Distributed Systems

Project Report

Graph Algorithms using MPI

Abhinav Vaishya (2018121003), Sankalp Agarwal (20171161), Pragun Saxena (20171127)

1. Counting the number of triangles in a graph

Algorithms used:

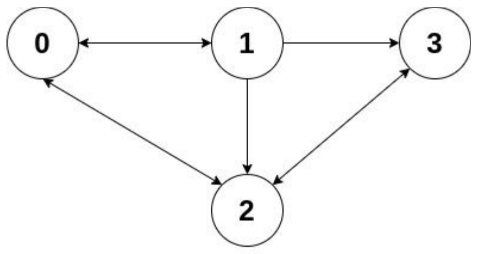
1. Sequential Algorithm
    a. Algorithm: For every node *x* in the graph, we iterate over its adjacency list. For every neighbor *y* of *x*, the set intersection of the adjacency list of x and y are taken. As x and y both have an edge, and the nodes in their intersection would have an edge to both of them, hence forming a triangle. The adjacency lists are sorted to efficiently calculate the set intersection of 2 lists. This algo also leads to multiple counting of the same triangle by different nodes, 3 ways of choosing the first node and 2 for choosing the second node, hence the ans is divided by 6.

    b. Complexity:
        i. Time Complexity: O($N * E$)
        ii. Space Complexity: O($N + E$)

2. Parallel Algorithm (np is the number of processes)
    a. Algorithm: The above mentioned sequential algorithm is converted to parallel, each process has the adjacency lists of all nodes. Now the nodes x are divided among each process. This means, for each process only a certain set of nodes would iterate over their adjacency lists and calculate the set intersection, this would reduce the time complexity by np times but increase the space complexity by np times.

    b. Complexity:
        i. Time Complexity: O($N * E / np$)
        ii. Space Complexity: O($(N + E) * np$)
        iii. Message Complexity:O(np)

c. Example:



As we can see in this example, there are 2 triangles.
Assuming np = 2, we have 2 sets of nodes {0, 1} and {2, 3}.

Process 1 -

Iterates over its set of nodes and then their adjacency lists
0 -> 1, now set intersection = {2} cnt = 1
0 -> 2, now set intersection = {1} cnt = 2
1 -> 0, now set intersection = {2} cnt = 3
1 -> 2, now set intersection = {0,3} cnt = 5
1 -> 3, now set intersection = {2} cnt = 6

Process 2 -

Iterates over its set of nodes and then their adjacency lists
3 -> 1, now set intersection = {2} cnt = 1
3 -> 2, now set intersection = {1} cnt = 2
2 -> 0, now set intersection = {1} cnt = 3
2 -> 1, now set intersection = {0,3} cnt = 5
2 -> 3, now set intersection = {1} cnt = 6

Merging both counts we get 12 and then dividing by 6 we get 2.

3. PATRIC: A Parallel Algorithm for Counting Triangles in Massive Networks
   a. Algorithm: PATRIC is a state of the art algorithm for triangle counting in massive networks, for the graph G(V, E), the graph is partitioned into np subgraphs where in each subgraph G_i, V_i set of nodes is selected where $V_i \cap V_j = \Phi$, and another set V^c_i is also calculated containing V_i and N_v(neighbour set of all vertices in V_i) Similarly edge set E_i is also calculated containing all edges present in the graph containing the vertices in the set V_i.
   Each process "i" is responsible for counting triangles incident in V_i. The triangles which are distributed among different groups are taken care of by enquiring through message passing as we already have N_v.

