Parallelization of Depth-First Traversal using Efficient Load Balancing

VSN Harish Rayasam*
IIT Madras, India
rayasam@cse.iitm.ac.in

Rupesh Nasre
IIT Madras, India
rupesh@cse.iitm.ac.in

***Abstract*—Graphs have widespread use in almost every field of study. Recent advances have parallelized several important graph algorithms such as computing the shortest paths, community detection, etc. However, simple depth-first traversal (DFT) is considered to be challenging for effective parallelization. Theoretically, DFT has been proven to be inherently sequential. This poses practical restrictions in implementing algorithms and applications that are naturally based on DFT. In this work, we propose a new method for depth-first traversal, which is amenable to high degree of parallelism. Unlike existing techniques, the algorithm provides good load balancing across workers by fair work-division across threads of a multi-core system.**

I. Introduction

Graph traversal is a systematic way to visit vertices in a graph and is a fundamental component in constructing many graph algorithms like identifying paths between two vertices, finding connected components, computing reachability, etc. The parallelization of these algorithms is critically dependent on effectively parallelizing the traversal. Breadth-first traversal has been explored both theoretically and experimentally, and has been shown to exhibit a good degree of parallelism (depending upon the graph structure). However, depth-first traversal (DFS) poses challenges, as it is not naturally amenable to divide-and-conquer regime. In fact, theoretically, DFS has been shown to be inherently sequential [1].

Reif [1] has shown that DFS is P-complete, suggesting that DFS is difficult to parallelize. There are solutions for certain special instances of the problem, for example, in planar graphs [2], but in the general case, DFS continues to be challenging: the best known algorithm is randomized and requires $O(log_7(n))$ parallel time using $O(n^{2.376})$ processors [3], which is far from being work efficient. The primary difficulty in parallelizing DFS is the ordering property that requires visiting the out-edges of a vertex in order. In many applications, such as reachability, graph search, and garbage collection [4], ordering is not necessary. Parallel DFS algorithms [3], [5], [6], [7] can be asymptotically work efficient, performing $O(|V| + |E|)$ work, where $|V|$ and $|E|$ are the number of vertices and edges respectively, when ignoring load balancing and scheduling costs. However, when the cost of scheduling operations is included in the analysis, all the known algorithms can incur large overheads.

In this paper we present a way of parallelizing Depth First Traversal (DFS) using simple and efficient load balancing scheme with small overheads in terms of scheduling and improve its running time on undirected graphs. Central to our technique lies a vertex labeling scheme that assigns an integral identifier to each graph vertex (which is different from the unique vertex id). We partition the graph evenly across available threads for traversal, and using the labeling, compute the DFS numbers for each vertex. The across-thread partitioning of the graph provides an almost perfect initial load-balancing. As the algorithm progresses, label-merging may alter the balance. Arbitrary graph structures such as cycles pose challenges for correct, yet load-balanced DFS. As long as the parallelism benefits in the initial phase outperform the imbalance overheads in the later part of the algorithm, we expect benefits out of our proposed scheme (which we observe to be significant on real-world graphs).

In particular, we make the following contributions.

- We propose a new divide-and-conquer technique for depth-first traversal of undirected graphs. We use a new vertex labeling and merging in an effective way to compute the final DFS number of each vertex.
- We show that the modified parallel DFS can be used in several applications including finding connected components and satisfying reachability queries.
- Using a set of real-world and synthetic graphs from SNAP [8], we illustrate the effectiveness of our approach. Our parallel DFS offers speedup ranging

Algorithm 1 DFS algorithm

```
1: procedure DFS(G)                          ▷ Performs DFS on Graph G
2:     for each v ∈ V do
3:         visited[v] = false
4:     for each v ∈ V do
5:         if !visited[v] then
6:             // Generate new label
7:             label[v] = newlabel
8:             EXPLORE(v)
```

Algorithm 2 DFS explore

```
1: procedure EXPLORE(v)
2:     // explores connected component of current vertex
3:     visited[v] = true
4:     previsit(v)
5:     for each edge e = (u,v) ∈ E do
6:         if !visited[u] then
7:             label[u] = label[v]
8:             EXPLORE(u)
9:     postvisit(v)
```

from 3.1x to 14.2x using 16 threads on real world graphs over its sequential counterpart.

