Compressed Sparse Row: An Efficient Representation for Sparse Graphs

Andrea Bosisio
Computer Science and Engineering
andrea2.bosisio@mail.polimi.it
Politecnico di Milano

Salvatore Gabriele Karra
Computer Science and Engineering
salvatoregabriele.karra@mail.polimi.it
Politecnico di Milano

30 June 2022

Abstract

This is the report of our work done for the Data Structure track of the High-Performance Graph And Data Analytics (HPGDA) contest, in which we had the chance to play with different sparse graph representations and explore their runtime performance to improve. In particular, the challenge requires the creation of an appropriate data structure for the implementation of a graph, a population algorithm and a neighbor search algorithm for each node. The data structure interfaces with real data from the LDBC Graphanalytics Benchmark suite dataset. The evaluation is based on: graph population time, graph size in memory and execution time of the BFS and DFS algorithms.

I. Introduction

In many real application graphs are sparse in the sense that the number of edges (E) really present are much smaller than the possible number of edges. In practice, sparse matrices and graphs are often stored in Compressed Sparse Row (CSR) format, which packs edges into an array and takes space proportional number of vertices and edges. Sparse storage formats pay for these space savings with the cost of updates. CSR format supports fast queries such as membership or finding all neighbors of a vertex, but may require changing the entire data structure to add or remove an edge. In the original formulation, it uses three arrays to store a sparse graph: a node array, an edge array, and a values array. Each entry in the node array contains the starting index in the edge array where the edges from that node are stored in sorted order by destination. The edge array stores the destination vertices of each edge and the values array contains the value associated to that edge.

II. Data pre-processing

To handle graphs with non-sequential node IDs and graphs that have isolated nodes (i.e. nodes that doesn't have any outgoing edges) we mapped the original node IDs list into an sequential increasing integer list (i.e. from 0 up to $V - 1$ where V is the number of nodes of the graph). We've done this through a bidirectional map in order to have the results with the original node IDs. Moreover, to exploit a possible meaning of the ID of the nodes, we've also sorted the original node IDs list in ascending order in the method `load_graph`.

III. Our CSR implementation

The first optimization we've done is to merge the array containing the destination vertices of each edge and the array containing the weight associated to that edge in an unique array of pairs. This was done in order to improve spatial locality and reduce conflicts, since those information are always needed in couple. We named the original node array as `row_ptr` and the merged array described above as `col_idx_weights`.

I. Parallelization

We chose the CSR representation also because it is easy to populate its implementation in a parallel way. For that we used the *OpenMP* library which makes concurrent programming possible in a simple way.

IV. Methods

We've implemented the method `populate` that given and edge list it populates the CSR data structure, and the method `get_neighbors` that given a vertex it returns all the neighbors of that vertex with the corresponding weight.

I. Populate

For the populate method we started with the analysis of the edge list that we receive as input. For the CSR representation we must receive this list in an orderly manner: it must be sorted with respect to the source vertex. Therefore, while looking for a proper sorting algorithm, we realized that the *counting sort* uses an array (`count`) which has the same meaning of the `node` array of the CSR. Firstly, this method counts how many neighbor each vertex has through a parallel for (using the `reduction` clause) in the `row_ptr` array. Then it performs a cumulative sum over that array through the `std::partial_sum` method which turned out to be faster than our custom parallel implementation. Then we copy the content of the `row_ptr` array in the `count` array which will be decremented for the actual counting sort algorithm. The last phase is to compute the new position on the `col_idx_weights` for each edge. This is done in a parallel way as well using locks on the current source node.
Lastly we need to sort the neighbors of each vertex to obtain always the same order and therefore obtain the same BFS and DFS sum independently from each iteration.

II. Get Neighbors

For the `get_neighbor` method we used a customized iterator to scroll through the data structure. The start position of the information of the neighbors of node i is given by `row_ptr[i]` and the end position by `row_ptr[i+1]`. We used an Input Iterator, because we need only to read-only once each value pointed, in this way we don't need to keep trace of the size of the arrays.
This method turned out to be fast because CSR store all the information about neighbors of closer indexed vertices in a sequential memory area.

V. Results

In this section we show the result we obtained on the following graphs:
wiki-Talk - 5.021.410 edges, 2.394.385 nodes
cit-Patents - 16.518.947 edges, 3.774.768 nodes
dota-league - 50.870.313 edges, 61.170 nodes

I. Our Machine

System: We ran our experiments on an laptop with Intel Core i7-8550U, and 1.8GHz clock speed, 4 cores. The machine had 8GB of RAM, 256KB of L1 cache, 1M of L2 cache, and 8MB of L3 cache. Programs were written in C++ and compiled with GCC 9.4.0.

