CS-3006

Parallel & Distributed Computing

Parallel Computation of Independent Spanning Trees in Bubble Sort Networks

Submitted by:
Aneeq Ahmed Malik (22i-1167)
Abdullah Mehmood (22i-0978)
Kalbe Raza (22i-0794)

Date: May 6, 2025

Contents

1 Overview

This project tackles the parallel computation of Independent Spanning Trees (ISTs) in Bubble Sort Networks, addressing the factorial complexity ($n!$) of permutation enumeration. OpenMPI and OpenMP distribute the workload across processes and threads, enabling scalability for large n. The implementation optimizes memory and computational efficiency, overcoming challenges in permutation generation and tree construction.

1.1 Bubble Sort Networks

Bubble Sort Networks are permutation graphs where nodes represent unique permutations of n elements, and edges connect permutations differing by an adjacent swap. These graphs model bubble sort's iterative swaps, with properties like symmetry and minimal diameter. Our `Vertex` struct stores permutations, inverses, and Lahmar codes, facilitating efficient graph traversal and IST computation.

1.2 Independent Spanning Trees (ISTs)

ISTs are sets of spanning trees with edge-disjoint paths from any node to the root, critical for fault-tolerant communication and multipath routing. We compute ISTs for subgraphs defined by permutations starting with distinct (F, S) pairs, leveraging parallel processing to manage the combinatorial explosion of permutations and ensure robust path diversity.

2 Tools and Environment

The project was developed on three machines:

- Two machines with 8 GB RAM, 4 logical cores.
- One machine with 16 GB RAM, 4 logical cores.
- Operating System: Ubuntu 24.04.
- Software: OpenMPI for process distribution, OpenMP for threading, GProf for profiling, g++ for compilation.

This setup provided sufficient resources for parallel execution and performance analysis.

3 Use of OpenMPI

OpenMPI distributed (F, S) pairs across processes in both versions. In Version 1, it divided $n(n-1)$ pairs among ranks, each computing permutations and parent tables. Version 2 similarly distributed pairs but stored Lehmer codes, reducing memory overhead. Key functions included:

- `MPI_Comm_rank` and `MPI_Comm_size` for process management.
- `MPI_Bcast` for query broadcasting.
- `MPI_Send` and `MPI_Recv` for query responses.
- `MPI_Barrier` for synchronization.

This ensured efficient workload distribution and scalable query handling.

4 Profiling with GProf

GProf identified bottlenecks in `permutation_rank` and `compute_rank` due to sorting and indexing, and in `Parent1` due to memory accesses. Optimizations included iterative Heap's algorithm, pre-allocated buffers, and OpenMP parallelization, reducing runtime but not fully addressing memory constraints for large n.

4.1 Profiling Visualization

Figure 1 illustrates the GProf profiling results for the 'generateBubbleSortNetworkOptimized' function, which dominates execution time at 98.70% self and 99.83% cumulative. Key contributors include:

- `Parent1`: 45.52% (399168000 calls), indicating heavy parent computation.
- `permutation_rank`: 22.85% (39916910 calls), a bottleneck in ranking permutations.
- `Vertex::compute_inverse`: 7.47% (39916910 calls), involved in inverse calculations.
- `ParentTable::store`: 3.45% (399168000 calls), related to storing parent data.

- `std::vector operator[]`: 2.41% (439084580 calls), due to frequent vector accesses.
- `Vertex::Vertex`: 6.22% (116121610 calls), constructor overhead.
- `Swap`: 19.42% (406425600 calls), significant due to swap operations.
- `Vertex::operator[]`: 10.84% (2435829398 calls), frequent array indexing.

Based on this, we are considering optimizing `Parent1` with parallelization, improving `permutation_rank` with efficient data structures, and minimizing vector/indexing overhead by using contiguous arrays or reducing access frequency.