II. Graph Traversal

Graph traversal is a systematic way to visit its vertices. There are several ways in which a graph traversal may be performed: breadth-first search, depth-first search (DFS), uniform cost search, etc. DFS is listed in in Algorithm 1. It starts traversing the graph at a designated start vertex $root$, explores its first unvisited child x, again explores x's first unvisited child, until a goal node is hit or there are no more children to explore. If the goal node is hit, the procedure may terminate (based on the application requirements). In the other case, backtracking is used to return to the last not-yet-fully-explored vertex.

Running time of DFS is linear in terms of the graph size. DFS can be naturally implemented using recursion (as shown in Algorithm 2 where function `EXPLORE()` calls itself).[1]

DFS parallelization is challenging because the graph connectivity could be arbitrary and cannot be predicted a priori. For instance, if we divide a graph into two equal parts and assign $N/2$ vertices to each thread, it is unclear what label should the first vertex in the second partition be given, as it depends upon whether the vertex is connected to another vertex in the first partition or not. Finding that itself would involve graph traversal!

Our parallelization technique crucially relies on a vertex labeling scheme, which enables us to identify some structural properties of a vertex's neighborhood.

[1]However, for big real-world graphs, recursive DFS quickly grows out of stack. So we use an iterative DFS with an explicit stack.

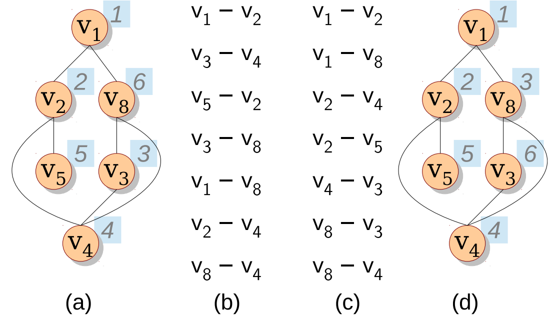

Figure 1. (a) An example graph and its FCFS vertex labeling (b) The example graph represented as an edge-list (c) The same graph with a differing ordering of the edge-list (d) The modified vertex labeling

The labeling helps our algorithm perform load-balanced task-distribution in a later step.

III. Vertex Labeling

Vertex labeling is a standard procedure which takes edge list of a graph as input and generates a label for each vertex which has atleast one neighbor in the graph (that is, if a vertex is part of atleast one edge in the edge list).

A. FCFS Vertex Labeling

A First-Come-First-Served (FCFS) vertex labeling approach numbers the vertices in the range $[1..|V|]$, where $|V|$ is the number of vertices in the graph. The numbers are assigned based on the order in which the edges are processed by the labeling algorithm (presented in Algorithm 3). As an example, consider the graph shown in Figure 1(a) containing six vertices, and edge-list as shown in Figure 1(b). The vertex labels computed are shown beside each vertex in the figure.

We use a data-structure `LabelMap` which is an abstract map containing key-value pairs, where keys are vertices and values are the labels. Based on the application scenario, one may wish to use a different implementation of LabelMap. For instance, using a binary search tree provides an $O(log_2 n)$ lookup-time, whereas a minimal-collision hash-table may provide a constant-time lookup.

The FCFS labeling algorithm goes through the following steps for the example in Figure 1.

1) As v_1 is not present in LabelMap we map v_1 to 1 and since v_2 is not in LabelMap we map v_2 to 2.
2) As v_3 is not present in LabelMap we map v_3 to 3 and since v_4 is not in LabelMap we map v_4 to 4.

Algorithm 3 LABEL

```
1: procedure LABEL(EdgeList E)
2:     // Labels the vertices based on edges information
3:     // Edge List contains edges of form (u, v)
4:     // LabelMap maps each vertex to a label
5:     counter = 0;
6:     for each edge e = (u, v) ∈ E do
7:         if !LabelMap.hasKey(u) then
8:             // Generate new label for u
9:             LabelMap.put(u, ++counter)
10:        if !LabelMap.hasKey(v) then
11:            // Generate new label for v
12:            LabelMap.put(v, ++counter)
```

3) As v_5 is not present in LabelMap we map v_5 to 5 and since v_2 is in LabelMap we skip it.
4) As v_3 is in LabelMap we skip, and since v_8 is not in LabelMap we map v_8 to 6.
5) As v_1 is in LabelMap we skip, and since v_8 is in LabelMap we skip.
6) As v_2 is in LabelMap we skip, and since v_4 is in LabelMap we skip.

