Counting triangles in an undirected graph

Michele Lotto 875922

1 Implementation

The implementation is done in 'class ListsUgraph'. The class represents undirected graphs created by file and stored as a list of edges: the constructor reads a '.txt' file called 'edges.txt' from which it creates a 'vector<pair<size_t, size_t >>'. The 'Counting Triangles in an undirected Graph' algorithm used by the class is summarized as:

1. For each edge (a,b):
    1. insert "a" in adjacency list of node "b".
    2. insert "b" in adjacency list of node "a".
2. For each edge (a,b):
    1. Compute the intersection between the adjacency list of node "a" and the adjacency list of node "b".
    2. Sum the size of the intersection with the total number of triangles.
3. Divide by 3 the total number of triangles and return the result.

The implementation is done through two methods: 'count_triangles()' (sequential execution) and 'count_triangles_multi(size_t n_threads)' (parallel execution). Each method returns a tuple of 3 'unsigned long long': the number of triangles, the data structure construction time and the algorithm execution time.

The parallelization is done for both the (1) and (2) steps by dividing the number of edges for each parallel execution; for step (1) 'std::thread' was used, while for step (2) 'std::future' combined with 'std::async' were used. For step (1) it has been necessary to use a mutex for every adjacency list to prevent conflicts. The data reading was not parallelized given the poor speedup that could be achieved with only one disk. The data reading performance was not measured.

2 Project structure and usage

The project structure is the following:

- '\data': a directory where test data are stored; every sub-directory represents a specific graph, containing a 'nodes.txt' file and a 'edges.txt' file. The 'nodes.txt' contains a single number, representing the number of nodes. Every line of 'edges.txt' represents a specific edge in the form of two indexes, representing two different nodes in the graph. The indexes should be greater or equal to zero and less than the total number of nodes. The total number of lines should be less than $\binom{\#nodes}{2}$ (maximum number of edges in an undirected graph), indeed self-loops and duplicated edges are not allowed. Furthermore the '\data' directory contains 'data_generator.py', a python script to generate random test data and 'data_preparation.py', a python script to check and modify '.txt' data to be used by the class.
- \results: a directory containing the timings for all test data.
- ListsUgraph.h and ListsUgraph.cpp: header and cpp files containing the 'class ListsUgraph' definition and implementation, respectively.
- 'test.cpp': cpp file containing the test function and the main function. The test function takes in input a graph name and populate the '\results' directory with the mean timings for the specific graph. An usage example is also present, as a comment, in the main function.
- 'plot_generator.ipynb': python notebook for the speedup plot generation.

The project was compiled and executed using `g++ -std=c++11 -O3 *.cpp -o test && ./test > test_output.txt`.

3 Testing and Results

The sequential algorithm complexity is: $T(n,m) = [\theta(m)2\mathcal{O}(\log n)]_{(1)} + [\theta(m)\mathcal{O}(n)]_{(2)} = \theta(m)\mathcal{O}(n)$, where $n = \#nodes$, $m = \#edges$ and:

(1) construction of adjacency lists: for each edge, insert one element in two ordered vectors.

