# CS406 Project-Parallel Computing

Ilgın Şara Hacipoğlu, Buse Ak

June 2021

Abstract

Depth Limited Search is one of the most used Graph Traversal algorithms to detect cycles in graphs. The Depth First Search algorithm has advantageous properties such as space complexity. However if a wrong branch is chosen to expand, the termination may not be possible. Therefore, throughout this project, depth limited search with a certain depth algorithm, where the depth is the desired cycle length, is applied to increase efficiency in terms of space and time complexity. While implementing Depth Limited Depth First Search algorithm, CPU parallelization techniques are utilized using the OpenMP library.

## 1.Introduction and Problem Description

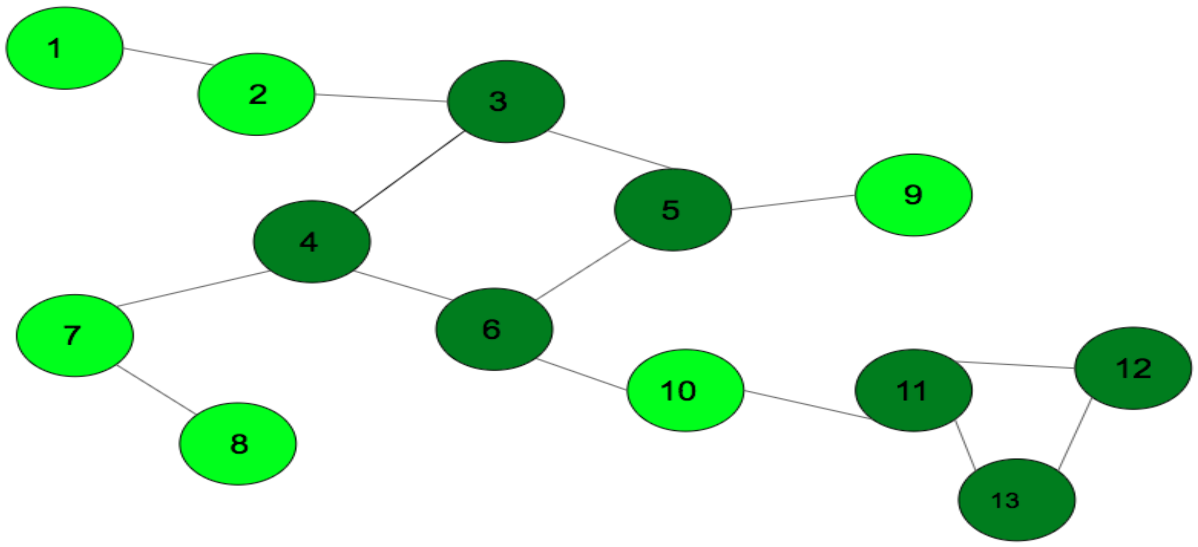


*Figure 1.0 Graph Traversal*

The problem is finding the number of length k cycles containing the vertex u for each u in vertex set V, given an undirected graph G=(V,E), and a positive integer 2<k<6. If the length k is taken as an input and not known because of it, the problem becomes NP-complete but in this project, the cycle length k is given, so the problem can be solved in O(VE) time.

For example with the graph example given above, when k=4 input is given the program will find the cycle 3-4-5-6-3 and increment the 4 length-cycle count for each of the vertices u in the given path, similarly when k=3 is given as input the found cycle will be 11-12-13-11 and program will increment cycle count for each of the respective vertices present in the path.

In this project, a recursive DFS algorithm and Depth Limited Search idea has been utilized to find simple small cycles of given length. Depth-Limited Search algorithm is in fact very similar to DFS with a predetermined limit. With this algorithm, the node at the depth limit will be treated as if it has no further successors so that traversing further down the graph is avoided if the cycle length threshold is reached. As this will be pruning the adjacency search, we anticipate a significant reduction in runtime. Upon this approach, a CPU paralleization with OpenMP has been performed in an attempt to increase throughput.

## 2.Data Preprocessing

The program will be run with two inputs from the user, with the inputs file path and the k respectively. The input file structure is such that for each ui<vi the edge source and destination node is given. Therefore, the reverse orientation (for any ui and vi pair, vi and ui creation) is also added before creating the CRS structure from the adjacency list.