We may not label vertices with zero degree. The labeling above is very naïve and it cannot exploit the neighborhood structure. FCFS labeling may result in arbitrary labels to the graph vertices depending upon the edge-order in the input.

B. Our Approach: Clustered Labeling

In our approach, we cluster the edges and then reuse the existing FCFS ordering to result into an improved labeling. If the edges are given in such a way that all edges corresponding to a vertex are clustered together in the input then we can generate a better labeling. Thus, if a vertex v_i gets a label k and if v_i has m neighbors then all the labels of the neighbors will be in the range $k+1$ to $k+m$. Using such a clustered labeling has the advantage that we can clearly differentiate whether any particular target vertex falls into the current neighborhood by simply checking whether *currentlabel + maxdegree of graph* $\geq$ *target label*. This property forms the basis for our load-balanced DFS traversal.

For instance, consider an alternate edge ordering as shown in Figure 1(c). Here, the edges are clustered according to the source node ids. The FCFS labeling on this ordering results in the vertex labels as shown in Figure 1(d). Note that unlike the FCFS labeling, the clustered labeling mostly follows the connectivity of the vertices. Thus, for instance, neighbors of vertex v_1 are labeled 2 and 3, while vertices v_5 and v_3 are labeled away from the label of v_1. Clustered labeling helps us achieve better load balancing for parallel execution.

Algorithm 4 ParallelTraverse

```
1: procedure PARALLELTRAVERSE(Graph G)
2:     Compute Clustered Labeling
3:     Compute LoadBalanced Indices
4:     P = Runtime.availableProcessors();
5:     for i in 1..P do
6:         // DFSTask is an instance of Thread
7:         DFSTask i = new DFSTask();
8:         //set Thread i to start at loadbalanced index i
9:         i.setStart(loadbalanced[i]);
10:        i.start();
11:    // Wait for all threads to finish
```

Algorithm 5 Per thread processing

```
1: procedure RUN()
2:     // This method is invoked by each thread
3:     // independently
4:     // Start DFS procedure at specified start index
5:     // Assign next unique label
6:     label[start] = nextUniqueLabel()
7:     dfs(start);
8:     while unvisitedVertices.size() > 0 do
9:         v = nextUnprocessedVertex()
10:        label[v] = nextUniqueLabel()
11:        DFS(v);
```

IV. Algorithm for Load Balanced DFS

Once the vertices are labeled, we can start the parallel DFS execution. One naïve way to parallelize DFS is by assigning each unvisited vertex to a new thread. Unfortunately, this confines threads to the same region of the graph – increasing the synchronization requirements. It would be ideal to be able to divide the graph into chunks and assign each chunk to a thread. We calculate the load factor based on the number of vertices in the graph $|V|$ and the number of processing elements (threads) available P. Thus, we assign $|V|/P$ number of vertices to each thread achieving an almost perfect initial load-balancing.

Algorithm 4 shows the overall processing of the master thread. It first computes the clustered labels as discussed in Section III-B, and then finds partitions of the input graph to be processed by threads. It then invokes a worker from the thread-pool on each of the partitions (`for`-loop in Algorithm 4). Each thread executes the procedure listed in Algorithm 5. For the graph partition assigned to the thread, each worked thread starts DFS at the first node in the partition. Each unvisited vertex is assigned a new unique label.

Algorithm 6 lists the DFS traversal using an explicit stack (for memory efficiency and performance). It is similar to the standard DFS except for assignment and management of vertex labels. If vertex v_i has been assigned a label l_i and suppose we are able to reach v_i from vertex v_j with label l_j while performing dfs,

Algorithm 6 DFS traversal

```
1: procedure DFS(v)
2:     //Start DFS procedure at the specified start index
3:     Stack stack;
4:     stack.push(v);
5:     while !stack.empty() do
6:         curr = stack.pop();
7:         visited[curr] = true;
8:         for each neighbor w of curr do
9:             if !visited[w] then
10:                visited[w] = true;                    ▷ Early marking
11:                //Avoids other threads pushing
12:                //this vertex to their stacks
13:                labels[w] = labels[curr];
14:                stack.push(w);
15:            if visited[w] then                        ▷ Merge Labels
16:                // visited before curr is visited
17:                // Merge labels
18:                L_1 = labels[w]
19:                L_2 = labels[curr]
20:                labelEquivMap[L_1].add[L_2]
```

Algorithm 7 Merging of vertex labels

```
1: procedure LabelMerge()
2:     // Merge equivalent labels after DFS
3:     WeightedQuickUnionPathCompressionUF uf;
4:     for label in labelEquivMap.getLabels() do
5:         for otherLabel in labelEquivMap[label] do
6:             uf.union(label, otherLabel);
7:     // Now unique labels refer to final
8:     // number of connected components
```

we merge the two labels (Line 20 of Algorithm 6). This signifies that all the vertices in the connected component of v_i will also be in the connected component of v_j.

