Homework #9 – Implement Graph Algorithms with GridGraph

Yoke Kai Wen, 2020280598

May 9, 2021

1 Introduction

In this assignment, I implemented two graph algorithms, `conductance.cpp` and `pagerank_delta.cpp`, with GridGraph and C++, aiming to use the graph computing system to solve a problem.

2 Implementation details overview

Everything was completed using the GridGraph framework from https://github.com/coolerzxw/GridGraph. My algorithms were tested on the LiveJournal dataset, a directed and unweighted graph, which was preprocessed by partitioning into a 4x4 grid with the command below, following the github README.

```
./bin/preprocess -i /data/LiveJournal -o /data/LiveJournal_Grid -v 4847571 -p 4 -t 0
```

Then, I wrote the `conductance.cpp` and `pagerank_delta.cpp` scripts with the GridGraph framework, saved them under the `examples` subdirectory, and modified the `Makefile`. After `make all`, the scripts will be compiled and the applications are stored in the `bin` subdirectory. They can then be executed with the commands below, where I set the memory budget to 8GB for this assignment.

```
./bin/conductance LiveJournal_Grid 8
./bin/pagerank_delta LiveJournal_Grid 20 8
```

3 Conductance algorithm

Conductance is calculated over a cut of a graph, and intuitively measures how well-connected the two disjoint subsets (say red and black) of the graph are. After a cut, each vertex is either red or black, and each edge of the graph can be classified into three types: a crossover edge if the source and target vertices are different in colour; a red edge if both source and target vertices are red; a black edge if both source and target vertices are black. Then, conductance is calculated by:

$$conductance = \frac{\#crossover\_edges}{min\{\#red\_edges, \#black\_edges\}}$$

For this assignment, the graph vertices are classified as red or black based on the lowest bit value of the vertex ID. If $v\&1! = 0$ the vertex v is classified as red, otherwise it is black.

3.1 Implementation with GridGraph framework

I implemented the conductance algorithm with the edge streaming interface. For each edge, I check if it is a crossover, red or black edge, then increment the counts for crossover, red and black edges accordingly using the `write_add` accumulate atomic operation. Finally after iterating through all the edges, I calculate the conductance value based on the formula. The code fragment below from `conductance.cpp` shows the process.

```
int crossover_count, red_count, black_count, count;
crossover_count = red_count = black_count = count = 0;

graph.stream_edges<VertexId>(
  [&](Edge & e){
    if ((e.source&1) != (e.target&1)) {
      write_add(&crossover_count, 1);
    }
    else if ((e.source&1) !=0) {
      write_add(&red_count, 1);
    }
    else {
      write_add(&black_count, 1);
    }

    write_add(&count, 1);
    return 0;
  }, nullptr, 0, 0
);

float conductance;
if (red_count < black_count) {
  conductance = (float)crossover_count/(float)red_count;
}
else {
  conductance = (float)crossover_count/(float)black_count;
}
```

4 Pagerank Delta algorithm

The PageRank Delta algorithm is similar to the original PageRank, except that we only update vertices which have PageRank scores changing by more than a certain `propagation_threshold`. For each iteration of the pagerank delta algorithm, the following equations are computed:

$$Rank(A) = Rank(A) + Delta(A)$$

$$Delta(A) = 0.85 * (\frac{Delta(B)}{L(B)} + \frac{Delta(C)}{L(C)} + ...)$$

where $\frac{Delta(x)}{L(x)}$ is only added if it is larger than `propagation_threshold`.

