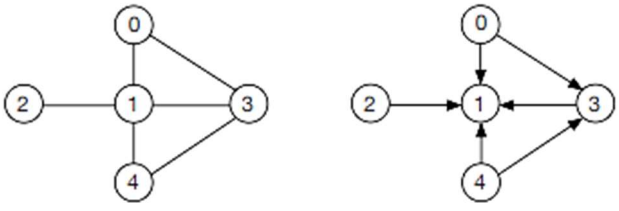
Counting Triangles in an undirected Graph

The algorithm for counting triangles was taken from the document "www.cs.cmu.edu/~jshun/triangle.pdf", since the algorithm takes full advantage of parallelism in a multicore system and is able to optimize memory access being cache-oblivious by building a memory hierarchy. To count the number of triangles in a graph, the algorithm performs two main steps:

```
1: procedure Rank-By-Degree(G = (V, E))
2:     Compute an array R such that if R[v] < R[w] then d(v) ≤ d(w)
3:     par for v ∈ V do
4:         A+ [v] = {w ∈ N(v) | R[v] < R[w]}
5:     return A+
6: procedure TC(A+ )
7:     count = 0
8:     par for v ∈ V do
9:         par for w ∈ A+ [v] do
10:        I = intersect(A+ [v], A+ [w])
11:        count += |I|
12:    return count
```

1. *Ranking Step*: by calling the **create_RankList** function which takes a list of edges as input, it creates a map structure whose key is the node label and the value is the degree of that node. Subsequently the **create_adjList** function is invoked which, taking in input the list of edges and the *RankList*, creates the *ranked adjacency list* A^+ , defined with a map structure. The key contains the node label and the value contains a vector with all vertices in the neighborhood of that node satisfying the following property: $\{w \in N(v) \mid R[v] < R[w]\}$. This causes our indirect graph to be transformed into a directed graph where each node contains as neighbors only the nodes with the highest rank of its. The ranking helps to improve the asymptotic performance and ensures each triangle is counted only once. *example*:

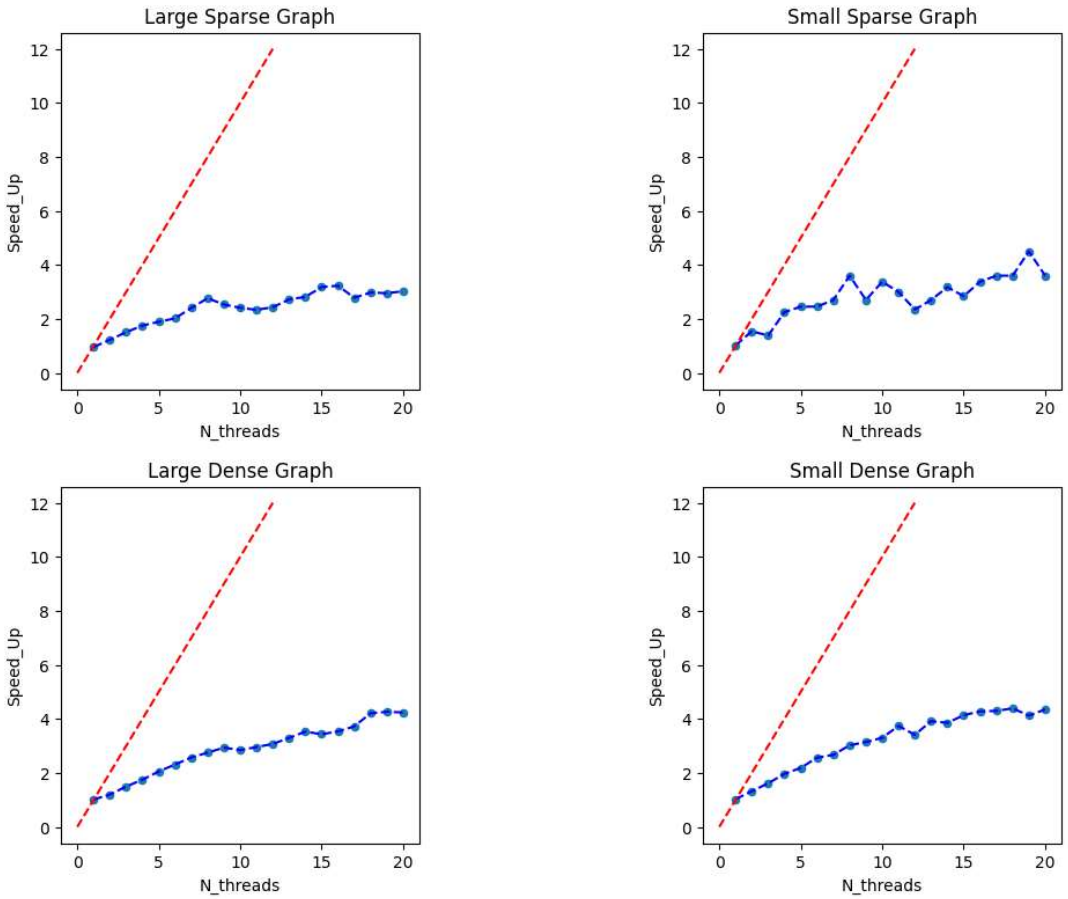
Vertex	0	1	2	3	5
Rank	1	4	0	3	2



2. *Counting Step*: to count the number of triangles it use the adjacency list created in the previous step. So for each vertex $v \in V$ in the graph it takes all the vertices in the neighbor vector $A^+[v]$ and makes the intersection between the two neighbor vectors ($A^+[v] \cap A^+[w]$). Each common out-neighbor u corresponds to a triangle (v, w, u) where $R[v] < R[w] < R[u]$. To count the number of nodes, add the size of the output vector from the intersection.

Parallel Implementation: The intersection between two vectors of neighbors ($A^+[v] \cap A^+[w]$) can easily be performed in parallel, for this reason the notation *#pragma omp parallel for num_threads(n_threads) reduction(+:count)* to parallelize the first loop, so for each node the intersection with the vectors of its neighboring nodes will be performed in parallel, this creates a memory hierarchy which helps increase the number of cache hits.

In the graphs below it can be seen that the speed-up grows as the number of threads increases, however two different behaviors can be noted based on whether a sparse or dense graph is being analyzed.



In Sparse graphs the growth of the speed-up is very unstable, sometimes it rises and sometimes it falls, in a zigzag pattern. This may be caused by the fact that there are few edges between the nodes and even if the algorithm is cache friendly, in some cases cache misses occur as the neighbor vectors size are small and the spatial locality helps up to a certain limit.

In Dense graphs instead we note that the growth is more constant, with fewer oscillations probably because the algorithm is cache-oblivious, it saves the data in a hierarchical way and the spatial locality can be exploited. Therefore, since a node has several neighbors, the same vector of neighbors will be used several times to perform the intersection.

However, as more threads increase, the growth is not so noticeable, caused by the fact that the threads have to coordinate and this creates overhead, wasting resources. The maximum speed-up reached on average by all graphs is around the value 4, this shows that although the speed-up oscillates more and one less, the algorithm works excellently with both dense and sparse graphs.