Table 1: *Compered result average populating time and memory usage for Adjacency List (AL) and CSR - Serial*

CSR vs AL		
AL	CSR	Graph
$695ms - 454MB$	$350ms - 1MB$	wiki-Talk
$8967ms - 3640MB$	$2337ms - 1023MB$	dota-league
$5443ms - 1182MB$	$3225ms - 28MB$	cit-Patents

Table 2: *Compared result average BFS and DFS time for Adjacency List (AL) and CSR - Serial*

CSR vs AL (BFS-DFS)		
AL	CSR	Graph
$(767ms - 696ms)$	$(142ms - 219ms)$	wiki-Talks
$(197ms - 217ms)$	$(1ms - 1ms)$	dota-league
$(186ms - 188ms)$	$(40ms - 35ms)$	cit-Patents

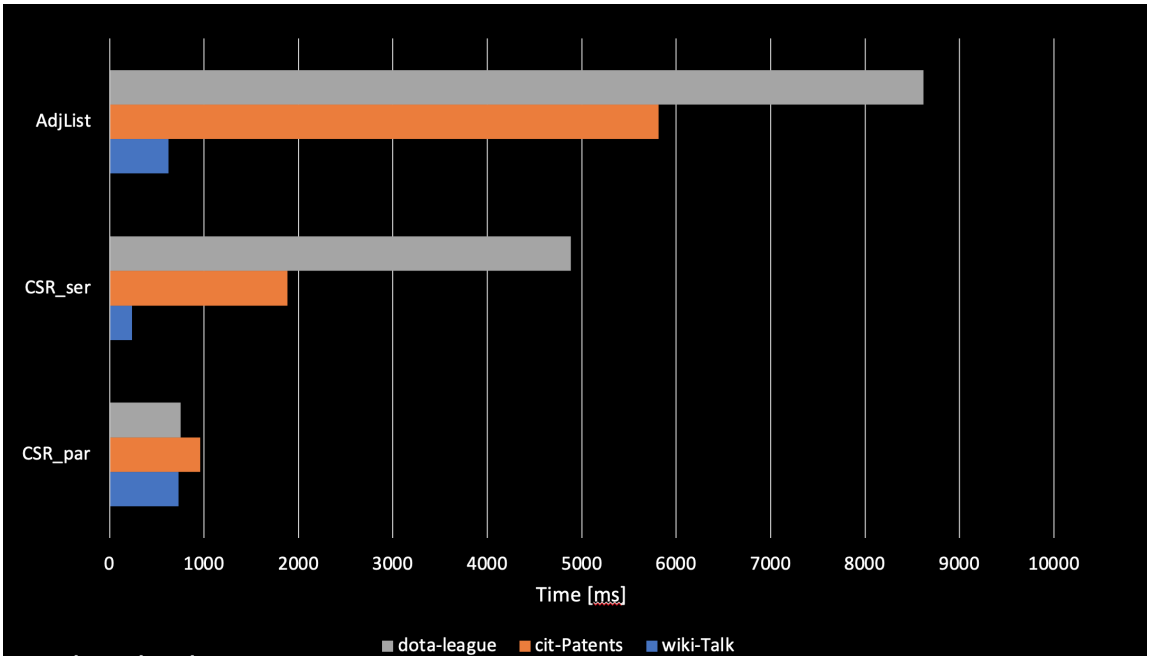
Table 3: *Average parallel populating time for CSR*

CSR - Parallel	
CSR	Graph
$135ms$	wiki-Talks
$1867ms$	dota-league
$1789ms$	cit-Patents

II. NAGS33 @NECTSLab

System: NAGS33 @NECTSLab, 40 Threads
Finally, with this system we had the opportunity to see more meaningful results of our parallel implementation of the CSR.

Figure 1: *Populating time comparison between the example implementation (AL), the serial version of our CSR implementation and the parallel one on three different graphs*



As expected, on small graphs (e.g. *wiki-Talk*) the time results of parallelization are worst with respect to the serial version because of the overhead of the setup of the threads and because of the presence of locks.

III. Memory results of the parallel implementation

In Figure 2, the final results of memory usage of the CSR parallel implementation.

Figure 2: *Memory usage comparison between the example implementation (AL) and the parallel version of our CSR implementation on three different graphs*