Multi-threaded implementation of PageRank

Learning with Massive Data - Tonetto Davide - 884585

1 Introduction

This project aims to implement a multi-threaded version of the PageRank algorithm. The algorithm is used to rank web pages in search engines, and it is based on the idea that a page is important if it is linked by other important pages. The algorithm is iterative and computes the PageRank values of the nodes in a graph.

2 Data structures

In the project, two different data structures are used to store the graph: the first one stores the graph by column and the second one stores the graph by row.

More specifically, the first one stores the graph as a vector of vectors (using pointers), where each vector represents a column of the adjacency matrix and stores the IDs of the nodes that are linked with the node represented by the column.

The second one stores the graph as a vector of vectors, where each vector represents a row of the adjacency matrix and stores the IDs of the nodes that are linked with the node represented by the row. It also stores the out-degree of the node in a vector of size N and the deadends column IDs in another vector of size equal to the number of deadends.

The spatial complexity of the two data structures is the following (the first implementation is more efficient in terms of memory usage):

- The first one has a spatial complexity of $O(N + E)$, where N is the number of nodes and E is the number of edges.
- The second one has a spatial complexity of $O(2N + E + D)$, where N is the number of nodes, E is the number of edges and D is the number of deadends.

Both data structures avoid the use of a sparse matrix to store the graph, and they are more efficient in terms of memory usage since they don't store the zero values of the adjacency matrix.

3 implementations of the sequential PageRank algorithm

Each data structure has a function to compute the PageRank values of the nodes in the graph called `seq_page_rank`. The functions are iterative and compute the PageRank values of the nodes in the graph until the difference between two consecutive iterations is less than a given threshold ($1e^{-7}$).

The two implementations are different in terms of performance, the complexity of the two implementations is the following:

- The first one has a time complexity of $O(I \cdot (N + E + N)) = O(I \cdot (N + E))$, where N is the number of nodes and E is the number of edges and I is the number of iterations needed to reach the convergence. This is obtained by iterating over the nodes and for each node iterating over its out-neighbors to compute the matrix-vector product and the weight of the deadends in $O(1)$ with a total complexity of $O(N + E)$. The second part of the complexity is due to the application of the damping factor and the application of the deadends weight previously computed to the PageRank values, all this using a for loop with a complexity of $O(N)$. Then this is repeated for I iterations.
- The second one has a time complexity of $O(I \cdot (N \cdot D + E))$, where N is the number of nodes, E is the number of edges, D is the number of deadends and I is the number of iterations needed to reach the convergence. This is obtained by iterating over the nodes and for each node iterating over its in-neighbors to compute the matrix-vector product with also the application of the dumping factor with a total complexity of $O(N + E)$. Also inside the iteration over nodes, there is a loop to compute the weight of the deadends in $O(D)$ giving a total complexity of $O(N \cdot D)$. Then this is repeated for I iterations.

4 Parallelization of the PageRank algorithm

The two implementations are parallelized using OpenMP. The parallelization in both cases is done by parallelizing the outer loops of the iteration that computes the PageRank values of the nodes in the graph. The parallelization is done by using the `parallel for` directive of OpenMP and is parametrized by the number of threads (see `par_page_rank` functions). In the first implementation, also the loop for the application of the damping factor and the deadends weight is parallelized.

Race conditions. Only the first implementation needs to be protected from race conditions since the matrix-vector product is computed column by column making it possible to have two threads that update the same PageRank index at the same time. To avoid this, we can use the `critical` directive of OpenMP or the `atomic` directive. The second one is more efficient since it uses atomic operations to update the PageRank values.

Load Balancing. Initially, the parallel implementation of the two algorithms was not balanced, and the execution time was not optimal. To understand the reason for this, the code was profiled using Score-P and the results were analyzed using Vampir (graphs will be shown during the oral presentation). To solve this problem dynamic scheduling was used in the parallel for loop of the two algorithms with a chunk size based on the number of threads and the size of the given graph. This improved the load balancing of the threads and the execution time of the parallel implementation.

5 Results

To execute the tests, the code was run on a machine with the following specifications: 8 cores, 16 threads AMD Ryzen 9 5900HS processor, 24GB RAM. Figure 1 shows the speedup of the parallel implementation of the PageRank for the two algorithms concerning their sequential implementation and the best sequential implementation for the graphs presented in Table 1.

Graph name	$\|V\|$	$\|E\|$	# deadend	Density	Best seq. time (ms)	Best parallel time (ms)
wiki_vote	7115	103689	1005	0.00204854	1 (Alg. 1)	3 (Alg. 1 13 threads)
p2p_Gnutella31	62586	147892	46199	3.7757e-05	7.2 (Alg. 1)	2 (Alg. 1 10 threads)
twitter_combined	81306	2420766	11209	0.000366195	175.2 (Alg. 1)	87.2 (Alg. 1 16 threads)
dense2	4000	7425226	40	0.464193	70.8 (Alg. 1)	20 (Alg. 2 15 threads)
wiki_topcats	1791489	28511807	0	8.88376e-06	6801.2 (Alg. 1)	1097.2 (Alg. 2 16 threads)
dense3	10000	49995000	1	0.5	731 (Alg. 1)	271.8 (Alg. 2 12 threads)
soc_pokec_relationships	1632803	30622564	200110	1.14861e-05	6819.4 (Alg. 1)	1305 (Alg. 1 16 threads)
soc_LiveJournal1	4847571	68993773	539119	2.93604e-06	17002 (Alg. 1)	3645 (Alg. 1 21 threads)
dense4	10000	99990000	0	1	99.4 (Alg. 1)	50 (Alg. 2 9 threads)

