specific element or finding all paths from the root to the leaves, can differ significantly depending on which subtree the element is located in.

Figure 3 shows an unbalanced binary tree in which the number of nodes and paths in left sub tree is need not be equal to the number of nodes and paths in the right sub tree.

From Table 2 we can see three possible paths 40→20→10, 40→20→30 and 40→50→80→90 processed simultaneously by both the processors and terminated at different time. We can also notice that Number of nodes in the left sub tree is equal to the right sub tree. But only difference is left sub tree has two terminals and right sub tree has one terminal. Hence the number of path in the left hand side is two and in the right hand side is one. So the time taken for unbalanced tree depends on height of a binary tree.

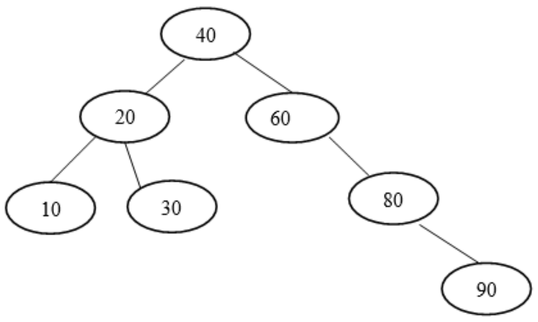


**Figure 3**. An asymmetrical binary tree

## 4. TIME COMPLEXITY OF PARALLEL ALGORITHM

### 4.1. Symmetricbinary tree

In a symmetric binary tree height of the tree can be ignored because all the terminals of the tree will be in the same height. So the time taken to travel from the root node to terminal node will be the same for all paths. For finding the all paths of a tree it is necessary to visit the entire node once. The time complexity of sequential algorithm to find all paths in DFS is O(|V|+|E|) where V is the Vertices and E is the edges in a binary tree. For the symmetric binary tree, the number of vertices in our Example 1 (Figure 2) is 7 and the edges are 6. Hence the time complexity of sequential algorithm is O (7+6) i.e., 13.

Figure 4 shows our parallel algorithm process the tree by starting in sequential and then by parallel. Hence the time taken for sequential portion is O (1+2) (1 vertices and 2 Edges) and time taken for one parallel portion is O (3+2) (3 vertices and 2 Edges). Hence the total time taken for the Example 1 is O (3+5) i.e., O (8).

Now the speedup by Amdahl's Law is [10, 11]:

$$\text{Speedup} = \frac{\mathit{Sequential\ Execution\ Time}}{\mathit{Parallel\ Execution\ Time}}$$

Now efficiency of the parallel algorithm is given as:

$$\text{Efficiency} = \frac{\mathit{Sequential\ Execution\ Time}}{\mathit{Parallel\ Execution\ Time\ X\ Processor\ used}}$$

Similarly, we can prove if the level of the binary tree increases the speed and efficiency also increases as shown in the Table 1 [10, 11].

Table 3 shows the speedup (1) and efficiency (2) of the parallel algorithm of Figure 4. Both the Speedup and efficiency [10, 11] are satisfying the Amdahl's Law. Figure 5 graphically represents the time taken for Sequential and Parallel algorithms.

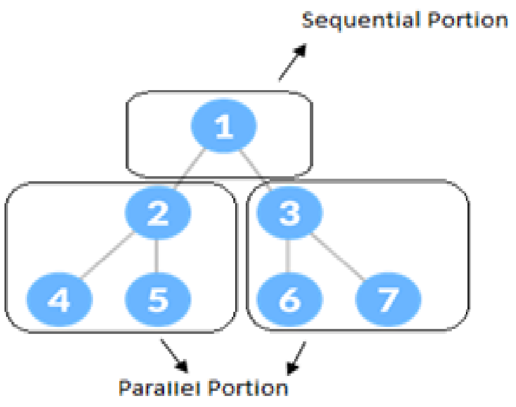


**Figure 4.** Sequential and parallel portions of balanced binary tree

**Table 3**. Speedup and efficiency of parallel algorithm for balanced binary tree

| S. No | Tree Levels | No. of Vertices | No. of Edges | Sequential Time | Parallel Time | Speedup (1) | Efficiency (2) |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 7 | 6 | 13 | 8 | 1.60 | 0.81 |
| 2 | 3 | 15 | 14 | 29 | 16 | 1.81 | 0.90 |
| 3 | 4 | 31 | 30 | 61 | 32 | 1.90 | 0.95 |

