Graphical user interface

Description automatically generated with medium confidence

# **PARALLEL AND DISTRIBUTED COMPUTING**

# PROJECT REPORT

# AN ASSIGNMENT PRESENTED TO:

# SIR AADIL UR RREHMAN

|  |  |  |
| --- | --- | --- |
| SR. | NAME | ROLL NO. |
| 1. | M. ZAYYAM HASSAN | 22I-1247 |
| 2. | M. RAYYAN | 22I-1022 |
| 3. | UMER FAROOQ | 22I-1007 |

# GROUP MEMEBERS

**Serial Implementation of the Bubble Sort Network (Infinite Spanning Trees)**

**1. Introduction**

The goal of this project is to generate a family of **Infinite Spanning Trees (ISTs)** for the symmetric group **Sₙ** (the set of all permutations of \*n\* elements). These trees are structured such that each permutation has a unique path back to the **identity permutation** (sorted order). This structure is fundamental in constructing **distributed sorting networks**, where multiple parallel swaps can be performed efficiently.

The serial implementation:

* Generates all possible permutations of \*n\* elements.
* Constructs \*n-1\* distinct spanning trees (**T₁, T₂, ..., Tₙ₋₁**).
* For each tree **Tₜ**, defines parent-child relationships between permutations based on specific swap rules.
* Outputs the tree structure in **DOT format** for visualization.

**2. Key Concepts**

**2.1 Permutations and the Symmetric Group (Sₙ)**

* A **permutation** is an arrangement of elements (e.g., for \*n=3\*, possible permutations are [1,2,3], [1,3,2], [2,1,3], etc.).
* The **identity permutation** is the sorted order [1,2,3,...,n], which serves as the root of all spanning trees.

**2.2 Spanning Trees (T₁ to Tₙ₋₁)**

* Each tree **Tₜ** defines a unique way to "sort" permutations back to the identity by applying **adjacent swaps**.
* The rule for moving from a permutation to its parent depends on:
  + The current permutation structure.
  + The tree index **t** (each tree has a different swap strategy).

**2.3 Parent-Child Relationships**

* **Parent**: The permutation obtained by swapping two adjacent elements (based on tree rules).
* **Children**: All permutations that can reach the current permutation via one valid swap.
* The goal is to ensure **no cycles** and that every permutation eventually reaches the identity.

**3. Algorithmic Flow**

**3.1 Generating All Permutations**

1. **Enumerate all *n!* permutations** using a systematic permutation generator.
2. **Map each permutation to a unique index** for efficient lookup.

**3.2 Building the Trees (T₁ to Tₙ₋₁)**

For each tree **Tₜ** (\*t\* ranges from 1 to \*n-1\*):

1. **Define the Parent Rule**:
   * If the permutation is already the identity, it remains its own parent.
   * If the last element is **n**, apply a special case to avoid cycles.
   * If the last element is **t**, swap it with its neighbor.
   * Otherwise, perform a standard adjacent swap based on **t**.
2. **Construct Parent-Child Relationships**:
   * For each permutation, compute its parent in **Tₜ**.
   * Store children of each permutation (all nodes that have it as a parent).

**3.3 Outputting the Tree Structure**

* The trees are saved in **DOT format** (Graphviz) for visualization.
* Each node represents a permutation, and edges show parent-child relationships.

**4. Implementation Details**

**4.1 Data Structures Used**

* **Permutation**: A vector of integers representing a permutation.
* **Hash Map**: Maps permutations to unique indices for O(1) lookups.
* **Children List**: Stores children for each permutation in each tree.

**4.2 Key Functions**

1. getIdentity(n) → Returns [1,2,...,n].
2. findPosition(perm, x) → Finds where element \*x\* is in the permutation.
3. Swap(perm, x) → Swaps \*x\* with its right neighbor.
4. getParent(perm, t, n) → Computes the parent in tree **Tₜ**.
5. generate(n, perms, children, permIndexMap) → Constructs all trees.

**4.3 Optimization Considerations**

* **Precomputing all permutations** avoids redundant generation.
* **Hash-based indexing** speeds up parent-child lookups.
* **Tree structures are stored compactly** using vectors.

