Algorithms for Triangle Counting

Harsha Vardhan 150050073
Mani Shankar 150050081
Joshi Kosuru 150050094

The algorithms discussed here are majorly drawn and attributed to doctoral thesis of Thomas Schank

Naive algorithm

We iterate over all triples of vertices v1,v2 and v3 (via three for loops) and check if there is an edge between v1-v2,v2-v3 and v3-v1. If there is one, we increase our triangle count.

Even assuming a very optimistic adjacency matrix representation (needs huge memory) , this brute force takes $O(n^3)$ time with $O(n^2)$ space.

For any n > 1e4, this algorithm is practically infeasible wrt time and even space.

Note : In all our aforementioned triangle counting algorithms, we assume that the reader takes care of trivial details like say dividing the answer by a constant c because the same triangle is counted in few symmetric places. Like if we count the triangle formed by v1,v2 and v3, in all its permutations we divide our answer by 6 to get the final answer.

Matrix multiplication

We know that the (i,j)th entry of a matrix A^k represents the number of paths starting from vertex i and ending at vertex j of length k. Let us consider a triangle uvw. This triangle contributes 2 paths from vertex u to u itself - one via v first (uvw) and the other via w first (uwv). Similarly it contributes 2 paths to vertices v and w. So number of triangles in the graph would be the trace of the matrix A^3 divided by 6.

The time complexity is $O(n^x)$ where x is the fastest known index for matrix multiplication. (in practice, something around 2.37) Space complexity is $O(n^2)$ because adjacency matrix representation is used.

Though using fast matrix multiplication is the best method to count number of triangles, (We are able to do it in $O(n^x)$ time even when there are $O(n^3)$ triangles in case of a fully connected graph!) , It is not feasible for practical graphs where number of nodes are high and the adjacency matrix is sparse. So now we shift to algorithms utilising the adjacency list representation of graph thus needing only O(m) space. (Though $O(n^2)$ in the worst case)

Node Iterator

We iterate over all nodes of the graph. For each node of the graph, we consider all pairs of its neighbours and then check if there is an edge between their neighbours. If so we have found a triangle. But note that for efficient O(1) lookup we need an adjacency matrix again. Instead we might store a hash map of neighbouring vertices of each vertex and find it in amortised constant time but the constant would be too high.

The time complexity would be $\mathbf{\Sigma}(O(n * d^2)) \sim O(n * d_{max}^2)$ where d is the degree of each vertex and d_{max} is maximum among them.

[Implemented in nodeIterator.cpp using sets (balanced search trees) instead of hashing for lookup]

Edge Iterator

We iterate over all edges of the graph. For each edge e with endpoints say u and v , we find the intersection of the neighbours of u and v. For each intersection point w, we have found a triangle uvw.

To calculate the intersections efficiently, we do a preprocessing where we first sort the neighbours of all vertices according to their numbering. Then to find the intersection of neighbours of u and v, we can do it in $O(d_u + d_v)$ time where d_u and d_v are degrees of u and v. This can be done by a simple two pointer algorithm as used during merging phase of a merge sort algorithm.

Time complexity is $\sim O(n * \log(d_{max})) + \Sigma(O(d_u + d_v))$

We could have also done this with hashing too by checking if a neighbour of u is present in the adjacency list of v, but considering the higher cost of operations, we ignore it as better algorithms are known.

Naive square root decomposition

Inspired from this. Implemented as a sub-routine of this
Let us calculate c = sqrt(m). If degree of vertex is greater than c, we mark it as heavy. Else we mark it as light. Now we count number of all heavy vertexed triangles, all light vertexed triangles, 2 heavy - 1 light vertexed and 1 heavy - 2 light vertexed separately each in $O(m^{1.5})$ time. The constant factor is high due to large size of the hashmap stored.

Edge Iterator Forward

Now to avoid repetitive counting of a triangle as in previous algorithm and avoid using excess memory (quite a bit!) we turn the undirected graph into a directed one by some particular arbitrary topological ordering. Following the above procedure, we save half the space and more importantly do not waste time in recounting a triangle (and also scaling by a constant afterwards).

[Implemented in edgeIterator_forward.cpp]

Node iterator core

To make the complexity upper bounded by a nice function, we use a particular ordering when iterating through the nodes. We start from the node with the lowest degree and count all the triangles. After iterating through a particular node, we remove the edges which it is part of and then repeat the above algorithm. This algorithm is proven to be upper bounded by $O(m^{1.5})$.

Note that in the worst case, $O(m^{1.5})$ is still $O(n^3)$.