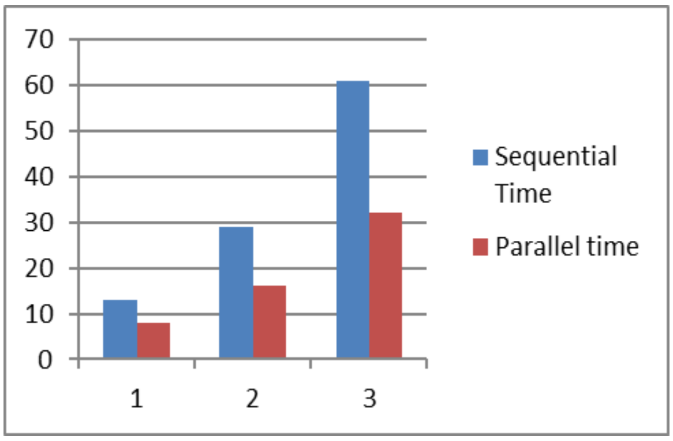


**Figure 5**. Comparing parallel and sequential time of balanced binary tree

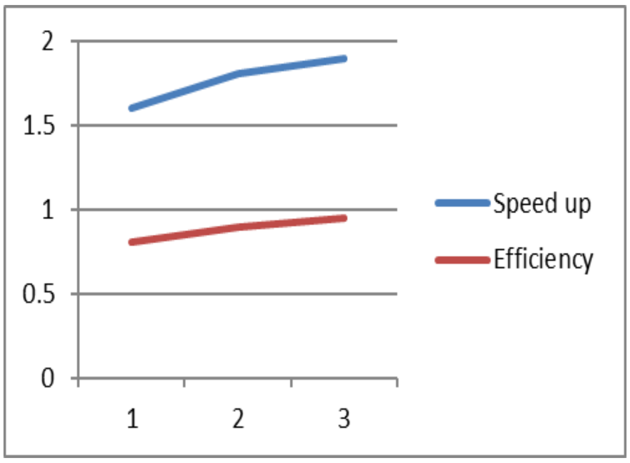


**Figure 6**. Speedup and efficiency of balanced binary tree

Figure 6 shows that the if the level of a balanced binary tree increases then the speedups and efficiency also increases.

A. Asymmetrical binary tree

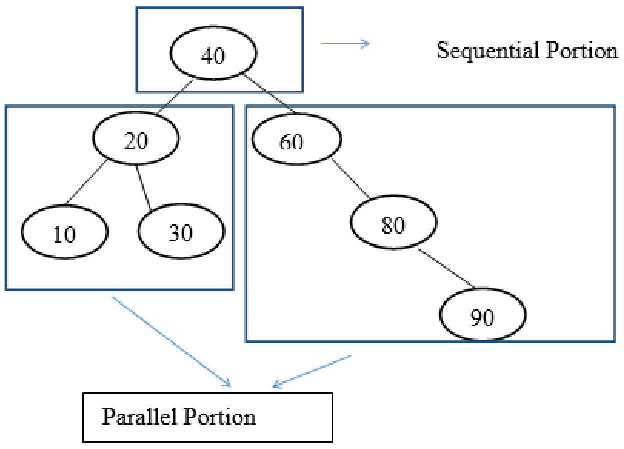
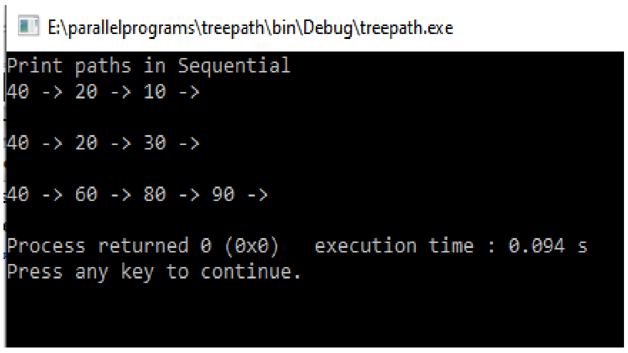


