Report Assignment 1 - CM0622-1 (CM90) - March 25, 2023

RunJie Xia - 879779

Counting Triangles in an undirected Graph

Problem:
Count the number of triangles in an undirected graph G = (V,E), where V is the set of vertex with |V| = n, and E⊆V×V is the set of edges with |E| = m. A triangle in an undirected graph is formed by three vertices and three edges, such that each vertex is connected to the other two with an edge.

Data structures:
An undirected graph is represented using an adjacency matrix implemented using vector<vector<bool>>. The matrix has a row and column for each vertex in the graph, and the entries in the matrix indicate whether there is an edge between the corresponding vertices. If there is an edge between vertices i and j, then the (i, j) and (j, i) entries in the matrix are set to 1. Otherwise, they are set to 0.
In order to build the matrix, the program takes as input a text file representing a graph, where each line (i, j) contains integers that represents the edge between the node i and j.
Furthermore, to test more graphs, the function that generates graph has been added.

Sequential Algorithm:
This algorithm loops through all edges in the graph represented by the adjacency matrix, and for each edge (i, j) , if there is an edge, it calls the getIntersection() function to count the number of common neighbors between i and j, excluding i and j themselves. This count corresponds to the number of triangles that contain the edge (i, j).
After looping through all edges and counting their triangles, the total count is returned and divided by three to eliminate duplicates since each triangle is counted three times, once for each edge.

The getIntersection() function takes two vectors that represent the adjacency matrix rows and columns corresponding to two nodes i and j, and it counts the number of common neighbors between them, excluding i and j themselves.

Complexity:
The time complexity of this algorithm is O(n^3), where n is the number of nodes in the graph. The outer loop iterates through all possible pairs of edges (i, j), which takes O(n^2) time. For each pair of edges, the getIntersection() function is called, which iterates over a row and a column of the adjacency matrix, taking O(n) time. Therefore, the overall time complexity is O(n^2 * n) = O(n^3).

Parallelization:
As for the parallelization of the algorithm, OpenMP (Open Multi-Processing) is used. It’s an API that supports multi-threading programming in C++.
The parallelized version uses OpenMP directives to parallelize the outer loop that iterates over all edges.
The directive and clauses used are:
- ”#pragma omp parallel" directive, indicating that the following block of code should be executed in parallel by multiple threads.
- “num_threads”: the number of threads to use for the parallel region.
- “reduction(+:count)”: that the "count" variable should be combined across all threads at the end of the parallel region using the "+" operator.
- “schedule(dynamic)”: the scheduling strategy to use for dividing the work among threads. In this case, it's set to "dynamic", which means that the work is divided into chunks and assigned to threads as they become available.
- “shared”: specifies the variables that should be shared among all threads.
- “default(none)”: that no variables should be assumed to be shared or private by default, and that all variables must be explicitly specified using the "shared" or "private" clauses.

The outer loop iterates over all edges in the graph in parallel. Each thread is assigned a different range of edges to work on. The inner loop iterates over all edges that come after the current edge, and checks if there is an edge between those nodes. If there is an edge, it calls the getIntersection() function to count the number of triangles involving that edge. The count of triangles is accumulated using the "count" variable.
After the parallel region is finished, the total count of triangles is divided by 3 and returned.

Project structure

The project has the following structure:
- ‘main.cpp’ file containing the all the test needed to run the evaluations.
- ‘Graph_ds.cpp’ and ‘Graph_ds.h’, source code and header files where all the functions needed are.
- a ‘/data’ directory where all the test data are stored.

- a ‘/execution_times’ directory containing the execution times of the algorithm on the graphs with varying number of threads saved as “results_numberofnodes_density or source ”.
- ‘plot_data.ipynb’ jupyter notebook used to read and plot the execution times.

The project was created and executed on the Clion IDE.

Experimental evaluation

Hardware components

The program was executed on a MacBook Pro 2017 with the following specifications:
• Processor: 2,3 GHz Dual-Core Intel Core i5
• RAM: 8 GB
• SSD: 256 GB
• macOS Monterey 12.0.1

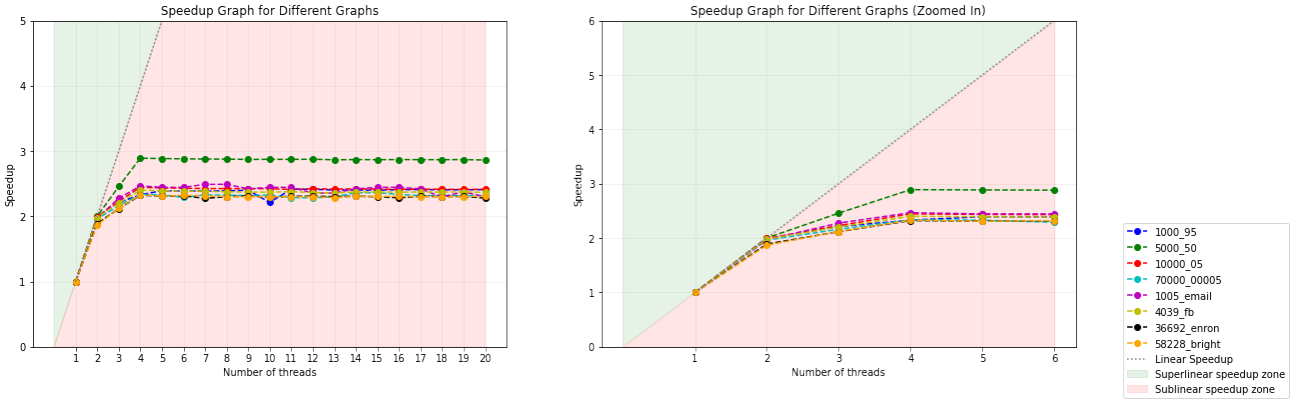
Dataset

The program execute the algorithm, for each of the datasets listed below, using a number of threads from 1 to 20.
The datasets used to test the program are the following:
4 from http://snap.stanford.edu/data/index.html, and 4 generated using the getGraph_dense() function.

Graph name	Source	No. nodes	No. edges	Density	N. triangles	Description
email-Eu-core network	SNAP data	1005	25571	0.05068	105461	small, sparse
Social circles: Facebook	SNAP data	4039	88234	0.01081	1612010	small, sparse
Enron email network	SNAP data	36692	183831	0.00027	727044	medium, sparse
Brightkite	SNAP data	58228	214078	0.00012	494728	large, sparse
1000_95	Generated	1000	~474283	0.95	x	small, dense
5000_50	Generated	5000	~6249615	0.5	x	small, dense
10000_05	Generated	10000	~2498420	0.05	x	medium, sparse
70000_00005	Generated	70000	~122498	0.00005	x	large, sparse

The number of edges for the four generated graphs was calculated using the getGraph_dense() function, which takes the number of nodes and density as input. The number of triangles is not reported because it can vary based on the generated graph. However, we can assume that the algorithm is correct since it worked for the four graphs from the SNAP dataset.

Results



The algorithm maintained linear speedup using only 2 threads, which is expected given that the maximum number of cores available is 4 (also determined using omp_get_max_threads()). All performance tests followed the typical results. The speedup remains consistent as the number of threads increases, as shown in the first plots. Speedups across different types of graphs are generally similar, with little impact on graph size for low densities. However, the graph with the highest density (0.5) demonstrated a better speedup compared to the others. Although the considered graphs were mostly sparse, overall, the algorithm scaled better for the denser graph compared to sparser ones.