Procedure LabelMerge (listed in Algorithm 7) is used to unify equivalent labels using union-find data structure. The implementation is optimized by performing union by size and path-compression. After merging all the equivalent labels, the number of unique labels determines the number of connected components in the input graph. We also optimize our parallel implementation with *early marking* of vertices (Line 10) within stack; it prevents the same branches of vertices being visited by other threads, which gives considerable improvement in running time, and also reduces the number of labels generated. Similar to other parallel and incremental DFS algorithms, our parallel DFS does not guarentee that the visiting order would be the same as in the sequential DFS. However, it guarentees to generate a valid DFS spanning tree of the input graph.

V. Applications

Effective parallelization of DFS improves performance of several graph applications. In particular, we have applied our proposed approach to improve the query times for reachability, path queries, computation of the number of connected components and the size of each connected component.

Connectivity. Once our DFS traversal is performed, we can answer reachability queries in $O(1)$ time. If we wanted to test whether two vertices u and v are connected in the input graph, we simply need to check the labels assigned to them by the parallel DFS procedure. If both the labels are equivalent then u and v are connected, else they are guaranteed to be in different components. This is because, if they are connected, at some point during the traversal in Algorithm 6, the labels would be propagated from u to v (or vice versa) invoking the label merging at Line 20. On the other hand, if there were no connectivity between u and v, their initial and final labels would continue to remain different, as their labels would never be merged.

Connected Components. Our DFS traversal maintains unique labels for each connected component. The maintenance is achieved using an optimized union-find data structure. Thus, the number of connected components can be easily deduced as the number of unique labels at the end of the traversal.

Size of Connected Components. Efficient maintenance of union-find data structure necessitates union-by-size, which requires tracking the size of each component. The counts are added on a label merge. Thus, our data structure directly has each connected component's size, which can be returned for a label (or a vertex).

Path Queries. To answer path queries, we maintain parent information for each vertex in the DFS tree. Thus, if the query is to find a path between vertices u and v, we first check if u and v are in the same connected component, and if yes, then traverse their parents to check for the possibility of a path between them.

VI. Experimental Evaluation

We evaluated our parallel DFS algorithm on a 64-bit Intel(R) Xeon(R) 32-core CPU with 2.60 GHz clock and 100 GB RAM running CentOS 6.5 with 2.6.32 kernel. We experimented with several real world graphs from SNAP dataset [8], listed in Table I. We also used a synthetic tree input, as trees are more amenable to effective parallelization. The graph sizes range over three orders of magnitude. We present results as an average of ten independent runs on each graph.

Figure 2 shows the speedup achieved by our parallel DFS over the sequential DFS on our set of input graphs. We observe that DFS *is* amenable to parallelism using our load-balancing methodology. e.g., for AS-Skitter graph, our parallel DFS achieves a moderate speed-up of

Graph	#Vertices	#Edges
roadNet-CA	1,965,206	5,533,214
com-Youtube	1,134,890	2,987,624
CA-AstroPh	18,815	396,160
AS-Skitter	1,696,415	11,095,298
roadNet-PA	1,088,092	3,083,796
Tree	1,000,000	999,999
CA-CondMat	23,133	93,497

Table I
INPUT GRAPHS

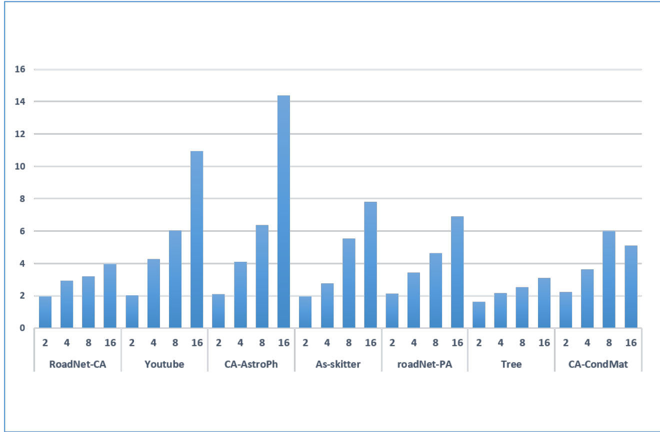

Figure 2. Speedup over sequential

7.8 using 16 threads. This is due to better load-balancing and less overheads in terms of task-scheduling.