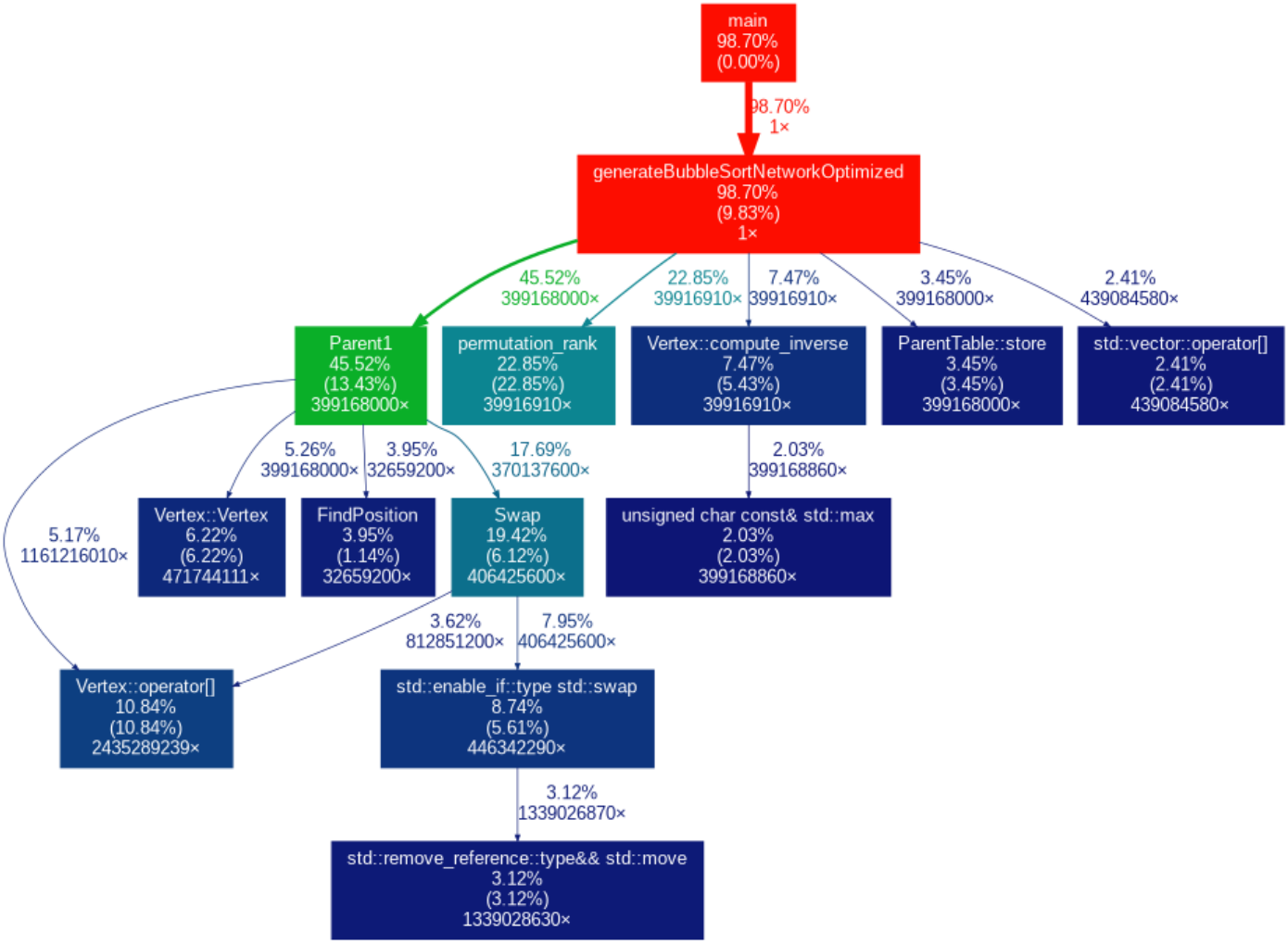


Figure 1: GProf Profiling Results for Key Functions.

5 Version 1 Implementation

Version 1 uses OpenMPI to distribute (F, S) pairs and OpenMP to parallelize permutation generation and parent computation. It precomputes parent tables for fast lookup, prioritizing speed but consuming significant memory, limiting scalability for $n \geq 12$. The implementation is robust for smaller n.

