1. Project Title:
Hamiltonian Cycle based Traveling Salesman Problem Using OpenMP & MPI

Group Number: 14 (according to Project List uploaded on GCR)
Group Members:

- 22k-4173
- 22k-4406
- 22k-4625

2. Code Repository and Selection:

Algorithm Explanation:
https://math.libretexts.org/Bookshelves/Applied_Mathematics/Book%3A_College_Mathematics_for_Everyday_Life_(Inigo_et_al)/06%3A_Graph_Theory/6.04%3A_Hamiltonian_Circuits

Sample GitHub Repository: The code is original hence no repository is provided.

Code Description:
This project focuses on solving the Hamiltonian Cycle problem using the nearest neighbor heuristic. The Hamiltonian Cycle problem involves finding a cycle that visits each vertex exactly once in a given graph and returns to the starting point. This is an NP-complete problem, making it computationally intensive, especially for large graphs.
The nearest neighbor approach provides a greedy solution, where the algorithm starts at a random vertex, then iteratively visits the nearest unvisited vertex until all vertices have been visited. While this method doesn't guarantee the optimal solution, it provides a fast approximation and serves as a good candidate for parallelization.

Complexity Analysis:
The nearest neighbor algorithm for the Hamiltonian Cycle has a complexity of $O(n^2)$ in its sequential form, where n is the number of vertices in the graph. Each iteration involves calculating the distance to every unvisited vertex, resulting in quadratic time complexity. For large graphs (e.g., n = 1000), the execution time can be significant when using a single thread.
The goal of this project is to parallelize the nearest neighbor approach to reduce execution time, enhance scalability, and improve efficiency. The existing code is initially sequential and will first be parallelized using OpenMP for shared-memory parallelism on a single machine. Once the OpenMP parallelization is complete, the code will be separately converted to MPI for distributed-memory parallelism, enabling it to handle larger graphs across multiple machines.

3. Parallelization Strategy:

OpenMP Implementation:

- **OpenMP Parallelization:** In the first step, the code will be parallelized using OpenMP. Each thread will independently compute the cost of a Hamiltonian cycle starting from a different vertex. The computation for each vertex will be performed independently, where the task is to calculate the cycle that starts from that vertex using the nearest neighbor heuristic. The #pragma omp parallel for directive will be used to parallelize the computation for different vertices. This allows multiple threads to calculate the Hamiltonian cycle cost for different vertices simultaneously.

Challenges:

- One of the key challenges during OpenMP parallelization will be managing synchronization when multiple threads access shared data, such as the list of unvisited vertices. However, since each thread operates independently on a different vertex, minimal synchronization will be required.
- Load balancing will also need to be considered to ensure that all threads have a roughly equal share of work, and that no thread is left idle.

MPI Implementation:

- **MPI Parallelization:** After the OpenMP parallelization is complete, the code will be separately parallelized using MPI. The graph will be distributed across multiple processes, where each process will independently compute the cost of a Hamiltonian cycle starting from a different vertex. Each process will work independently on its assigned vertex, calculating the full cycle cost. Communication between processes will be necessary to distribute vertices and gather results when all cycles have been computed.

Challenges:

- Communication overhead in MPI will be minimized, as the processes will work independently on different vertices, with each process calculating its own cycle.
- Ensuring that all processes have a balanced workload, especially in the case of a large number of vertices, will be crucial for optimal performance.

Code Segmentation:

- **Vertex Assignment:** The graph's vertices will be divided into segments, with each thread (in OpenMP) or process (in MPI) being assigned a unique vertex to compute the Hamiltonian cycle. This distribution ensures that each thread/process works independently on its assigned vertex, calculating the cycle's total cost without interference from other threads/processes.
- **Nearest Neighbor Search:** For each vertex, the nearest neighbor search will be performed independently. The algorithm will begin by selecting the current vertex as the starting point of the cycle and will then iteratively select the nearest unvisited vertex as the next one to visit, until all vertices have been visited. This search is done independently by each thread/process. Since the threads/processes are working independently, there will be no need for synchronization or data sharing during the search phase, aside from the initialization phase.
- **Cycle Completion:** After completing the nearest neighbor search for a vertex, the cycle will be returned as the computed Hamiltonian cycle for that starting vertex. Each thread or process will compute the cycle's cost independently, and once all threads/processes have completed their tasks, the results will be gathered (in the case of MPI) or remain within the shared memory (for OpenMP).
- **Cost Calculation:** The total cost of each cycle will be computed based on the distances between consecutive vertices in the cycle. Each process/thread will calculate this cost for its assigned cycle independently, using the distances stored in a matrix or an adjacency list.
- **Final Results:** After all cycles are computed, the results will either be gathered in the main process (MPI) or be available directly (OpenMP) to determine the best cycle, which can be chosen based on the lowest cost.

.

4. Execution Plan:

Hardware Specifications:
The hardware consists of 2 cores, and 8 GB RAM.

Baseline Execution:

- **Single-Thread Execution:**
  We will run the code on a single thread to establish a baseline time, expected to be approximately 20-30 minutes on a matrix size of (n = 8000).
- **Parallel Execution:**
  - **Testing Scenarios:**
    Performance testing will involve varying the number of MPI processes (e.g., 1, 2, 4) and OpenMP threads (e.g., 2, 4, 8 per process).
    We will also test on varying input sizes (n = 1000), (n = 3000), (n = 6000) to analyze scaling effects.

5. Data and Performance Metrics:

Data Size:
Testing will involve large matrices with sizes up to (n = 8000) to ensure complexity and effectiveness in parallelization.

Performance Metrics:

- **Execution Time:** Measure total execution time for both sequential and parallel versions.
- **Speedup:** Calculate speedup achieved by parallelization.
- **Efficiency:** Assess efficiency across multiple cores and nodes.

6. Numerical Results and Visualization:

Expected Results:
We expect a significant reduction in execution time—from 20-30 minutes sequentially to half or so of the sequential time when optimized with OpenMP and MPI.

Graphical Representation:

- **Visualization Tools:** We will use Excel and/or WPS Sheets to create all line graphs for visualizing the results.
- **Execution Time Comparison:** Line graphs will show the execution times of both the sequential and parallelized code using different combinations of MPI processes and OpenMP threads across varying matrix sizes (n = 1000, 2000, 5000, 8000).
- **Speedup Analysis:** Line graphs will depict the speedup factor achieved through parallelization, illustrating the reduction in execution time as more processes and threads are introduced.
- **Efficiency Analysis:** Another set of line graphs will show efficiency metrics, highlighting the relationship between the number of processes/threads used and the effectiveness of resource utilization.

7. Testing and Validation:

Testing Methodology:

- **Correctness Validation:** Validate against baseline output and test on small matrices.
- **Stress Tests and Edge Cases:** Large matrix stress tests will confirm stability and edge-case handling.
- **Validation of Results:** Performance gains will be validated by comparing baseline and parallel results, ensuring consistent accuracy across output.

8. Contribution of Group Members:

- **22k-4173:** Handles code conversion to MPI.
- **22k-4406:** Responsible for implementing OpenMP code segments.
- **22k-4625:** Responsible for documentation, numerical result analysis, and visualization.

9. Plagiarism and Originality Declaration:

This project is original and all programs, serial and parallelized, have been coded from scratch.