## 2.1 Compressed Row Storage (CRS)

CRS uses to arrays; first array known as the adj array holds all the contents of the adjacency list appended row by row, the second array known as the xadj array corresponds to index in adj array where vertices have an edge from the given vertex start.

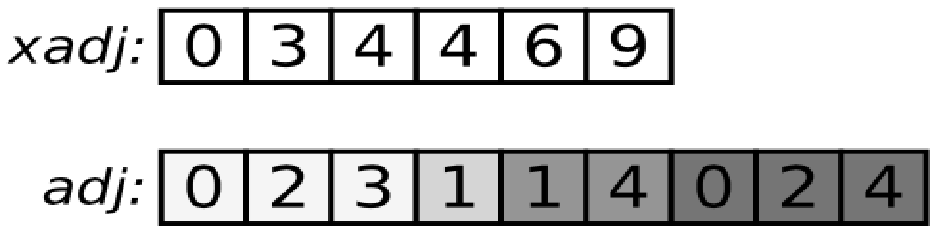


*Figure 2.1.0 CSR Representation of a graph*

If xadj[i]= k and xadj[i+1]=k+3 for instance, it implies that there is an edge from vertex i to all vertices stored in adj array starting from adj[k] to adj[k+2].

## 3. Other Attempted Solutions

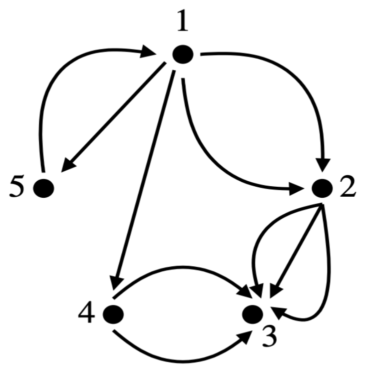
### 3.1 General BFS Approach (Breadth-First Search)

There are 3 main BFS approaches known as Top-down, Bottom-up and Hybrid heuristic, where Bottom-Up and Top-down approaches are combined based on vertex count in a specific frontier. Nodes in the same level are pushed into a frontier and upon iteration over those present in the frontier, next frontier is created for the upcoming levels. One drawback of BFS algorithm is that each frontier of the tree needs to be saved into memory to expand into the next levels. This will require significant time, especially if cycles are located farther down the graph as the algorithm will need to proceed into much lower levels and might even need to traverse the entire graph. Although literature search suggests that a general BFS algorithm is more suitable for parallelization than a general DFS, given the nature of the problem and graph sizes, it did not appear feasible to continue with this approach.

## 3.2 Graph Coloring

The general approach with this algorithm is to use a DFS algorithm to mark all vertices with unique colors(or numbers) and push similarly marked vertices into an adjaceny list once the graph traversal is complete to find out unique cycle paths. For this there will need to be 3 groups of coloring: partially visited, visited and non-visited. Whenever a partially visited vertex is encountered along the adjacency trace to find a cycle, algorithm needs to backtrack until the start vertex mark all the vertices along the process with cycle number and increment cycle count for each of these vertices. This algorithm worked will during our trials with smaller sample graphs however managing backtracks, coloring and cycle numbers became overly complicated with very large graphs and parallelized methods.

## 3.3 Powers of Adjacency Matrix



$$A = \begin{bmatrix} 0 & 2 & 0 & 1 & 1 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} \quad A^2 = \begin{bmatrix} 1 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 1 & 1 \end{bmatrix} \quad A^3 = \begin{bmatrix} 0 & 2 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 8 & 0 & 0 \end{bmatrix}$$

*Figure 3.3.0 Directed Graph and Matric Multiplication*

Given an adjacency matrix A, A^k matrix provides a count of a number of length k walks between vertices i->j in directed graphs. We tried to utilize this idea such that diagonals of A^k give walks of length k each vertex i is included in (starts in i ends in i). This approach seemed very viable initially, as we thought we could directly compute matrix-matrix multiplication k times in parallel using the CRS structure efficiently. Matrix multiplications are in fact easier to parallelize than recursive algorithms like DFS however could also be very memory intensive, creating yet again a bottleneck.
All things considered, this approach was problematic, as the definition of "walk" of k length from same vertex i to i does not necessarily imply a cycle as a walk can backtrack to a previously visited node via an already used edge. Some trial efforts with smaller graphs yielded wrong results so we moved away from this approach as well.

