COL380 A2

Sarthak Panda (2022CS51217)

March 2025

1 O0: Brief Discussion on Basic Implementation

The basic idea involves gathering the vertex-to-color map from all the other processes at each execution thread. Then, using this vertex-to-color map in each execution thread to compute the partial degree centrality, which is further accumulated at the root process and returned. For the first half, to gather the vertex color map at each execution thread, we used `MPI_Allgatherv` and for the second half, to gather/accumulate the degree centrality, we used `MPI_Gatherv`. (The 'v' suffix to each MPI function indicates that we are accumulating variable sizes from each execution thread.)

2 List of optimizations performed

2.1 O1: Optimization of Message Structure

In my base implementation, I was using a nested map structure to store the partial degree centrality where the outer map had color as its key and the inner map had vertex as its key. As opposed to the case where the outer map's key corresponds to vertex and the inner map's key corresponds to color, here we can save a lot of space by sending the color, then a delimiter (say, -9), and vertex, partial degree pairs corresponding to that color.
Earlier Message Structure from each Process: C1, V1, P1, C2, V2, P2, ...
Updated Message Structure: C1, -9, V1_1, P1_1, V1_2, P1_2, ..., C2, -9, V2_1, P2_1, ...
This reduces the size of the message and, theoretically, it should improve the performance in both aspects of space/memory and time.

2.2 O2: Optimization of Sorting Algorithm

Consider the second half when constructing the entire degree centrality map (the nested map described earlier). For each color, we keep account of non-degree nodes. But the problem needs only top-k for each color, so rather than sorting the entire list for each color, we can try to use some `partial_sort` algorithm. We can use either a heap structure to partially sort which will result in $O(n \log k)$ complexity; otherwise, we can use the `nth_element` functionality in `C++` which takes an array and k as input and places the top k elements in the first k places (in an unsorted manner), which takes $O(n)$, and then sort these first k elements which would take $O(k \log k)$, resulting in an overall time complexity of $O(n + k \log k)$. For $k \ll n$ (i.e., $k \ll |V|$ given in the statement), the second approach to perform partial sorting would theoretically be better performing.
Note: Current optimization also includes all the previous/above optimizations discussed.

2.3 O3: Utilization of Vector Data Structure over Map-based Implementation and Compiler Optimization Flags

If we note, after the first half when we accumulate the vertex-to-color mappings, we can also keep track of all colors in an ordered set simultaneously. We can then iterate this ordered set to create a virtual color map (which maps the original color to the range 0 to $|C| - 1$). This is possible because we are only concerned about the ordering of color values and not the absolute values of colors. These virtual colors are then used to store the partial degree centrality and degree centrality maps. Furthermore, we know vertex labeling is in the range 0 to $|V| - 1$. We can preallocate to create a 2D-array structure instead of a nested map. Since the upper bounds on $|V|$ and $|E|$ clearly indicate a dense graph, this is very likely to perform better than the map structure where each insertion and search will take $O(\log n)$ (which is slower than $O(1)$ search and insertion in a preallocated array). (As a note, a preallocated array would have been worse compared to a nested map if the bound on $|V|$ were much larger than the upper bound on $|E|$, i.e., the case of a sparse graph.) Furthermore, I added compiler flags such as `-O3`, `-funroll-loops`, and `-march=native` in the Makefile to improve performance.
Note: Current optimization also includes all the previous/above optimizations discussed.