[Implemented this particular order of traversal in nodeIterator_core.cpp]

Edge Iterator Compact Forward

To upper bound the complexity as in the above case, we use a heuristic which guarantees the running time to be bounded by $O(m^{1.5})$. The heuristic is to add an edge from the vertex of lower degree to the higher degree one during the conversion of undirected graph to directed one and proceed as in Edge Iterator Forward. By doing this it is proven that adjacency list of each vertex can have at most $\sqrt{m}$ vertices which leads to bound of $m^{1.5}$.

[Implemented this heuristic in edgeIterator_compactforward.cpp]

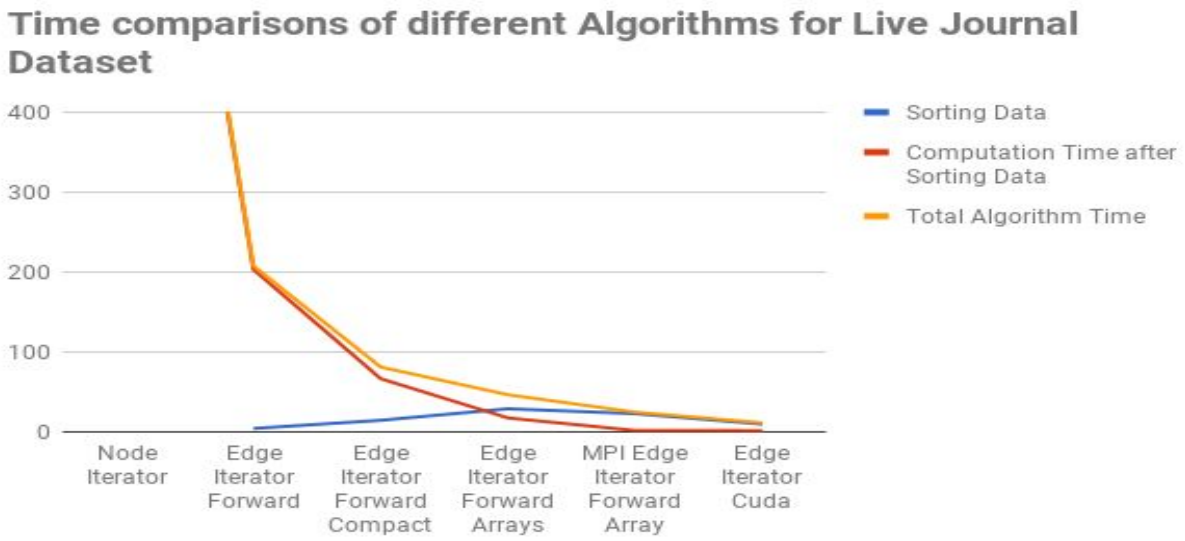
Edge Iterator Compact Forward using native arrays
[Attributed to Adam Polak's work]

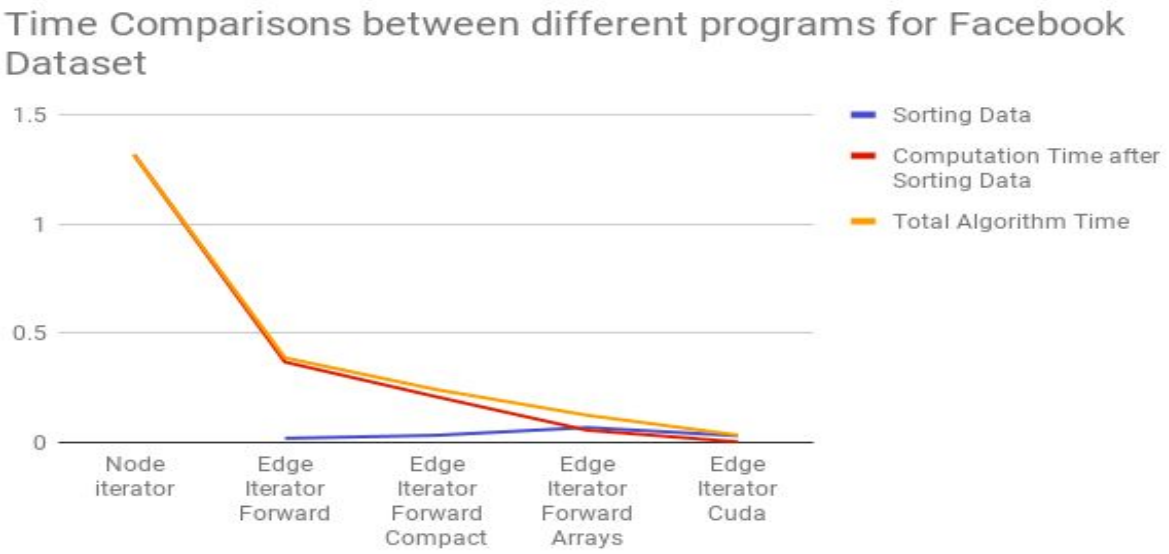
We have been using 2-dimensional adjacency lists for storing the graph and that too using stl vectors. Now we'll use only 1-dimensional native arrays to speed it up significantly and also make the data structures used compact enough for easy transfer to device memory for running it using cuda.

