Learning Over Massive Data

Assignement 1 – Page Rank

Davide Vitagliano 904120

Academic Year 2023/2024

1. Compilation

In the zip, in addition to the code files, there is the *CMakeLists.txt* used for compilation and there is the *datasets* folder containing all the graphs used during testing. Follows the steps to compile and execute the code starting inside the project directory something/LEARN_Assignment1$:

```
mkdir build && cd build && cmake .. && make
```

```
./LEARN_Assignment1 ../datasets/<filename> <n_threads>
```

Or `srun -v -l -c 20 ./LEARN_Assignment1 ../datasets/<filename> 20`

2. Algorithm

To solve the task of performing page rank, we build a class called page_rank. The class is used to represent a directed graph and the constructor read a file containing the edges of the graph. In the first pass it determines the number of nodes and edges, while in the second pass it populates two unordered_map<ulong, unordered_set<ulong>> representing the adjacency matrix but storing only non-zero values: *graph_out* row wise and *graph_in* column wise. Then the constructor builds the *oj* vector<ulong> containing the out degree of each node, computed with *graph_out*. Lastly, it initializes the *rank* vector<float> with equal rank to each node and build the transition matrix represented as a Compressed Sparse Row (CSR) matrix, computed with *graph_in*, as follows:

- For each node, it iterates over its incoming edges.
- If there is an edge i -> j, it appends a value of 1/out_degree(j) to *vals* and j to *cols*.
- If the out degree of node j is zero, instead it appends a value of 1/n to *vals* and j to *cols*.
- When all the incoming edges j of a node i are processed, it appends the size of *vals* to *rows*.

Follows the pseudocode of the core portion of the assignment, which is the computation of the matrix vector multiplication that performs the page rank:

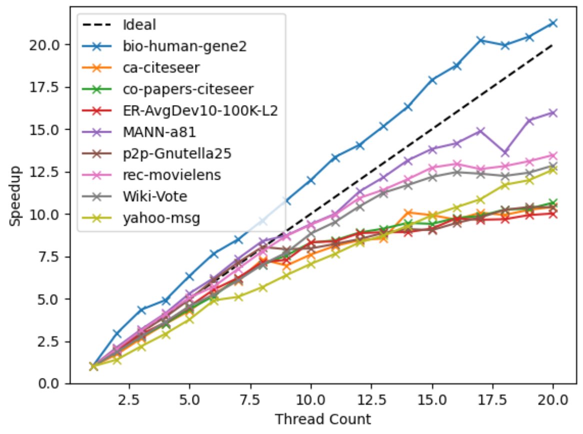
```
function compute_page_rank(n_threads, iter, beta):
    create c <- (1.0 - beta) / n
    create results[0..n-1] <- 0.0
    for k in 0..iter-1 do:
        results[0..n-1] <- 0.0
        parallel for i in 0..n-1 do:
            create sum <- 0.0
            parallel for j in rows[i]..rows[i+1]-1 do:
                sum <- sum + vals[j] * rank[cols[j]]
            results[i] <- beta * sum + c
        rank[0..n-1] <-> results[0..n-1]
    return rank
```

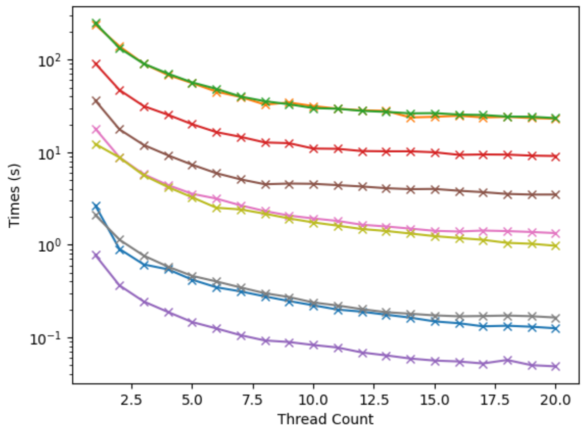
The parallel for uses a dynamic scheduling to improve scalability and load balancing, which should offset the overhead due to the synchronization. The number of threads is managed by the parameter n_threads, which means that with one thread it runs the sequential version of the algorithm. The second parallel for performs a reduction on the sum. The time complexity of the algorithm is **O(iter*n*m)** and the space complexity is **O(n+m)**, where n is the number of nodes and m is the number of edges.

3. Testing

All testings are performed on the university's cluster with 20 cores and 20 threads. The correctness of the algorithm is verified by comparing the results with the sequential version of the algorithm. The performance and scalability of the algorithm is evaluated by measuring the speedup of the parallel version with respect to the sequential version. The execution time is measured with the omp_get_wtime() function. The cache miss rate is measured with the perf tool on a local machine with a Ryzen 5 5600x (6 cores and 12 threads) and 32GB of RAM.

Name	# nodes	# edges	Density rate	Cache miss rate
bio-human-gene2	14341	9041364	0.0439648	8.55%
ca-citeseer	227321	814134	0.000015755	process killed
co-papers-citeseer	434103	16036720	0.0000851003	process killed
ER-AvgDeg10-100K-L2	100001	499359	0.0000499354	13.92%
MANN-a81	3322	5506380	0.499112	4.87%
p2p-Gnutella25	22687	54705	0.00010629	9.15%
rec-movielens	71568	10000054	0.00195241	34.40%
Wiki-Vote	8298	103689	0.00150605	14.10%
yahoo-msg	100001	6359436	0.000635937	29.53%





From the chart, we notice that lower the density, lower the speedup despite the dimension of the graph. Almost all the graphs have a sublinear speedup as expected, except the **bio-human-gene2** graph that has a superlinear speedup. One interesting observation is that it has a better speedup than the **MANN-a81** graph, despite having a lower density. This may be due to the distribution of the edges that allows a better load balancing.

Moreover, the cache miss rate is quite high for all the graphs, which is due to the random access pattern of the CSR matrix that does not exploit the cache locality. As a matter of fact, the **MANN-a81** graph has the lowest cache miss rate because of its high density. Lastly, the speedups on the local machine have relatively lower performance than the cluster, having the speedup range from 2 to 10 at a parity of graphs. This may be due to the different architectures and the number of cores; specifically, the cluster uses one thread per core while the local machine uses two threads per core.