Table 1: Test results

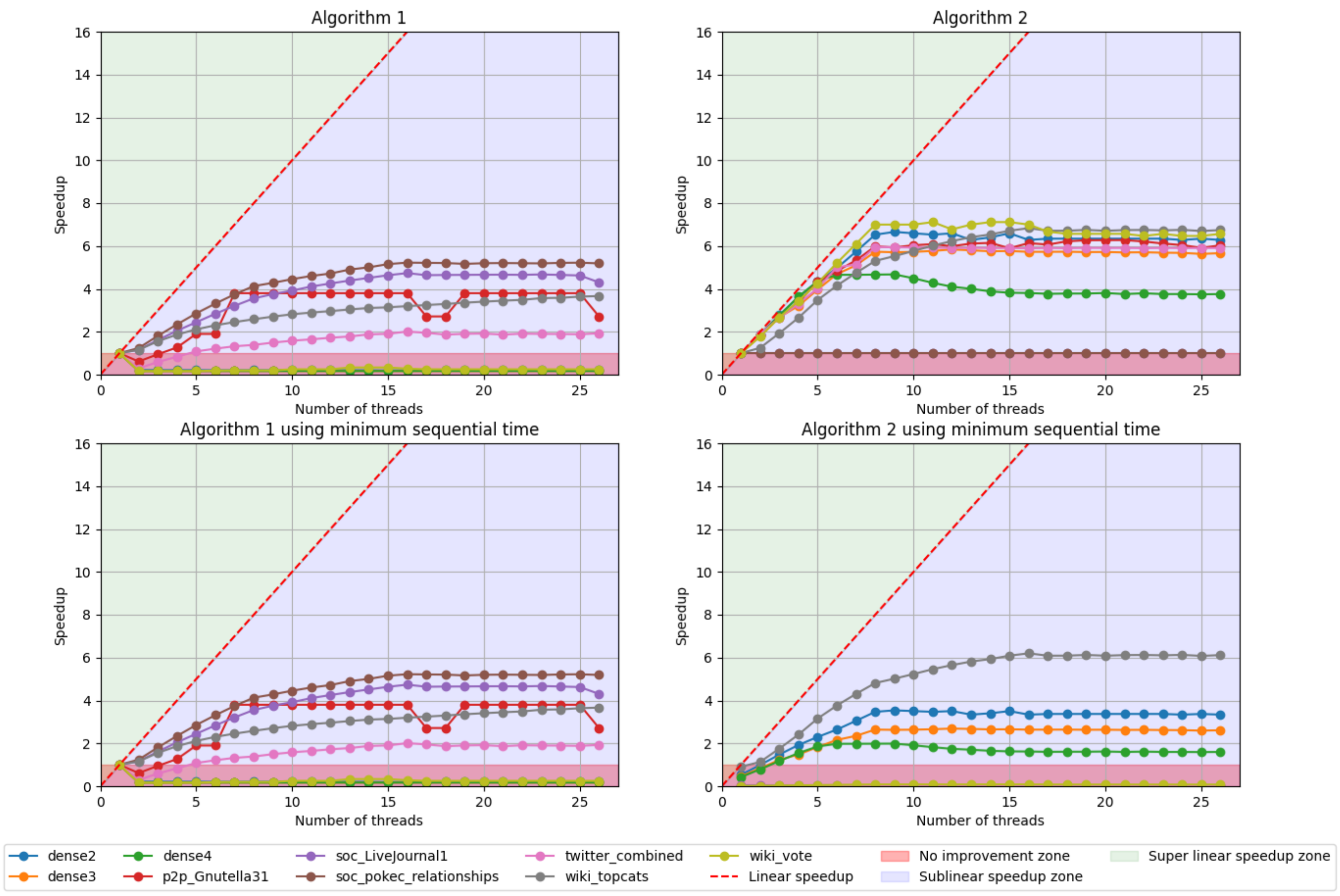

Figure 1: Speedup vs number of threads for the two algorithms.

6 Analysis

As we can see from the results, the parallel implementation of the PageRank algorithm is faster than the sequential one (unless for specific cases such as too small graphs for Alg. 1 or too sparse graphs for Alg. 2). The speedup is not linear, and it is influenced by the number of threads and the algorithm used. The sequential version of the first algorithm is faster than the second one for all the graphs, and the parallel version of the first algorithm is faster than the second one for all the graphs except for dense graphs and the ones with a small number of deadends.

Cache analysis. The reason for this is that the first algorithm has a better cache behavior than the second one. To understand this, the code was profiled using Perf and the results were analyzed. The results showed that the second algorithm has a high cache miss rate when the number of deadends is high, and this is because the second algorithm has to access the deadends vector for each node in the graph. This makes the second algorithm slower than the first one for graphs with a high number of deadends.

For example, for the graph `twitter_combined` Perf report the following cache miss rates:

- The first algorithm has a L1-dcache-miss rate of 2.05%.
- The second algorithm has a L1-dcache-miss rate of 16.44%.