Figure 3 shows the number of labels generated by the algorithm on various input graphs. In practice, the number of labels depends both on the size of the graph as well as the connectivity. We observe that the number of lables is influenced by the graph size. For our graph with the highest number of edges (AS-Skitter with 11 million edges), the number of labels generated is considerably large (over 1000). The divide-and-conquer algorithm

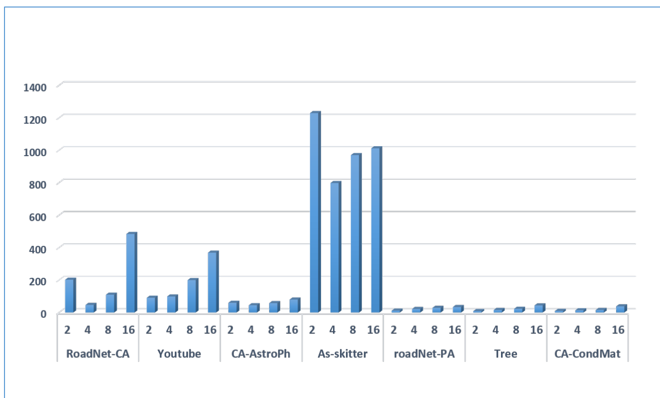

Figure 3. Number of labels generated

is amenable to generating a large number of labels. This also explains why the number of labels generated increases with the number of threads. However, early marking technique (see Algorithm 6) diminishes the danger of label explosion. Reduction in the number of labels leads to fewer concurrent accesses and updates to the underlying map used for maintaining the labels. This helps improve performance. On the other hand, larger the number of labels, the graph is more amenable to parallel processing.

VII. CONCLUSION

In this work, we demonstrated a practical way to parallelize DFS traversal by clustered labeling, which allows load-balanced task-distribution across workers. We illustrated that the parallel DFS performs much better compared to its sequential counterpart achieving an speedup in ranges of 3.1x to 14.2x using 16 threads on different real world graphs. We also showed how several graph applications such as finding connected components, path querying get benefitted by improving DFS running time. In future, we would like to explore more graph applications to assess its effectiveness.

REFERENCES

[1] J. Reif, "Depth-first search is inherently sequential," *Information Processing Letters*, vol. 20, no. 5, pp. 229–234, 1985.

[2] T. Hagerup, "Planar depth-first search in o(log n) parallel time," *SIAM J. Comput.*, vol. 19, no. 4, pp. 678–704, Jun. 1990. [Online]. Available: http://dx.doi.org/10.1137/0219047

[3] A. Aggarwal and R. Anderson, "A Random NC Algorithm for Depth First Search," in *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, ser. STOC '87. New York, NY, USA: ACM, 1987, pp. 325–334. [Online]. Available: http://doi.acm.org/10.1145/28395.28430

[4] R. Jones, A. Hosking, and E. Moss, *The Garbage Collection Handbook: The Art of Automatic Memory Management*, 1st ed. Chapman & Hall/CRC, 2011.

[5] P. Varman and K. Doshi, "Improved Parallel Algorithms for the Depth-first Search and Monotone Circuit Value Problems," in *Proceedings of the 15th Annual Conference on Computer Science*, ser. CSC '87. New York, NY, USA: ACM, 1987, pp. 175–182. [Online]. Available: http://doi.acm.org/10.1145/322917.322945

[6] Y. H. Tsin, "Some Remarks on Distributed Depth-first Search," *Inf. Process. Lett.*, vol. 82, no. 4, pp. 173–178, May 2002. [Online]. Available: http://dx.doi.org/10.1016/S0020-0190(01)00273-3

[7] A. Laarman, R. Langerak, J. Van De Pol, M. Weber, and A. Wijs, "Multi-core Nested Depth-first Search," in *Proceedings of the 9th International Conference on Automated Technology for Verification and Analysis*, ser. ATVA'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 321–335. [Online]. Available: http://dl.acm.org/citation.cfm?id=2050917.2050942

[8] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.