## 4. Our Approach: Depth-Limited DFS

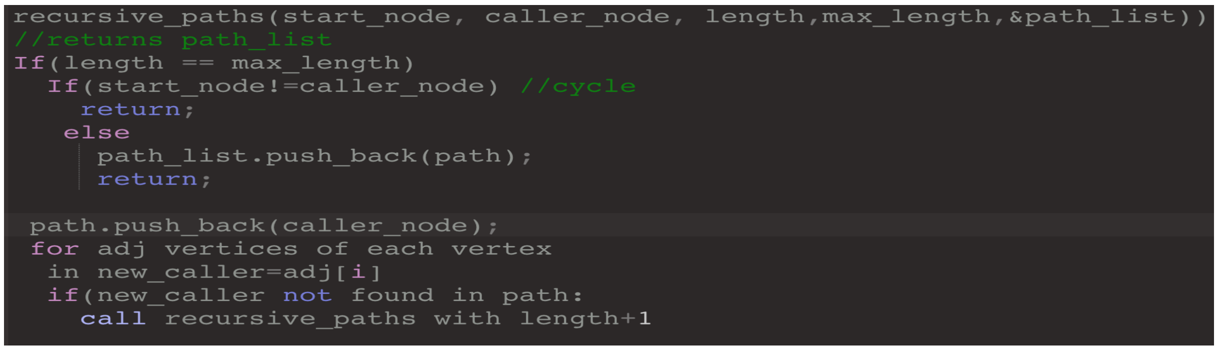
For each vertex, the adjacent nodes are traversed until a certain depth(k) to check cycle existence. With this algorithm, the node at the depth limit will be treated as if it has no further successors so that traversing further down the graph is avoided if the cycle length threshold is reached. As this will be pruning the adjacency search, we anticipate a significant reduction in runtime.

A recursive DFS based algorithm updates a 2D vertex called path_list holding all paths forming a cycle for each vertex. The vertex started is stored as the start node and does not change until all cycles are explored for a given vertex. The calling node is updated whenever an edge-sharing node calls the recursive function again. In order to ensure a node does not call its edge-sharing node which has already been visited, we traverse the path. As the path length is small, because it is limited to input k which will be at most 5, this search operation does not have much detrimental effect. If the parameter k was much higher though, this approach would not be efficient as the operation cost would increase in the order of O(log n).

When the caller and start node are eventually the same, which implies a cycle of a given length, the path list is updated with the path of the found cycle. In the main level, this function is called for each vertex present in the adj array and the procedure is parallelized with a pragma omp for statement. Scheduling for threads is done dynamically, as the adjacency matrix of the graphs is sparse, we anticipate the workload of each thread is not uniform and dynamic scheduling allows threads that had less work in an iteration to take on more work as soon as they finish.

The counting operation for each vertex is done in the critical section of the threads in order to prevent race conditions. For this purpose *#pragma omp critical* is utilized. While counting the cycles, the algorithm traverses the path list of each node and increments the cycle count number of the vertices contributing to form a cycle. Since this operation is done for each vertex, it is expected to increment the count of all the vertices of a cycle, k times. That’s why the counting loop increments the cycle count by the ratio of 1 to cycle length(k).