**Figure 7.** Sequential and parallel portions of Un balanced binary tree



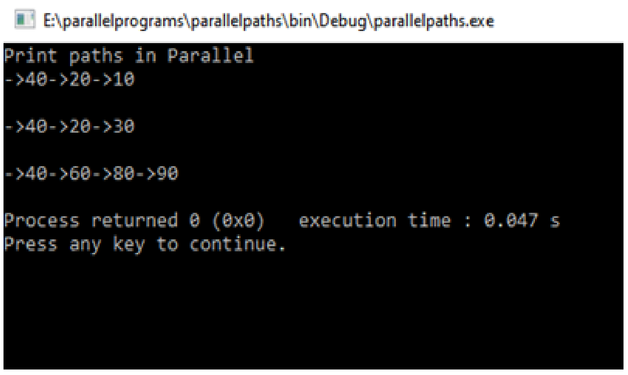
```
E:\parallelprograms\treepath\bin\Debug\treepath.exe
Print paths in Sequential
40 -> 20 -> 10 ->

40 -> 20 -> 30 ->

40 -> 60 -> 80 -> 90 ->

Process returned 0 (0x0)   execution time : 0.094 s
Press any key to continue.
```

**Figure 8**. Sequential program execution



```
E:\parallelprograms\parallelpaths\bin\Debug\parallelpaths.exe
Print paths in Parallel
->40->20->10

->40->20->30

->40->60->80->90

Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

**Figure 9.** Parallel program execution

In Figure 7, the height of left and right trees differs from each other. Note that number of edges and vertices are same in left sub tree and right sub tree. Height of left sub tree is 3 and height of right sub tree is 4. Hence the height is Max (3, 4)=4. Time complexity with branching factor and height of the tree for DFS is O (bm) [9] where b is he branching factor and m is the height of the tree. Hence the time taken for sequential algorithm is O (24)=16. From the Figure 7 time taken for parallel portion left tree is O (22)=4 and the parallel portion of the right sub tree is O (23)=8. Among these two parallel portions maximum of 8 is taken. Time complexity of sequential portion is O (21)=2. Therefore total time taken for parallel algorithm is 8+2=10. Table 4 shows the speedup and efficiency of the parallel execution of Figure 7.

**Table 4.** Speedup and efficiency of parallel algorithm for Un balanced binary tree

| Sequential Time | Parallel Time | Speedup (1) | Efficiency (2) |
|---|---|---|---|
| 16 | 10 | 1.6 | 0.8 |

From the Table 3 and Table 4 speedup and efficiency satisfies the Amdahl's Law. i.e., Amdahl's Law says that Criterium for Speedup (Sn) is 0<Sn<=n and Criterium for Efficiency (Es) 0<Es<=1. We can see that both the Tables 3 and 4 satisfies these criterium.

The OPENMP sequential and parallel programs were executed on Intel® Core (TM) i3-5005U CPU @2.00GHz (4 CPUs) machine using Code block software and gcc compiler with windows 10 operating system. The output of sequential program is shown in the Figure 8 and the output of the parallel program is shown in the Figure 9. An unbalanced tree in Figure 2 is given as input. From the Figures 8 and 9 the time taken for the parallel algorithm (0.047 Seconds) is less than the time taken for the sequential algorithm (0.094 Seconds).

## 5. CONCLUSION

This parallel algorithm reduces the time considerably for all types of binary trees. This algorithm uses maximum two processors because we can split the tree as Left Sub tree and Right sub tree. But in our laptop or desktop more than two processors are available. As a future enhancement this algorithm may be modified to utilize more than two processors. The same concept can also be applied for all types of DFS search algorithms like finding the path from source to destination in a tree.

