COL380_A2 Report

My program is made to calculate degree centrality in an undirected graph given a graph and an MPI system to handle parallelism. We wish to calculate centrality scores of each vertex by considering the color of the neighborhood of the vertex and then identify the top k vertices by each color. We began by breaking the graph into pieces such that each process would have a sub-mapping of the vertex–color mapping and a sub-edge list. The graph being spread across several MPI processes, the initial challenge was to create an entire global vertex–color mapping such that every process is able to properly include its local edges irrespective of what process initially retains the color info of a vertex.

To achieve this, we used MPI_Allgather and MPI_Allgatherv to distribute the local vertex–color pairs to all the processes. In this manner, each process builds the global mapping, allowing them to look up the color of any vertex during calculation of the degree centrality. Once we build the global mapping, each process computes its local contribution to the degree centrality by iterating over its partial edge list. For each edge (u, v), the code increments the count of u for the color of v and vice versa, to maintain the undirected nature of the graph. Once local computation is done, we store the centrality information in a flat vector and use MPI_Gather and MPI_Gatherv to accumulate all the local contributions at the root process. The root accumulates the centrality counts per vertex and sorts the vertices for each unique color in descending order of centrality, with ties broken by vertex id in ascending order, and then selects the top k nodes per color.

A number of optimizations were done to achieve the best performance without sacrificing correctness. First, we minimized memory reallocations by pre-allocating the needed capacity for

vectors such as the local vertex–color vector and the data vector for local centrality counts. This reduces the cost of dynamic memory allocation in computation. Second, we replaced the standard library map with an unordered_map for local aggregation of centrality counts. This reduces lookup and insertion time significantly because hash-based containers offer constant average time complexity as opposed to tree-based maps' logarithmic complexity. Third, we optimized the loop for processing edges by caching the result of the global vertex–color lookup so as not to duplicate searches for the same vertex during computation.

Performance-wise, the optimizations minimize both computation overhead and communication latency. By preallocating memory and utilizing hash maps, local process time per process is reduced, which is essential considering the graph size is enlarged. The usage of MPI_Allgather and MPI_Allgatherv, though essential in building the global mapping, is optimized by efforts to compute displacements and prevent redundant copying of data. In all, my design promotes good distributed workload among processes, and aggregation cost for the root process is maintained at reasonable levels. The performance gains are most pronounced when handling large graphs with high vertex and edge numbers, where enhanced local computation efficiency translates directly to shorter total execution times. In summary, my approach effectively calculates the necessary degree centrality values in parallel with careful attention to correctness and performance. Optimizations used—both in memory management and in data structure selection—lead to an efficient implementation that scales with growing graph sizes and number of MPI processes.