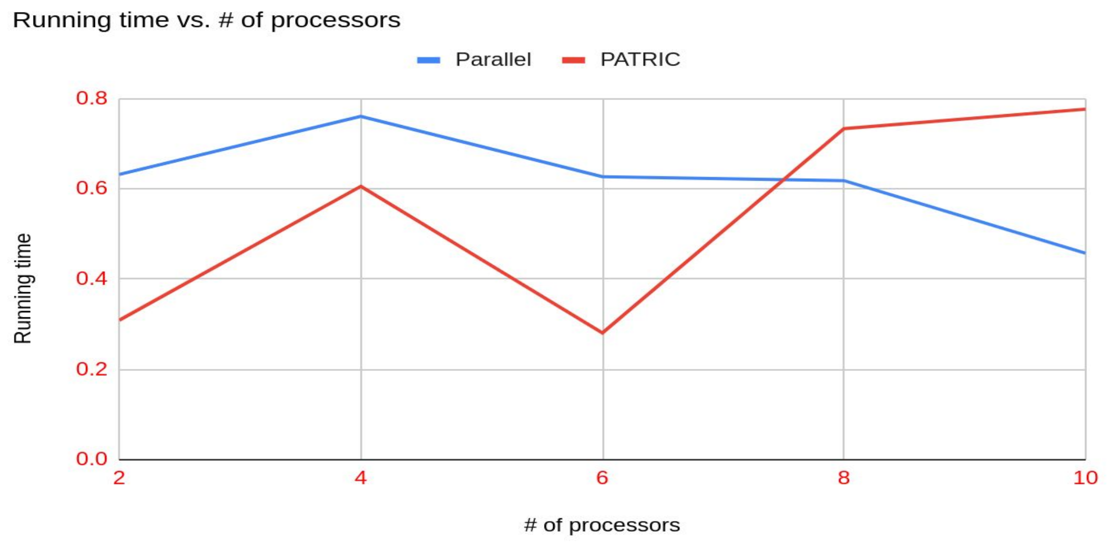
   b. Complexity:
      i. Time Complexity:O($N * E/np$)
      ii. Space Complexity: O($N + E$)
      iii. Message Complexity:O(N)

Results: The algorithms were tested on different test cases with varying numbers of nodes and edges in the graph. We report the running time for the test case which had N = $4*10^3$ nodes and E = $8*10^4$ edges.

The sequential algorithm took 1.051943s to give the output on the same input.

# of processors	Parallel	PATRIC
2	0.631976	0.308741
4	0.760576	0.605656
6	0.626933	0.280668
8	0.618172	0.733458
10	0.457337	0.776659

Graphs:



Conclusion:

If we compare our parallel program with that of PATRIC, their algorithm is mainly based for massive networks, hence their is not much difference in the runtime of our code and theirs, our parallel algorithm even performed better for number of processes 8 and 10, this mainly happened because of increased message passing in PATRIC. But overall if we see, PATRIC is very space efficient as it takes almost the same amount of memory as the sequential and takes time as much as our parallel code.

Hence, for a large number of nodes PATRIC is superior to our algorithm as it saves on the storage.

2. Counting the number of 4-cycles in a graph

Algorithms used:

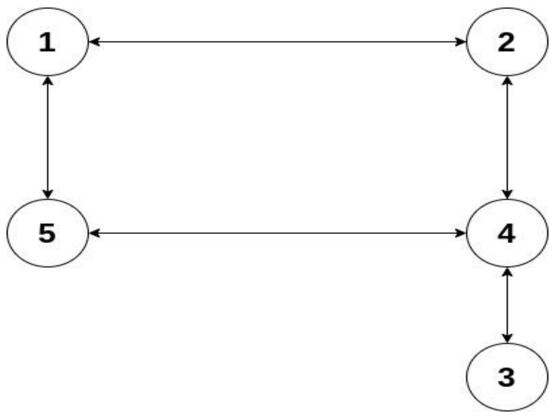
1. Sequential Algorithm
    a. Algorithm: For every node *x* in the graph, we iterate over its adjacency list. For every neighbor *y* of *x*, we iterate over its adjacency list. Now, for every neighbor *z* of *y*, such that $z \neq x$, the set intersection of the adjacency list of x and z are taken.
    Even this algo has cases of multiple counting, 4 ways of choosing the first vertex and then 2 ways of choosing the second vertex, hence the ans is divided by 8.

    b. Complexity:
        i. Time Complexity: O($N^2 * E$)
        ii. Space Complexity: O($N + E$)

2. Parallel Algorithm:
    a. Algorithm: The above mentioned sequential algorithm is converted to parallel, each process has the adjacency lists of all nodes. Now the nodes x are divided among each process. This means, for each process only a certain set of nodes would iterate over their adjacency lists, and for every neighbour y of x, y would iterate over its adjacency list and calculate the set intersection of adjacency list of x and z, 1 would be reduced from this count because that would be the node y itself. This would reduce the time complexity by np times but increase the space complexity by np times.

    b. Complexity:
        i. Time Complexity: O($N^2 * E / np$)
        ii. Space Complexity: O($N + E$)
        iii. Message Complexity: O($log(np)$)

    c. Example:



As we can see in this example, there is only 1 4-cycle.
Assuming np = 2, we have 2 sets of nodes {1, 2, 3} and {4, 5}.