**5. Expected Output**

For \*n=4\*, the program generates:

* **3 trees (T₁, T₂, T₃)**.
* **24 permutations** (since 4! = 24).
* **DOT files (**tree\_T\_1.dot**,**tree\_T\_2.dot**,**tree\_T\_3.dot**)** for visualization.

**Parallel Bubble Sort Network Implementation (Hybrid MPI + OpenMP)**

**1. Core Objective**

Generates n-1 Infinite Spanning Trees (ISTs) for permutations of n elements, where each tree defines a unique sorting path via adjacent swaps.

Key Features:

* MPI for distributed-memory parallelism (multi-node)
* OpenMP for shared-memory parallelism (multi-thread)
* Optimized data distribution (flattened permutations)
* DOT file output for Graphviz visualization

**2. Algorithmic Workflow**

**2.1 Initialization (MPI Rank 0)**

* Reads input n (2 ≤ n ≤ 10)
* Generates all n! permutations using std::next\_permutation
* Flattens permutations into 1D array for efficient MPI broadcast

**2.2 Data Distribution (MPI\_Bcast)**

* Broadcasts flattened permutations to all ranks
* Each rank reconstructs the vector<Permutation> locally

**2.3 Parallel Tree Construction**

* Tree Assignment: Each rank processes trees where (t-1) % size == rank
* OpenMP Parallelization:
  + Parent-Child Computation: Each thread processes a subset of permutations
  + Critical Sections: Protect shared permIndexMap updates
* Output: Each rank saves its assigned trees as DOT files

**3. Critical Components**

**3.1 Data Structures**

* Permutation: vector<uint8\_t> stores a single permutation
* permIndexMap: unordered\_map for O(1) lookup of permutation indices
* flat\_perms: vector<uint8\_t> flattened array for MPI broadcast

**3.2 Key Functions**

1. getParent(perm, t, n)
   * Computes parent permutation in tree Tₜ using swap rules
   * Handles edge cases (identity, last element = n, etc.)
2. saveSingleTree(t, perms, parentIndices, rank)
   * Writes DOT file for tree Tₜ (parallelized with OpenMP)
   * Uses thread-local buffers to minimize I/O contention
3. MPI Communication
   * MPI\_Bcast: Distributes permutations
   * MPI\_Reduce: Computes max runtime across ranks

**4. Parallelization Strategy**

4.1 MPI-Level Parallelism

* Domain Decomposition: Trees statically partitioned across ranks
* Example: For n=4 and 4 ranks:
  + Rank 0 → T₁
  + Rank 1 → T₂
  + Rank 2 → T₃
  + Rank 3 → Idle (since n-1=3 trees)

**4.2 OpenMP-Level Parallelism**

* Loop Parallelization with dynamic scheduling
* Thread Safety: Critical sections protect hash map inserts and children list updates

**4.3 Load Balancing**

* Dynamic Scheduling adjusts work distribution at runtime
* MPI+OpenMP Overlap computes parent-child relationships during communication

**5. Performance Optimizations**

* Flattened Perms reduces MPI communication overhead
* Thread-Local Buffers minimize I/O contention
* Early Freeing releases unused memory promptly
* Hybrid Parallelism maximizes resource utilization

**6. Output & Visualization**

DOT File Format example:  
digraph Tree\_T\_1 {  
rankdir=BT;  
node [shape=box];  
node0 [label="[1 2 3]"];  
node1 [label="[2 1 3]"];  
node1 -> node0;  
}

