CS3006 Parallel and Distributed Computing

Assignment 2-MPI

Deadline 12th of March

Instructions:

- For your assignment, make a code file and pdf file with Roll Number. For e.g. make ROLL-NUM_SECTION_A1.cpp (23i-0001_A_A1.c) and so on. Each file that you submit must contain your name, student-id, and assignment # on top of the file in comments.
- Combine all your work in one folder. The folder must contain only code and pdf file (no binaries, no exe files etc.).
- Rename the folder as ROLL-NUM_SECTION (e.g. 23i-0001_A) and compress the folder as a zip file. (e.g. 23i-0001_A.zip). Do not submit .rar file.
- All the submissions will be done on Google classroom within the deadline. Submissions other than Google classroom (e.g. email etc.) will not be accepted.
- The student is solely responsible to check the final zip files for issues like corrupt files, viruses in the file, mistakenly exe sent. If we cannot download the file from Google classroom due to any reason, it will lead to zero marks in the assignment.
- Displayed output should be well mannered and well presented. Use appropriate comments and indentation in your source code.
- **Be prepared for viva or anything else after the submission of assignment.**
- If there is a syntax error in code, zero marks will be awarded in that part of assignment.
- Understanding the assignment is also part of assignment.
- **Zero marks will be awarded to the students involved in plagiarism. (Copying from the internet is the easiest way to get caught).**
- **Late Submission policy will be applied as described in course outline.**
- **Use MPI to implement the task only**
- **Non-running code will be awarded zero marks**

Tip: For timely completion of the assignment, start as early as possible.
Note: Follow the given instruction to the letter, failing to do so will result in a zero.

Assignment # 2

Parallel and Distributed Processing (CS & AI)

1. **Implement the following problem scenario using MPI functions. You can use point to point and any collective functions?**

"Travelling Salesman Problem" (TSP). The TSP involves finding the shortest possible route that visits a set of cities and returns to the starting city, without visiting any city more than once. It is a classic optimization problem with numerous real-world applications, such as logistics and transportation. Parallelizing the TSP can provide significant performance improvements, especially for large problem instances. Represent the TSP as a graph, where cities are nodes, and edges represent the distances between cities. Choose an appropriate data structure for efficient graph representation, such as adjacency matrices or adjacency lists.

Parallel algorithm design: Design a parallel algorithm for solving the TSP using MPI. Divide the problem into **sub-problems that can be solved independently**. One approach is to partition the cities among multiple computing nodes, where each node solves the TSP for its assigned subset of cities.

Parallel search: Implement a parallel search strategy to **explore different routes simultaneously**. This can be done using techniques like parallel depth-first search or parallel branch-and-bound. Each computing node explores a portion of the solution space independently, eliminating the need for inter-node communication during the search.

Inter-node communication: Depending on the parallel algorithm design, you may need to incorporate inter-node communication for exchanging information or sharing intermediate results. This can involve techniques like **message passing or shared memory**, depending on the parallel programming framework you choose.

Performance evaluation: **Evaluate the performance of the parallelized TSP** solver in terms of runtime, speedup, and scalability. Compare the results with a sequential TSP solver on the same problem instances.

Write a **report summarizing your findings**, including the analysis of the performance metrics and the insights gained from the experiment.

2. **Implement the following functions using point to point (ptop) functions ONLY**.

- **MPI_Allgather**
    - **Similar** to **MPI_Gather**, but the **result** is **available** to all **processes**
- **MPI_Allgatherv**
    - **Similar** to **MPI_Gatherv**, but the **result** is **available** to all **processes**

- **MPI_Alltoall**
    - **Similar** to **MPI_Allgather**, each process performs a **scater** followed by **gather** process
- **MPI_Alltoallv**
    - **Similar** to **MPI_Alltoall**, but **messages to different processes** can have **different length**