Muhammad Talha i220839 CS-H

Ahmad Haroon i221295 CS-H

Parallel Permutation Tree Generation using OpenMP and MPI

Objective

This program aims to compute parent-child relationships in a permutation-based tree structure using parallel computing. It efficiently distributes the workload across multiple CPU cores and processes using **OpenMP (multithreading)** and **MPI (multiprocessing)**.

Key Features

1. **Hybrid Parallelism**:
   - **MPI** handles process-level parallelism for distributing work across machines or cores.
   - **OpenMP** is used within each process for multithreaded computation.
2. **Permutation Encoding/Decoding**:
   - encode_permutation() converts a permutation into a unique integer ID.
   - decode_permutation() reverses the encoding to retrieve the permutation.
3. **Tree Construction Logic**:
   - Each node in the tree represents a permutation.
   - The algorithm determines each node's parent using swap-based logic and permutation manipulation based on specific cases.
4. **Work Distribution**:
   - The total workload is divided among MPI processes (world_size) and further divided among OpenMP threads within each process.
   - Each process computes its portion of results and sends them to the root process for aggregation using MPI_Gatherv.
5. **Performance Measurement**:
   - The total execution time is measured using MPI_Wtime() to evaluate parallel efficiency.

The Code:
1. libraries used:

- #include <iostream>
- #include <vector>
- #include <string>
- #include <algorithm>
- #include <numeric>
- #include <omp.h>
- #include <mpi.h>

vector, algorithm, numeric are used for permutation storage and manipulation.

omp.h and mpi.h enable hybrid parallel execution.

---

2. Permutation Encoding & Decoding

- **decode_permutation(int id, int n, vector<int>& f)**
    - Converts an integer ID to a permutation of size n using factorial number system.
    - **remainder vector** helps track available digits.
    - Uses: block_size = f[(n-1) - i] to partition the ID space.
- **encode_permutation(vector<int>& p, int n, vector<int>& f)**
    - Reverses the above: converts a permutation back to an integer ID.
    - Uses: j * f[(n-1) - i] to build the final id.

---

3. Swap and Position Logic

- **swap(const vector<int>& v, int i)**
    - Performs a single right swap for element i in the permutation v.
- **r(const vector<int>& v)**
    - Returns the largest index i where v[i-1] != i. Used in positional decisions.
- **find_position(...)**
    - A key decision-making function that checks specific values in the permutation and applies swap rules accordingly.

Output

- The final result is a matrix of parent IDs for all nodes in the permutation tree.
- The runtime performance of the parallel execution is printed.

Applications

- Efficient enumeration of tree structures in combinatorics.
- Useful for analyzing permutations in distributed systems, parallel graph algorithms, and computational mathematics.

Challenges and Solutions (Code-Level)

1. **Permutation-ID Mapping Errors**
   Used encode_permutation() and decode_permutation() functions with factorial logic to ensure bijective mapping.
2. **Uneven MPI Workload Distribution**
   Implemented balanced chunking using chunk_size, start_k, and extra calculations.
3. **OpenMP Race Conditions**
   Ensured thread-safe writes with #pragma omp for schedule(static) for parallel loop control.
4. **High Latency in MPI Gather**
   Replaced MPI_Gather() with MPI_Gatherv() and calculated recvcounts and displs.
5. **Incorrect Conditional Swap Logic**
   Refactored decision tree logic for swaps using clearly structured if-else based on vn, v[n-2], etc.
6. **Factorial Overflow for Larger n**
   Used long long and precomputed factorials in factorial_arr[].
7. **Debugging Parallel Execution**
   Used #ifdef DEBUG blocks and rank/thread-specific logging to isolate issues.
8. **Thread-MPI Data Sync Conflicts**
   Isolated OpenMP local buffers (local_ids) per MPI process before synchronized global collection.
9. **Incorrect Permutation Decoding**
   Validated permutations via round-trip tests (encode → decode → encode).

10. **Difficulty in Edge Case Handling**
    Added boundary checks and assertions to prevent invalid memory access or logic errors.