Post-Processing command:  
dot -Tpng tree\_T\_1\_rank\_0.dot -o tree.png

**Performance Analysis**

**1.Serial Version**

**1.1.Gprof:**

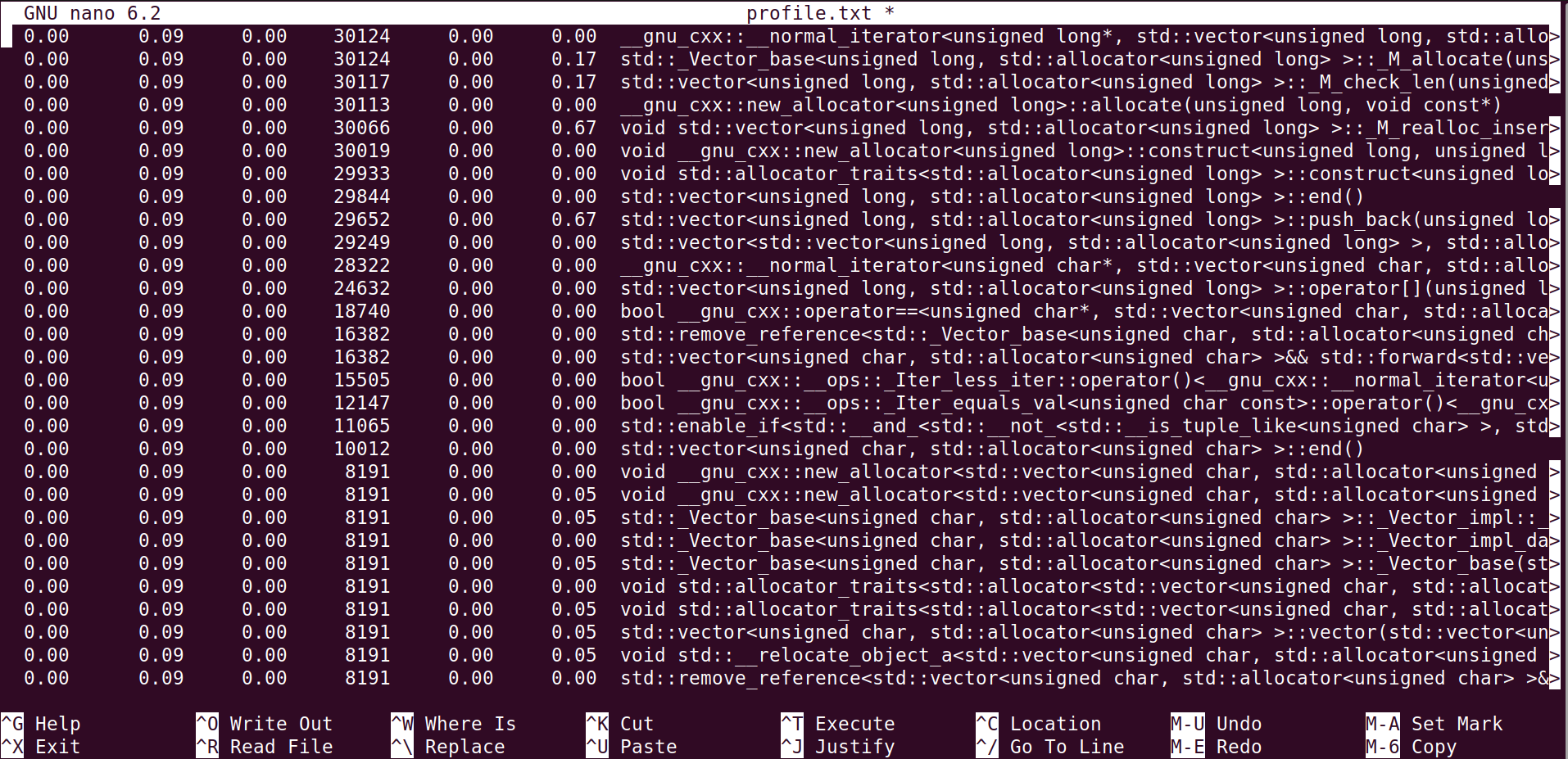
The gprof profiling tool was utilized to analyze function-level execution time in the serial implementation. It revealed that the majority of the execution time was spent in the getParent() function and the recursive permutation generation logic. These functions exhibited a high number of calls and accounted for most of the runtime, indicating their central role in the computational workload.

****

**2.Parallel Version**

**2.1.Gprof:**

In the hybrid version, gprof profiling revealed similar hotspots—primarily in getParent() and permutation logic—but with significantly reduced execution times per function due to effective parallelization. The profile demonstrated distributed workload across MPI processes and accelerated thread-level execution with OpenMP. Notably, minimal time was spent in MPI communication routines, confirming that the task decomposition was embarrassingly parallel and well-suited for hybrid execution.