4.1 Implementation with GridGraph framework

I modified the `pagerank.cpp` example code to implement the `pagerank_delta.cpp` algorithm. Other than defining the `degree, pagerank, sum` BigVectors, I also defined another `delta BigVector` to store the changes in pagerank scores. Also, I set the `prop_thresh = 0.5` arbitrarily. See code fragment below.

```
BigVector<VertexId> degree(graph.path+"/degree", graph.vertices);
BigVector<float> pagerank(graph.path+"/pagerank", graph.vertices);
BigVector<float> sum(graph.path+"/sum", graph.vertices);
BigVector<float> delta(graph.path+"/delta", graph.vertices);

float prop_thresh = 0.5;
```

Following the `pagerank.cpp` code structure, I first use the edge streaming interface to compute the out-degree of each vertex. Then, I use the vertex streaming interface to initialise the `pagerank, sum, delta BigVectors`, with pagerank scores initialised uniformly to 1/numVertices, sum initialised to zero, and delta initialised to 1. See code fragment below.

```
degree.fill(0);
graph.stream_edges<VertexId>(
  [&](Edge & e){
    write_add(&degree[e.source], 1);
    return 0;
  }, nullptr, 0, 0
);
printf("degree calculation used %.2f seconds\n", get_time() - begin_time);
fflush(stdout);

graph.hint(pagerank, sum, delta);
graph.stream_vertices<VertexId>(
  [&](VertexId i){
    pagerank[i] = 1.f / (float)graph.vertices;
    sum[i] = 0;
    delta[i] = 1.0f;
    return 0;
  }, nullptr, 0,
  [&](std::pair<VertexId,VertexId> vid_range){
    pagerank.load(vid_range.first, vid_range.second);
    sum.load(vid_range.first, vid_range.second);
    delta.load(vid_range.first, vid_range.second);
  },
  [&](std::pair<VertexId,VertexId> vid_range){
    pagerank.save();
    sum.save();
    delta.save();
  }
);
```

Then, for each iteration, I first use the edge streaming interface to accumulate the delta scores divided by out-degree from each source vertex into its target vertex in the `sum BigVector`, if the fractional score change is larger than `prop_thresh`. Then, I use the vertex streaming interface to update the pagerank score of each vertex with the delta scores. I also reset the `sum BigVector` to zero for the next iteration. See code fragment below.

```
for (int iter=0;iter<iterations;iter++) {
  graph.hint(pagerank, delta);
  graph.stream_edges<VertexId>(
    [&](Edge & e){
      if (delta[e.source]/degree[e.source] > prop_thresh) {
        write_add(&sum[e.target], delta[e.source]/degree[e.source]);
      }
      return 0;
    }, nullptr, 0, 1,
    [&](std::pair<VertexId,VertexId> source_vid_range){
      delta.lock(source_vid_range.first, source_vid_range.second);
```

```
    },
    [&](std::pair<VertexId,VertexId> source_vid_range){
      delta.unlock(source_vid_range.first, source_vid_range.second);
    }
  );
  graph.hint(pagerank, sum, delta);
  graph.stream_vertices<float>(
    [&](VertexId i){
      delta[i] = 0.85f * sum[i];
      pagerank[i] += delta[i];
      sum[i] = 0;
      return 0;
    }, nullptr, 0,
    [&](std::pair<VertexId,VertexId> vid_range){
      pagerank.load(vid_range.first, vid_range.second);
      sum.load(vid_range.first, vid_range.second);
      delta.load(vid_range.first, vid_range.second);
    },
    [&](std::pair<VertexId,VertexId> vid_range){
      pagerank.save();
      sum.save();
      delta.save();
    }
  );
}
```

5 Performance analysis

The outputs from running `conductance.cpp, pagerank_delta.cpp` on the preprocessed 4x4 LiveJournal grid with the given server machine are shown below, with memory limited to 8GB. Both had short runtimes of roughly 6s despite the size of the graph dataset. I also printed more outputs for each algorithm to check for correctness.

```
./bin/conductance LiveJournal_Grid 8
conductance: 2.005733, cross_count: 34486389, red_count: 17313478, black_count: 17193906, count: 68993773
conductance calculation used 13.12 seconds

./bin/pagerank_delta LiveJournal_Grid 20 8
degree calculation used 0.49 seconds
20 iterations of pagerank took 11.30 seconds
Max: 471.902222, index: 214356
```

Overall, this exercise demonstrates that single machine out-of-core graph processing is a powerful tool.