**Pseudocode**



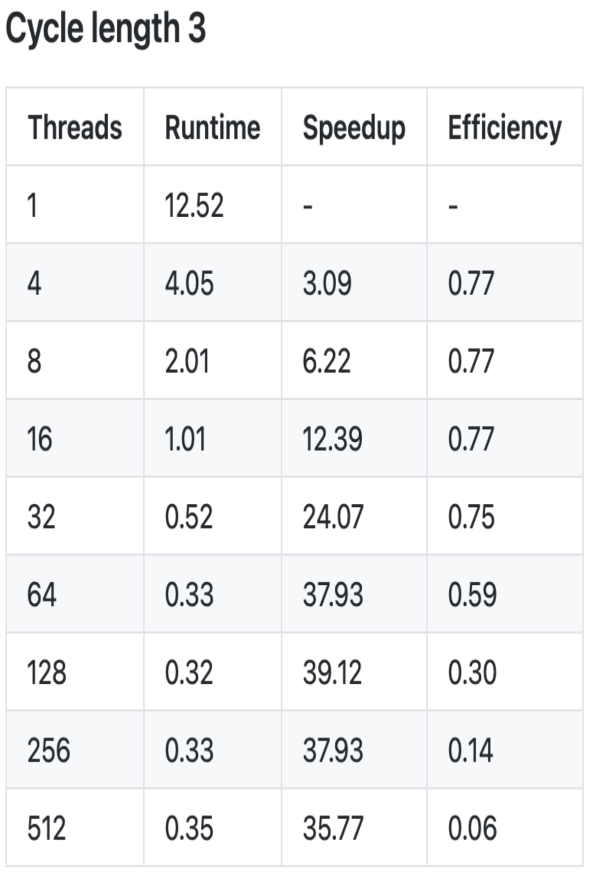
```
recursive_paths(start_node, caller_node, length,max_length,&path_list))
//returns path_list
If(length == max_length)
  If(start_node!=caller_node) //cycle
    return;
   else
     path_list.push_back(path);
     return;

 path.push_back(caller_node);
 for adj vertices of each vertex
  in new_caller=adj[i]
  if(new_caller not found in path:
    call recursive_paths with length+1
```

*Figure 4.0. Pseudocode*

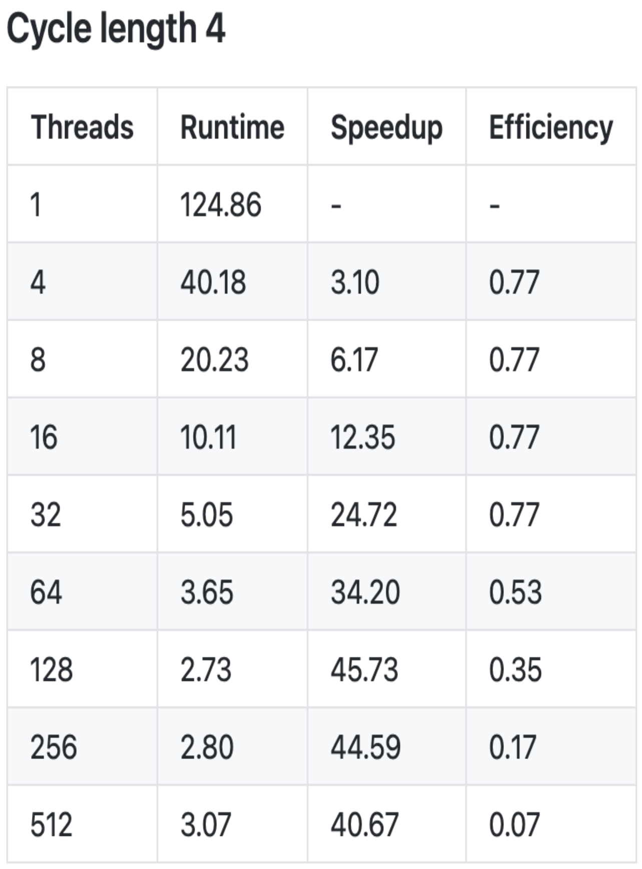
## 5. Multi-Core CPU Results

The biggest drawback with our approach was that the results get slower as the cycle length parameter k grows. While the runtime difference between k=3 and k=4 is not as significant when k=5 the results are significantly slower as the algorithm needs to explore more vertices further down the graph, going deeper in DFS fashion.



## Cycle length 3

| Threads | Runtime | Speedup | Efficiency |
|---|---|---|---|
| 1 | 12.52 | - | - |
| 4 | 4.05 | 3.09 | 0.77 |
| 8 | 2.01 | 6.22 | 0.77 |
| 16 | 1.01 | 12.39 | 0.77 |
| 32 | 0.52 | 24.07 | 0.75 |
| 64 | 0.33 | 37.93 | 0.59 |
| 128 | 0.32 | 39.12 | 0.30 |
| 256 | 0.33 | 37.93 | 0.14 |
| 512 | 0.35 | 35.77 | 0.06 |