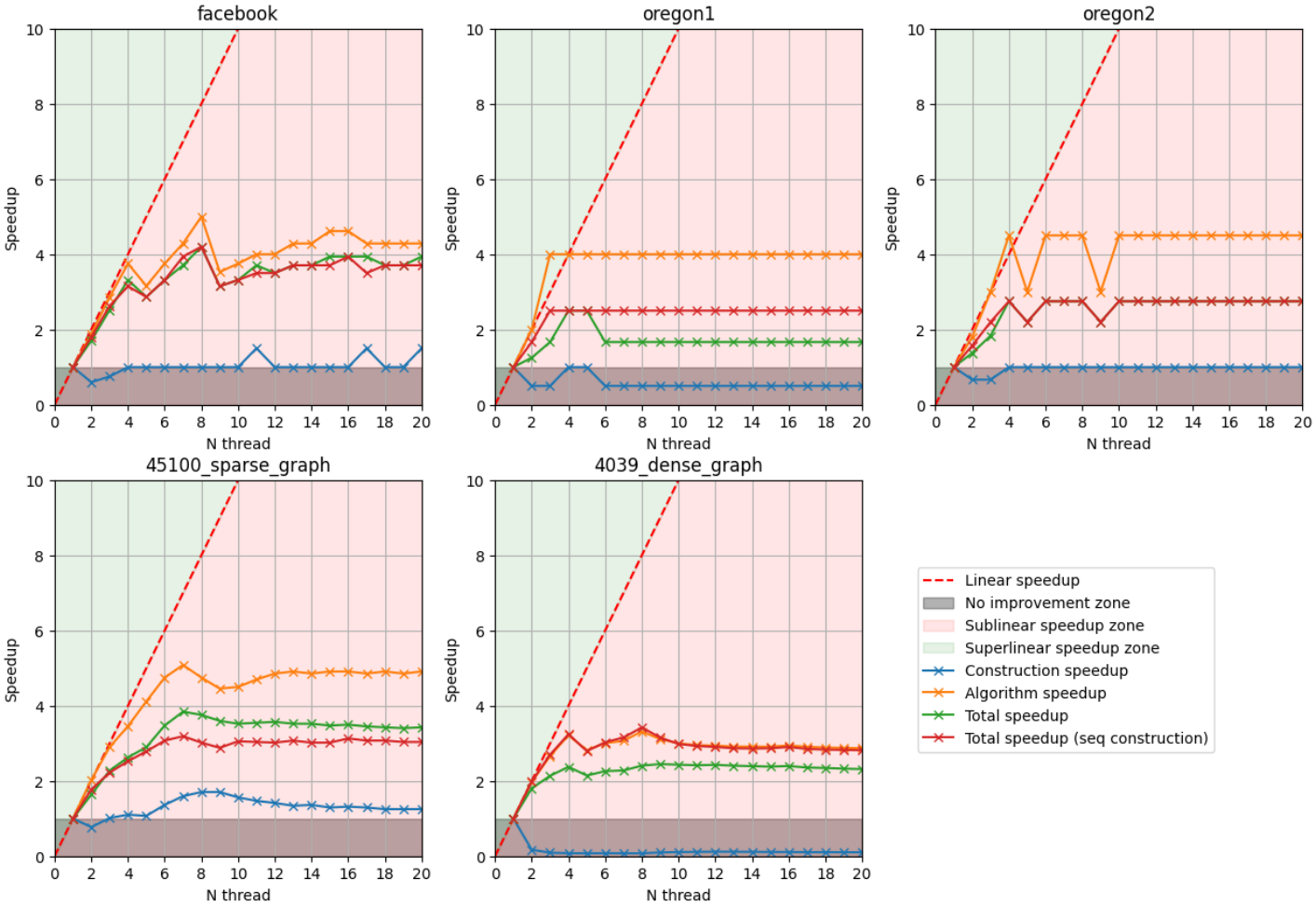
(2) counting triangle algorithm: for each edge, compute the set intersection between two ordered vectors.

The complexity differ based on the graph type:

- Sparse graphs, where $m = \mathcal{O}(n)$: $T(n) = [\theta(n)2\mathcal{O}(\log n)]_{(1)} + [\theta(n)\mathcal{O}(n)]_{(2)} = \theta(n)\mathcal{O}(n)$.
- Dense graphs, where $m = \mathcal{O}(n^2)$: $T(n) = [\theta(n^2)2\mathcal{O}(\log n)]_{(1)} + [\theta(n^2)\mathcal{O}(n)]_{(2)} = \theta(n^2)\mathcal{O}(n)$.

The implementation was tested on 'Intel i7-4790 (8) @ 4.000GHz' using the following graphs:

Graph name	Source	N° nodes	N° edges	Density	N° triangles	Description
facebook	SNAP data	4039	88234	0.01082	1612010	small, sparse
oregon1	SNAP data	11174	23410	0.00037	19894	medium, sparse
oregon2	SNAP data	11461	32731	0.00049	89541	medium, sparse
45100_sparse_graph	random	45100	990500	0.00097	14125	large, sparse
4039_dense_graph	random	4039	8097333	0.99296	10743434978	small, dense



As expected, the best total speedup is almost always reached with 8 threads. The total speedups of "oregon1" and "oregon2" remain stable (with some perturbations) after reaching 3 threads and 4 threads respectively and it is interesting to note that they initially reached a superlinear speedup with a small number of threads.

The speedup remains stable as the number of threads grows in the plots.

The parallelized data structure construction performs better than the sequential one only on large sparse graphs: a small number of edges and a large number of nodes guarantees a small number of conflicts (sequential bottlenecks) in the insertion of the nodes indexes.