First we sort all the edges by the first vertex and in case of tie by the second vertex. So now the list of edges looks like a concatenation of the sorted adjacency list of each vertex.

We'll store three arrays start,end and edge of sizes n,n,and 2*m each. For each edge i connecting u to v, we set edge[i] = u and edge[i+m] = v thus representing the list of edges in a single array. To figure out the neighbours of a given vertex, we need to figure out the range of values in edge responsible for a given vertex. So we store two arrays start and end where start[i] gives us the first responsible index of the vertex and end[i] gives the last one.

After implementing the above mentioned serial algorithms and seeing the performance over some datasets, we came to the conclusion of further parallelising our best sequential algorithm ie. Edge Iterator Compact Forward with native arrays. According to our analysis, the only algorithms bounded by $O(m^{1.5})$ functions are node-iterator-core and the aforementioned. We don't use node-iterator-core due to high constant factor in lookup. Edge iterator can be parallelised easily because counting number of triangles for each edge can be done independently of the other.





We can observe from the above graphs(time in sec vs program used) for the datasets Live-Journal and Facebook that Edge Iterator Forward Arrays is the best serial algorithm.

We can expect good speedups from CUDA versions because CUDA uses global shared memory and we indeed need shared memory across all edges for knowing the neighbours of other vertices when needed quickly.

In contrast, implementation of an MPI algorithm with distributed memory across nodes is impractical because a node doesn't know the complete information regarding its neighbours thus ultimately resulting in high communication costs. We can use distributed memory to count triangles if the graph has many connected components. But as we usually consider graphs of social communities, they are often connected.

So we have written a MPI version which broadcasts the whole data to each process. So each process of the MPI mimics the serial code for its share of edges with the process knowing full information about the graph.

We briefly discuss the CUDA and MPI implementations.

CUDA
[Majorly attributed to Adam Polak's work]

We translate our best serial algorithm to CUDA directly. Initially, after setting up the necessary arrays in host program serially, we transfer them to the device - do the relevant computations by appropriate kernel calls - add up the results by each thread and get back the answer.

NumTri() kernel call :

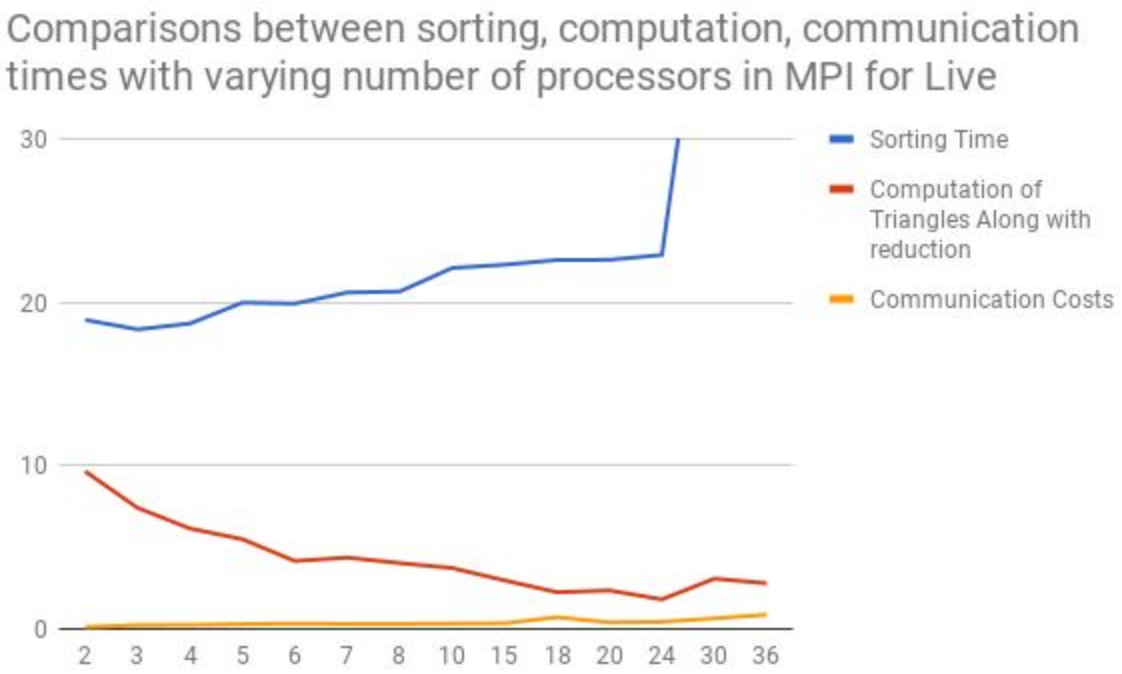
Let us say the number of blocks are b and number of threads per block are t. Total number of threads equal to x = b*t. We divide m edges into x chunks of size approximately c = m/x.
We assign the first m%x threads chunks of size c+1 and the rest c. Edges are assigned cyclically to the threads in the hope of workload balance. Each thread for its assigned edge(s) computes intersection between neighbouring vertices of its end points.