## Cycle length 4

| Threads | Runtime | Speedup | Efficiency |
|---|---|---|---|
| 1 | 124.86 | - | - |
| 4 | 40.18 | 3.10 | 0.77 |
| 8 | 20.23 | 6.17 | 0.77 |
| 16 | 10.11 | 12.35 | 0.77 |
| 32 | 5.05 | 24.72 | 0.77 |
| 64 | 3.65 | 34.20 | 0.53 |
| 128 | 2.73 | 45.73 | 0.35 |
| 256 | 2.80 | 44.59 | 0.17 |
| 512 | 3.07 | 40.67 | 0.07 |

*Table 5.0 Results in terms of runtime, speedup, and efficiency of different thread numbers and different cycle lengths*

The speedup is calculated as the ratio of runtimes of the sequential algorithm(1 thread) to multiple threads. In addition to that, efficiency is calculated as the ratio of speedup to the number of processors. In this case, the number of threads is considered as the number of processors. In other words, the efficiency is calculated as the ratio of the sequential runtime to the number of processors times multithread runtime.

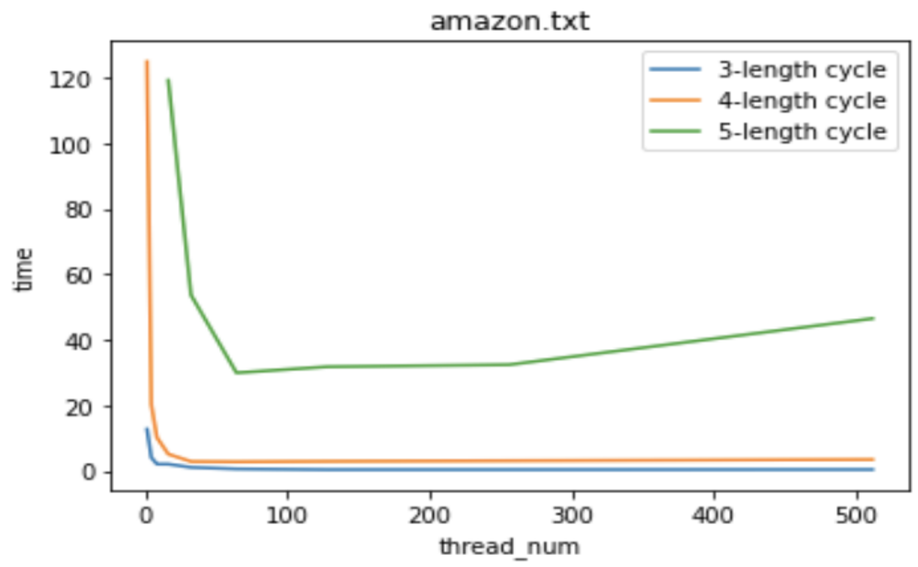


*Figure 5.0. Runtime versus Thread Number Graph for different lengths of cycles-Amazon*

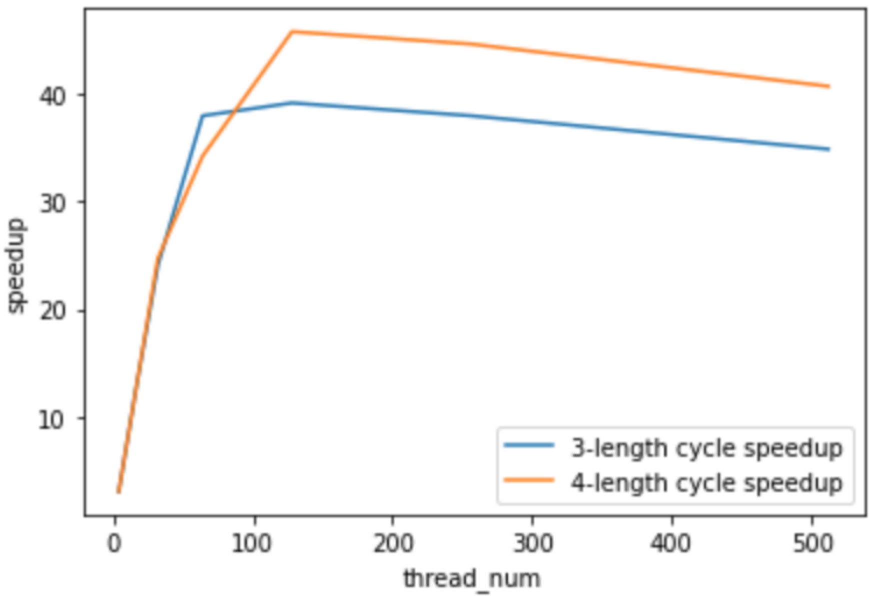


*Figure 5.1. Speedup versus Thread Number Graph for different lengths of cycles-Amazon*