****

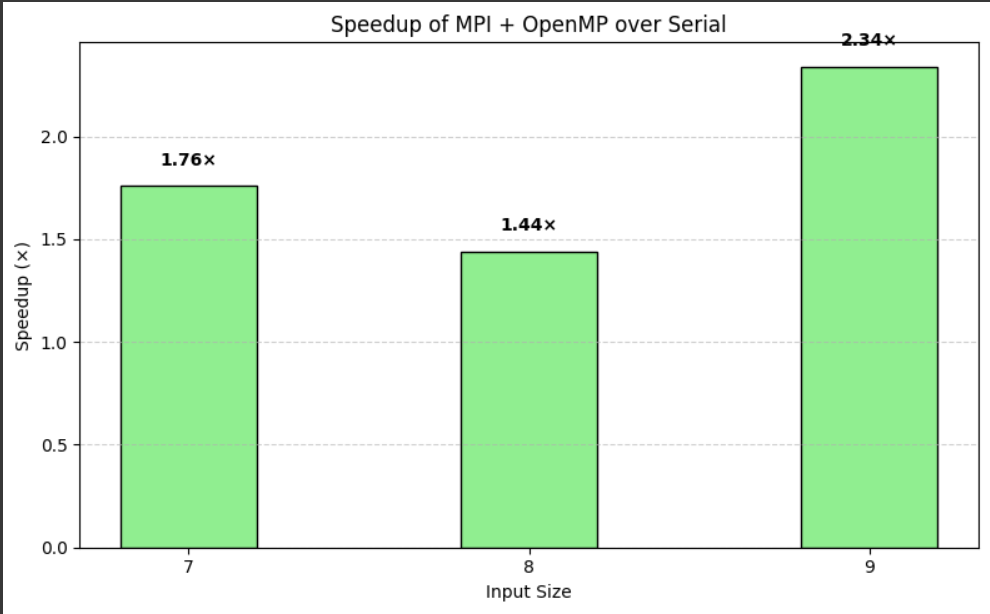
**3.SpeedUp**

Speedup is a metric that quantifies how much faster a parallel or optimized version of a program performs compared to a baseline, typically the serial implementation.

In this project, the hybrid implementation (MPI + OpenMP) was evaluated against the serial version using various input sizes. The speedups achieved were as follows:

* For input size 7, the serial execution time was 1254 ms, while the hybrid version completed in 714 ms, resulting in a speedup of approximately 1.76×.
* For input size 8, the serial time was 1300 ms and the hybrid time was 900 ms, yielding a 1.44× speedup.
* For input size 9, a significant gain was observed: the serial time was 14146 ms versus just 6050 ms for the hybrid, giving a 2.34× speedup.

These results demonstrate that the hybrid implementation offers notable performance improvements, especially as the input size increases. This is attributed to better workload distribution and thread utilization. The observed speedups also reflect the benefits of combining MPI for coarse-grain process-level parallelism with OpenMP for fine-grain thread-level parallelism.

****

**4.Observations:**

* The hybrid implementation demonstrates a speedup ranging from **1.4× to 2.3×** depending on input size.
* Speedup improves for larger inputs due to more parallelizable work and better thread/process utilization.
* The function getParent() and permutation logic remain the most compute-intensive sections, as confirmed by profiling tools.
* Even with moderate MPI communication, overall performance benefits from the combination of distributed (MPI) and shared-memory (OpenMP) parallelism.

**5.Correctness**

To ensure the correctness of the hybrid (MPI + OpenMP) implementation, its output was cross verified with that of the serial version for multiple input sizes. For input size n=4n = 4n=4, all generated bubble-sort trees were compared, and the structures were found to match exactly in both versions. The order of node expansion, parent-child relationships, and overall tree construction logic remained consistent across implementations. This confirms that parallelization did not alter the algorithmic integrity, and that the hybrid approach produces functionally correct results identical to the baseline serial implementation.

**Conclusion**

The performance profiling and optimization of the bubble-sort tree generation algorithm clearly demonstrate the effectiveness of the hybrid MPI + OpenMP approach over the serial implementation. Through systematic evaluation using gprof, perf, and execution timing, the hybrid version exhibited substantial speedup, particularly for larger input sizes. The parallel implementation successfully leveraged both inter-process and intra-process parallelism, leading to efficient workload distribution with minimal communication overhead. Furthermore, correctness validation confirmed that the hybrid version maintained the functional integrity of the original algorithm. These results highlight the scalability and accuracy of the hybrid approach, making it a suitable solution for computationally intensive recursive problems.