BigData HW9

tiago.melo.antunes

May 2021

1 How to run the code

To run the GridGraph code, make sure to first compile the Makefile that was given to us. The C++ code is available in the *normal/* folder.

Run the C++ code the same way as the GridGraph code.

2 Implementation details

The C++ and GridGraph implementation have the same core algorithm logic so the only change is how the graph is loaded and iterated in each solution.

2.1 Conductance

For the implementation of conductance, the solution is quite simple, requiring only one pass through the edge set that is available. The pseudocode is the following:

```
function conductance(G) {
    crossover := 0
    red := 0
    black := 0
    for each edge E in G {
        if (E.source & 1) == (E.target & 1) {
            crossver += 1
        } else {
            if both edges are pair {
                black += 1
            } else {
                red += 1
            }
        }
    }
    return crossover / min(red,black)
}
```

Implementation	Time taken (s)
C++	2.82
GridGraph	2.29

Table 1: Time taken for Conductance algorithm

For the C++ version, the graph is loaded while calculating, so one can see the for loop in the function above as a lazy iterator over the file. This helps reduce the speedup since only one iteration over the graph is needed. Since it is done together with the IO, no memory is required to be allocated besides the counters, and no parallelism is done.

For the GridGraph version, the same algorithm is applied but the graph is loaded before and the edges are streamed as a for loop. This results in an easy implementation over the framework, without having to load the edges by hand.

The performance is quite similar. Although C++ isn't parallelizable in this case without dividing the dataset before into chunks and then reading, the fact that it can process it while loading the data will help compensate the fact that GridGraph will only run after totally loading the dataset, so the computation gets a little masked by the IO reading, achieving similar performance in this case. Perhaps the results would be different for a bigger graph.

2.2 Pagerank

For pagerank, the algorithm implemented was pretty standard. In C++, the graph is loaded first. All the ranks are initialized to an equal score, $1/numNodes$. The solution follows the scatter apply paradigm introduced in class:

```
function Pagerank(G, iterations, threshold) {
    sum := array of integers, starts initialized to zeroes
    for iter := 0 to iterations {
        // scatter
        for each edge E in G {
            val := E.source.rank / E.source.degree
            if val > threshold {
                sum[e.target] += val
            }
        }

        // apply
        for each vertex V in G {
            V.rank += 0.85 * sum[V]
            sum[V] = 0
        }
    }
}
```

Implementation	Time taken (s)
C++ (indexed by source)	60.83
C++ (indexed by target)	51.22
C++ (sorted target indexing)	50.69
C++ (vector of edges)	48.43
GridGraph	56.35

Table 2: Time taken for PageRank on 100 iterations on multiple approaches with threshold=0

For C++, two different approaches were tried out and analysed. Some parallelization has been done to improve the performance in the inner iterations over the edges and vertices using OpenMP.

1. Indexing the edges by vertex and having a vector of Vertices to represent the edge connection. Three approaches for this type were done.
   - Index by source, vector of targets. In this case, we can save the intermediate calculation of $pagerank[source]/source.rank$ to speed up the calculation. It also has the advantage of being able to skip through edges, because the update value can be calculated before. It is required to save the data of *target* and so there will be memory conflicts and atomic writes are needed to protect the update.
   - Another way to avoid the atomic update restriction would be to have the graph edges indexed by destination, and go through each destination. This has the problem of having to recalculate the values of the sources, so an extra array is used to store these intermediate results (as if a cache). Another problem is that the graph is given in a sorted way, so other implementations would be having better memory patterns during computation.
   - To fix these memory patterns, one can sort the edge vectors before computing page rank. In pratice, this gave no speedup over the the previous solution.
2. Create a vector of edge structures, which contain information about the source and targets.

After seeing the results in Table 2 and Table 3 it is clear that some specialized implementations are able to outperform GridGraph in computing a single program. They need, however, a lot more effort to achieve such results. The indexation by souce yielded the best speedup, achieving a very high speed thank to its ability of skipping nodes. The test in Table 2 shows that it is a little slower than the others (since the others will have a better memory placement) but because of the pagerank delta format, it can easily outperform for a bigger threshold value. The vector of edges managed to perform very quickly in both cases and it is probably because of the graph being sorted and giving a very

Implementation	Time taken (s)
C++ (indexed by source)	1.14
C++ (indexed by target)	47.10
C++ (sorted target indexing)	47.36
C++ (vector of edges)	4.24
GridGraph	17.45

Table 3: Time taken for PageRank on 100 iterations on multiple approaches with threshold=0.5

good memory placement. It would require a sorting to handle all cases this quick.

For all the vertex indexation methods, the parallel edge stream loop had to use a guided scheduled for load balancing, as the size of the edges would often differ between nodes.