Methods used for Optimizing CUDA code:

- We noticed that sort can indeed be parallelized and so we did this on the GPU. We used thrust library for sorting the pair of vectors. We copy the host vector to a device vector - sort the device vector and get it back for setting our start and end arrays.
- Instead of summing the results naively or using an atomic add, we do a reduce over the result array. We use thrust for doing reduce for us.
- We noticed that our allocation scheme doesn't really use any locality in the edges array. (or caches for that matter). So instead of allocating chunks cyclically we allocate chunks in sequential blocks. So when a thread accesses consecutive elements of an edge array, we get a cache hit speeding up our computations.
- Our kernel function takes three int * pointers as input. Compilers usually assume that the input pointers might be equal or updating one pointer might affect the other pointers and generate machine code. But if we know initially that these pointers are not linked in anyway (no pointer aliasing) and we are only reading these pointers, then we can safely use __restrict__ and const. Quoted from stack-overflow, "*If you use both `__restrict__` and `const` to decorate global pointers passed to a kernel, then this is also a strong hint to the compiler, when generating code to cause those global memory loads to flow through the read-only cache. This can provide application performance benefits, often with little other code refactoring*".
- We initially assumed number of threads per block as 1024 and blocks per grid as 16 (Naively thinking maximum number of threads give good speedup). After tweaking with these values for a certain time,we came to the conclusion that best speed up can be attained when threads per block are 512 and blocks per grid as 32. Any further increase in the blocks per grid resulted in the same speedup on the other hand changing the threads per blocks reduced speed up very slightly. (We just measured the time taken by varying one parameter over the powers of 2 while fixing the other. No logical explanation for numbers used. Use number of threads as 1024 and 32 blocks may be because the effective number of cores on the gpu appear to be around 3000)
- Remove the usage of end array and do it only using start array. We can use start[next+1]-1 as a replacement for end array. But the next vertex might be an isolated vertex. If it is an isolated vertex, we set the start equal to start of the next sequential element which is not isolated. (Deal with boundary cases carefully!)
- (Can be done) Use uint64_t for sorting instead of a pair of int's

MPI Edge Iterator Forward arrays

The edge iterator forward arrays algorithm is parallelized using MPI. Iterations through different edges are parallelizable and have no interdependencies.



Since the input handling and sorting is done by processor zero, the corresponding times do not vary much with change in number of processors, but we can see that computation time decreases with increase in processors. Communication time increases slowly because data is broadcasted here and the constant factor is proportional to log(p) and so increases slowly. We can see a small rise in computational cost after p = 24. This could be because we calculated timings on a machine with 24 processors and any rise in p after that will involve virtual processors (numbers of processors in pixel machine are 24), during which swapping contributes for increase in time.

Analysis of Distributed Memory Paradigm

As discussed previously distributed memory application for this algorithm will result in high communication costs. We analysed this code (not written by us) for the following part. We will assume that adjacency list of a node is completely present at one processor. As above like sequential algorithms, we will iterate across the neighbours of the current node and if we have adjacency list of the neighbour, we will intersect the adjacency list and find out how many triangles are possible. If we don't have the adjacency list of the neighbour, we will find out which processor has that node and send the adjacency list of the node we are iterating to that processor (We will make sure that each processor will receive the adjacency list of a particular vertex at most once). If we receive a message from some node, we'll intersect both adjacency lists to find out the common vertices, each resulting in a triangle. This method involves large communication overheads such as the process will not send another adjacency list till the send

buffer is flushed out (so as to not corrupt the send buffer). This will involve delays in sending and receiving messages.

The high communication costs can be observed with amazon dataset clearly. Computational time taken by by the algorithm with 24 processors is 632 seconds, whereas the sequential node iterator takes approximately 2 seconds.

$$\sigma + \varphi = 2$$
$$\sigma + \varphi/p + \kappa = 632$$

We can directly say from above that the communication costs are exceedingly large.

Other citations :

Stanford Large Network Datasets
CMU project
Thrust
CUDA algorithm idea reference