Process 1 -

Iterates over its set of nodes and then their adjacency lists
1 -> 2 -> 1, x == z, continue
1 -> 2 -> 4, now set intersection = {2, 5} cnt = 1
1 -> 5 -> 1, x == z, continue
1 -> 5 -> 4, now set intersection = {2, 5} cnt = 2
2 -> 1 -> 2, x == z, continue
2 -> 1 -> 5, now set intersection = {1, 4} cnt = 3
2 -> 4 -> 2, x == z, continue
2 -> 4 -> 5, now set intersection = {1, 4} cnt = 4
3 -> 4 -> 2, now set intersection = {} cnt = 4
3 -> 4 -> 3, x==z, continue
3 -> 4 -> 5, now set intersection = {} cnt = 4

Process 2 -

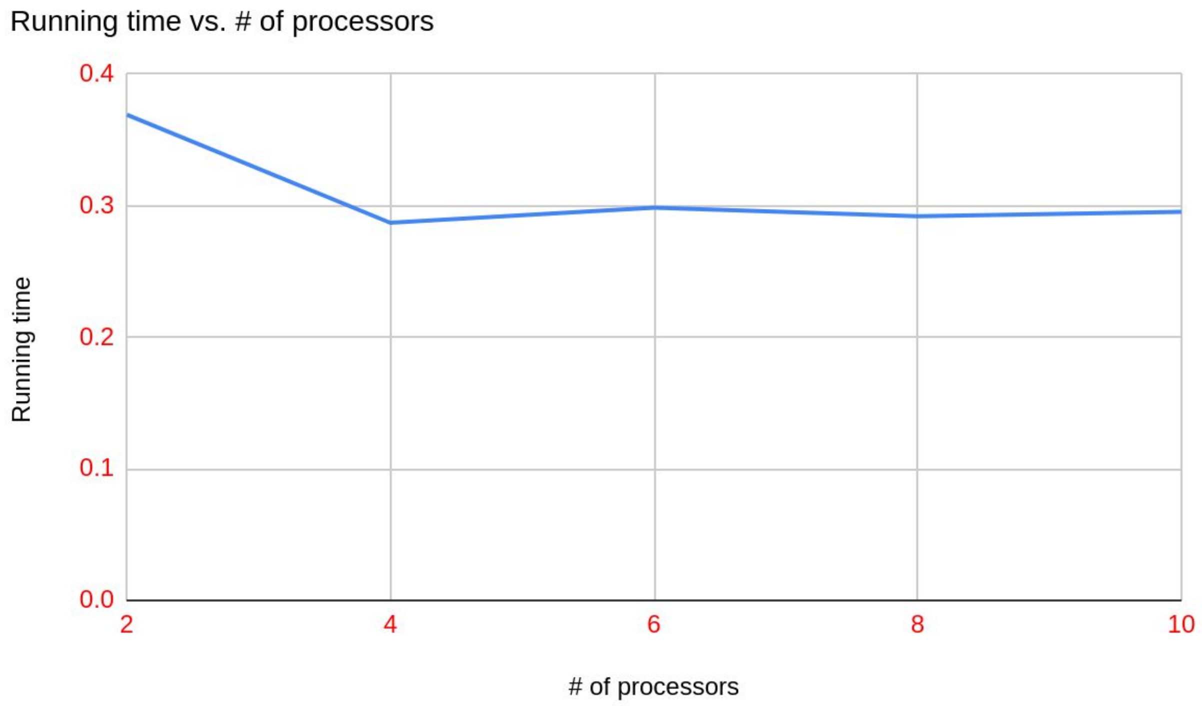
Iterates over its set of nodes and then their adjacency lists
Similarly we iterate for nodes 4 and 5, and get the count 4.
Merging both counts we get 8 and then dividing by 8 we get 1.

Results: The algorithms were tested on different test cases with varying numbers of nodes and edges in the graph. We report the running time for the test case which had N = 10^3 nodes and E = 10^4 edges.

The sequential algorithm took 0.773172s to give the output on the same input.

# of processors	Parallel
2	0.369274
4	0.287005
6	0.298587
8	0.291940
10	0.295379

Graph:



Conclusion:

As, we can see from the above graph, the runtime reduces to about half from sequential to parallel (for np = 2), but doesn't decrease much from thereon, this could be the case because our partitioning of the graph is not that efficient and probably a single process is taking much time and other processes are idle. But still the parallel algorithm works much better than the sequential.

Note : The complexities are calculated based on the following assumptions :-

1. There are **N** nodes and **E** edges in the graph
2. The number of processes are **np**
3. No adjacency list has more than **(N-1)** nodes, i.e., there are no self-loops or multiple edges.