## REFERENCES

[1] Naumov, M., Vrielink, A., Garland, M. (2017). Parallel depth-first search for directed acyclic graphs. In Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms, 1-8. https://doi.org/10.1145/3149704.3149764

[2] Sharma, R., Kumar, R. (2018). Design and analysis of parallel linear search algorithm. International Journal of Latest Trends in Engineering and Technology, 10(1): 35-38. https://doi.org/10.21172/1.101.06

[3] Al-Dabbagh, S.S.M., Barnouti, N.H., Naser, M.A.S., Ali, Z.G. (2016). Parallel quick search algorithm for the exact string matching problem using openMP. Journal of Computer and Communications, 4(13): 1-11. https://doi.org/10.4236/jcc.2016.413001

[4] Buluç, A., Madduri, K. (2011). Parallel breadth-first search on distributed memory systems. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 1-12. https://doi.org/10.1145/2063384.2063471

[5] Lai, X., Li, J.H., Chambers, J. (2021). Enhanced center constraint weighted a*algorithm for path planning of petrochemical inspection robot. Journal of Intelligent & Robotic Systems, 102: 78.

https://doi.org/10.1007/s10846-021-01437-8

[6] Wodziński, M., Krzyżanowska, A. (2017). Sequential classification of palm gestures based on a*algorithm and MLP neural network for quadrocopter control. Metrology and Measurement Systems, 24(2): 265-276. https://doi.org/10.1515/mms-2017-0021

[7] Ueno, K., Suzumura, T., Maruyama, N., Fujisawa, K., Matsuoka, S. (2017). Efficient breadth-first search on massively parallel and distributed-memory machines. Data Science and Engineering, 2: 22-35. https://doi.org/10.1007/s41019-016-0024-y

[8] Acar, U.A., Charguéraud, A., Rainey, M. (2015). A work-efficient algorithm for parallel unordered depth-first search. The International Conference for High Performance Computing, Networking, Storage and Analysis. https://doi.org/10.1145/2807591.2807651

[9] Thwe, P.P., Kyi, L.L.W. (2018). Performance comparison of parallel searching algorithms on the network of workstations. The Seventh National Conference on Science and Engineering, Mandalay Technological University, Upper Myanmar.

[10] Grama, A., Kumar, V. (1995). Parallel search algorithms for discrete optimization problems. ORSA Journal on Computing, 7(4): 365-385. https://doi.org/10.1287/ijoc.7.4.365

[11] Weiss, S. (2019). Chapter 6 performance analysis. CSci 493.65 Parallel Computing, Licensed under the Creative Commons Attribution-Share Alike 4.0 International License, 1-18.

[12] Riansanti, O., Ihsan, M., Suhaimi, D. (2018). Connectivity algorithm with depth first search (DFS) on simple graphs. In Journal of Physics: Conference Series, IOP Publishing, 948(1): 012065. https://doi.org/10.1088/1742-6596/948/1/012065

[13] Kaur, N., Garg, D. (2012). Analysis of the depth first search algorithms. Data Mining Knowl Eng, 4: 37-41.

[14] Akl, S.G. (1989). The design and analysis of parallel algorithms. Prentice-Hall, Inc.

[15] Schryen, G. (2022). Speedup and efficiency of computational parallelization: A unifying approach and asymptotic analysis. arXiv Preprint arXiv: 2212.11223. https://doi.org/10.48550/arXiv.2212.11223