As can be observed from results above, the sequential runtime of the algorithm increases significantly as k increases. Another observation is that the sequential algorithm itself is most likely not an optimal solution, thus yielding a high sequential runtime. Because it is not optimal, the speedup achieved with threads are high, for instance 30-40 fold speedup was observed in some cases. On the other hand, a thread number between 32-64 appears to yield the best performance, as both speedup and efficiency were observed to decrease with 128 threads and more. The parallel program scaling is well up to 64 threads, however it does not scale well after that.

## 6.Further Improvements

In this project, two graphs are expected to be analyzed. Our algorithm successfully analyzes the graph Amazon which is more sparse compared to the graph Dblp. The algorithm performs a cycle count operation for each vertex contributing to a cycle over and over, overhead of computation is observed. That's why the analysis of a graph with many cycles such as the Dblp graph becomes impossible for the present CPU because of the space waste of the algorithm. Further efficiency may be achieved by sorting the vertices in terms of their degrees. With this method, traversing the vertices with large degrees first would remove the need of computing the cycle counts of smaller degree vertices as the large degree vertices would take care of the edges of the others. Therefore, the overhead of computing would be minimized. Moreover, the implementation of GPU parallelism using CUDA is anticipated to yield more efficient space and time complexity.

## 7.How to Run

Programming language used is C++ along with OpenMP interface. OpenMP and Gcc 7.5.0 has to be installed before building.

```
g++ ./cpu_parallel.cpp -O3 -fopenmp
./a. amazon.txt 3
```

where the first parameter is the input graph and second is the cycle length. User then will be prompted to enter desired thread numbers. To see the results, the commented section in main where it says 'TO CHECK OUTPUT' needs to be uncommented, otherwise results will not be printed, only the run-time will be displayed.

# 8.Appendix

## CPU implementation of Depth Limited DFS

### cpu_implementation.cpp

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <omp.h>
#include <math.h> /* fabs */
#include <string.h>
#include <stdlib.h>
#include <limits.h>
using namespace std;


void recursive_paths(int start_node, int caller_node, vector<int> path, vector<vector<int>> &path_list, int length, int max_length,int *adj, int *xadj){


        if(length==max_length){//check if the desired length is reached

            if(start_node!=caller_node)//no cycle found
                return;

            else //cycle found
            {
                path_list.push_back(path);//add current path to path list
                return;
            }
        }


        path.push_back(caller_node);//add current vertex to the path


        for(int i= xadj[caller_node]; i<xadj[caller_node+1]; i++){//check the adjacents of current vertex


                int new_caller= adj[i];
                //check if the neighboring vertex is in the path i.e visited before
                //if the vertex in the path but it is the starting vertex
                if(std::find(path.begin(), path.end(), new_caller)==path.end() || (new_caller==start_node && length==max_length-1)){
                    //call the function again with new parameters
                    //inrement the length by one to keep track of the current depth
                    recursive_paths(start_node, new_caller, path, path_list, length+1 ,max_length, adj, xadj);
```

```
int main(int argc, char *argv[])
{
    int cycle_len = 0;

    string filename = argv[1];
    cycle_len = stoi(std::string(argv[2]));
    int nt;
    cout<<"Please enter thread number: ";
    cin>>nt;

    //Reading the Input File
     std::ifstream fin;
    fin.open(filename);

    int v1, v2;
    std::vector<std::pair<int, int>> intermediate;
    int num_vert = 0;
    int min = INT_MAX;
    while (!fin.eof())
    {
        fin >> v1 >> v2;

        if (v1 != v2)//avoiding self loops
        {
            //updating the both endpoints of an edge
            intermediate.push_back(std::pair<int, int>(v1, v2));
            intermediate.push_back(std::pair<int, int>(v2, v1));
        }
        min = (v1 < min) ? v1 : min;
        min = (v2 < min) ? v2 : min;
        num_vert = (v1 > num_vert) ? v1 : num_vert;
        num_vert = (v2 > num_vert) ? v2 : num_vert;
    }
```

