COL380 A2:- Parallel Graph Centrality Computation using MPI

Yash Bansal (2022CS51133)

1 Base Code

The following steps outline the computation of degree centrality per node in an undirected graph using MPI.

1. **Initialize MPI:** Each process initializes MPI using `MPI_Init()` and retrieves its rank and total number of MPI processes.
2. **Distribute Vertex-Color Mapping:**
   (a) Each MPI process has access to a subset of the vertex-color mapping and partial edge list of the graph.
   (b) Since each process requires the complete vertex-color mapping at some point, we first gather and broadcast all local mappings to ensure every process has the full vertex-color mapping.
3. **Gather and Broadcast Full Vertex-Color Mapping:**
   (a) Each process sends its local vertex-color mapping to the root process using `MPI_Gather()`.
   (b) The root process merges these mappings and broadcasts the complete mapping to all processes using `MPI_Bcast()`.
4. **Compute Partial Degree Centrality:**
   (a) Each process computes the degree centrality for its subset of vertices based on its local edge list.
   (b) The result is stored as a *partial degree map*, which maps each vertex to a sub-map tracking the count of neighbours per colour.
5. **Merge Partial Degree Maps:**
   (a) Each process converts its local degree map to a vector format and sends it to the root process using `MPI_Gather()`.
   (b) The root process merges all partial maps to construct the final degree centrality data.
6. **Determine Top-k Influential Nodes per Color (Root Process Only):**
   (a) The root process sorts the vertices based on degree centrality for each color.
   (b) It selects the top-k nodes with the highest degree per colour.
7. **Finalize MPI:** All processes call `MPI_Finalize()` to conclude the MPI session.

2 Optimizations

The following optimizations were implemented to improve execution time, as detailed in the report and recorded in `data.csv`. Performance was measured for graphs of varying sizes and averaged over 3 runs for a test case:

- $V = 40, E = 160, C = 2, k = 10$
- $V = 1000, E = 20000, C = 5, k = 10$ and 100
- $V = 10000, E = 5000000, C = 50, k = 100$ and 1000
- $V = 100000, E = 20000000, C = 50, k = 100$ and 1000

The number of nodes was varied from 1 to 4, with 2 MPI processes per node.

2.1 Optimization 1: Efficient Storage of Vertex-Color Mapping

1. Initially, vertex-color mappings were stored in maps. Since vertex indices range from 0 to $n - 1$, we replaced maps with vectors, reducing lookup complexity from $O(\log n)$ to $O(1)$.
2. This change reduced the broadcast data size from $2n$ to n and improved cache efficiency due to contiguous memory storage in vectors.
3. Similarly, the degree centrality map was changed from a map of maps to a vector of maps.
4. **Performance Improvement:** Execution time decreased by up to **55-65%** for large graphs, significantly improving efficiency for multi-node execution.

2.2 Optimization 2: Using Unordered Maps for Faster Lookups

1. Since maps (based on BSTs) have $O(\log n)$ complexity, we replaced them with unordered maps, reducing access time to $O(1 + \alpha)$.
2. Unordered maps are more cache-efficient than regular maps, further enhancing performance.
3. **Performance Improvement:** Execution time was reduced by **20-30%** compared to regular maps, particularly benefiting larger graphs.

2.3 Optimization 3: Mapping Colors to a Contiguous Range

1. Initially, colours were stored in arbitrary mappings, requiring maps or unordered maps for lookups.
2. We remapped colors to a contiguous range $[0, \text{num\_colors} - 1]$ at the root process before broadcasting.
3. This allowed storage of color-to-count mappings as a vector instead of a map, improving lookup time from $O(\log n)$ to $O(1)$ and enhancing cache efficiency.
4. After computations, the inverse mapping was applied to restore the original colour names.
5. **Performance Improvement:** Speedup of **5-10×** was observed for large graphs, as vector lookups outperformed map-based operations.

2.4 Optimization 4: Compiler Optimizations

The following compiler flags were used for performance improvements:

1. **-O3:** Enables aggressive optimizations, such as loop unrolling and vectorization.
2. **-march=native, -mtune=native:** Enables CPU-specific optimizations.
3. **-flto:** Performs link-time optimizations across translation units.
4. **-funroll-loops:** Expands loops to reduce loop control overhead.
5. **-ftree-vectorize:** Enables automatic loop vectorization.
6. **Performance Improvement:** Achieved a **2-5× speedup** by utilizing CPU-specific instruction sets and reducing redundant instructions.

2.5 Optimization 5: Flattening Data Structures

1. Previously, the degree centrality was stored as a vector of vectors.
2. Since colour indices are now contiguous, we flattened the structure into a single vector of size num_vertices × num_colors.
3. This improved cache locality, reduced pointer overhead, and accelerated MPI communication.
4. **Performance Improvement:** Execution time reduced by **30-50%**, with significant gains in distributed environments.

2.6 Optimization 6: Removing Redundant Reverse Mapping

1. Initially, reverse mapping was required to restore the original colour names.
2. Since the output only requires an ascending order of colours, we sorted the colours at mapping time and eliminated the need for reverse mapping.
3. **Performance Improvement:** Reduced unnecessary computations, leading to a **10-20%** performance boost.

2.7 Optimization 7: Using `MPI_Reduce` for Merging

1. Initially, merging was done by gathering all partial results at the root process and looping through them.
2. We replaced this with `MPI_Reduce` using the `MPI_SUM` operation, which performs the reduction in a tree-based manner, significantly improving efficiency.
3. **Performance Improvement:** Speedup of **10-20%** for large graphs, reducing communication overhead and improving scalability.

3 Overall Performance Analysis

3.1 Effect of Input Size on Execution Time

- **Base Code (O0)**: Execution time increased drastically with input size, reaching over 87,000 ms for large graphs.
- **Optimized Code (O7)**: Execution time for the largest input reduced to 500 ms on a single node, achieving a 165× improvement over the base code.

3.2 Effect of Number of Nodes on Performance

- **Small Inputs ($V = 40$)**: Scaling was irregular due to negligible workload per process.
- **Medium Inputs ($V = 1000, 10000$)**: Speedup was observed up to $1.5 - 2\times$ as nodes increased, with communication overhead becoming noticeable.
- **Large Inputs ($V = 100000$)**: Near-ideal parallel speedup was observed, with execution time reducing from 900 ms (1 node) to 300 ms (4 nodes), achieving over 3× improvement compared to the base code on a single node.

3.3 Final Summary

The combination of efficient data structures, compiler optimizations, and parallel communication improvements led to an overall $100 - 240\times$ speedup for large-scale inputs compared to the base implementation.

Parallel scalability was effective, with 4-node execution time being $2 - 3\times$ lower than 1-node execution for most cases.