```cpp
    if (min == 0)//checking if the vertices are 0-based
    {
        num_vert++;
    }
    else
    {
        num_vert = num_vert - min + 1;
    }

    std::stable_sort(intermediate.begin(), intermediate.end());


    //      ****CREATE CSR*****
    int *adj = new int[intermediate.size()]; //num of edges
    int *xadj = new int[num_vert + 1];       //number of vertex


    xadj[num_vert + 1] = num_vert - 1;

    memset(xadj, 0, sizeof(int) * (num_vert + 1));

    for (int i = 0; i < intermediate.size(); i++)
    {
        adj[i] = intermediate[i].second;
        xadj[(intermediate[i].first) + 1]++; //That's the trick
    }
    for (int i = 1; i <= num_vert; i++)
    {
        xadj[i] += xadj[i - 1];
    }
    std::vector<double> length_cyles(num_vert, 0.0);

    double start=omp_get_wtime();
#pragma omp parallel for schedule(dynamic) num_threads(nt)
    for(int i=0; i<num_vert+1; i++){//calling the function for each vertex
        if(xadj[i]!=xadj[i+1]){//check if the vertex has edges

            vector<vector<int>> path_list;
            vector<int> path;
            recursive_paths(i, i, path, path_list, 0, cycle_len, adj, xadj);


            #pragma omp critical
            for(int i=0 ; i<path_list.size(); i++){
                for(int j=0 ; j<path_list[i].size(); j++){
                  length_cyles[path_list[i][j]]+=1.0/cycle_len;//increment the cycle count the contributing vertices
                }

            }


        }
    }
    double end=omp_get_wtime();
    //TO CHECK OUTPUT
    //please replace 100 with num_vertices
        /*
     for(int a=0; a<=100;a++){
      cout<<a<<" "<<length_cyles[a]<<endl;
        }
        */

    cout<<"No threads: "<<nt<<" Time: "<<end-start<<endl;
    return 0;
}
```

## 9. Resources

[1]Alon N., Yuster R., Zwick U. (1994) Finding and counting given length cycles. In: van Leeuwen J. (eds) Algorithms — ESA '94. ESA 1994. Lecture Notes in Computer Science, vol 855. Springer, Berlin, Heidelberg. https://doi.org/10.1007/BFb0049422

[2]B. Monien. How to find long paths efficiently. *Annals of Discrete Mathematics*, 25:239–254, 1985.

**[3]**N. Alon, R. Yuster, and U. Zwick. Color-coding: a new method for finding simple paths, cycles and other small subgraphs within large graphs. In Proceedings of the 26th Annual ACM Symposium on Theory of Computing, Montréal, Canada, pages 326–335, 1994.

[4]Yuster, Raphael; Zwick, Uri.SIAM Journal on Discrete Mathematics; Philadelphia Vol. 10, Iss. 2, (May 1997): 14. DOI:10.1137/S0895480194274133

[5]Mahdi. (2012). An Algorithm for Detecting Cycles in Undirected Graphs using CUDA Technology.International Journal on New Computer Architectures and Their Applications, 193-212.https://www.researchgate.net/publication/230771392_An_Algorithm_for_Detecting_Cycles_in_Undirected_Graphs_using_CUDA_Technology

[6]Yang, C., Chang, T., Huang, K., Liu, J. and Chang, C., 2012. Performance Evaluation of OpenMP and CUDA on Multicore Systems. *Algorithms and Architectures for Parallel Processing*, pp.235-244.

[7]Gebremedhin, A. H., & Manne, F. (n.d.). Scalable parallel graph coloring algorithms. *CONCURRENCY: PRACTICE AND EXPERIENCE*, *2000*, 1131–1146.

[8]https://www.usenix.org/system/files/login/articles/login_winter20_16_kelly.pdf

[9] https://www.geeksforgeeks.org/detect-cycle-direct-graph-using-colors/

[10]https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-042j-mathematics-for-computer-science-spring-2015/readings/MIT6_042JS15_Session16.pd