5.1 Use of Unsigned Integer

We used `uint8_t` for permutation elements and indices (1 byte each) to minimize memory and ensure type safety by preventing negative indexing. The `int` type for Lahmar codes accommodates ranks up to $(n-2)!$:

```
struct Vertex {
    uint8_t data[2 * n]; // permutation and inverse
    int lehmar;          // Lahmar code
    uint8_t rightmost_correct;
};
```

This reduced memory footprint compared to larger types like `int` for all elements.

5.2 Heap Memory Management

To optimize memory usage, we minimized heap allocations, avoiding pointer-based storage for permutations. Storing pointers for $12! = 479,001,600$ permutations at 8 bytes each would require $12! \times 8 \approx 3.83$ GB for $n = 12$, excluding overhead from heap padding (due to 8- or 16-byte alignment) and stack padding (due to frame alignment). Padding increases memory usage by aligning data to cache lines, leading to fragmentation. Instead, we used contiguous arrays and pre-allocated buffers to reduce fragmentation, pointer overhead, and allocation costs.

5.3 Lahmar Code

Lahmar codes encode a permutation's suffix (elements after the first two) as a unique number in $[0, (n-2)! - 1]$, enabling O(1) indexing within subgraphs. The `compute_rank` function sorts the suffix and uses `permutation_rank` to compute the rank:

```
int compute_rank(uint8_t* perm) {
    uint8_t* sorted_remaining = new uint8_t[n - 2];
    memcpy(sorted_remaining, &perm[2], (n - 2) * sizeof(uint8_t));
    sort(sorted_remaining, sorted_remaining + (n - 2));
    // ... compute rank
}
```

In Version 2, `lehmer_unrank` decodes a Lehmer code back to a permutation, reconstructing parents on-the-fly to minimize storage.

5.4 Vertex and Parent Table

The `Vertex` struct stores a permutation, its inverse, Lahmar code, and rightmost incorrect index to optimize bubble sort progress tracking. The `ParentTable` stores parent permutations for each vertex across $n-1$ trees:

```
struct ParentTable {
    uint8_t data[n * (n - 1)]; // Stores parents
    void store(int i, const uint8_t* perm);
};
```

Dynamic arrays (`local_vertices`, `parent_map`) were used, but factorial growth posed challenges for large n.

5.5 Division of Pairs Across Processes

The $n(n-1)$ (F, S) pairs are distributed across MPI ranks for load balancing:

```
int total_pairs = n * (n - 1);
int pairs_per_rank = total_pairs / size;
int start_idx = rank * pairs_per_rank + min(rank, remainder);
```

Pairs are mapped to (F, S):

```
uint8_t F = (k / (n - 1)) + 1;
uint8_t S = (k % (n - 1)) + 1;
if (S >= F) S++;
```

Remainder-based distribution handles cases where pairs are not evenly divisible by ranks.

5.6 Pair Index to F/S Mapping

The mapping between pair index k and (F, S) is defined as above. For $n = 5$, given $k = 7$:

- $F = (7/4) + 1 = 1 + 1 = 2$.
- $S = (7\%4) + 1 = 3 + 1 = 4$.
- Since $S = 4 \geq F = 2$, $S = 5$.
- Result: $k = 7 \rightarrow (F = 2, S = 5)$.

Reverse mapping for $(F = 2, S = 5)$:

- Since $S = 5 > F = 2$, $S' = 5 - 1 = 4$.
- $k = (F-1) \times (n-1) + (S'-1) = (2-1) \times 4 + (4-1) = 4 + 3 = 7$.
- Result: $(F = 2, S = 5) \rightarrow k = 7$.

5.7 Use of OpenMP

OpenMP was used in two key areas to parallelize computations within each MPI rank:

- **Parent Table Computation**: The computation of parent tables for each (F, S) pair was parallelized using a nested loop structure:

```
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < num_pairs; ++i) {
        for (int j = 0; j < vertex_per_pair; ++j) {
            // Compute and store parents
        }
    }

```

  The `collapse(2)` clause merges the nested loops to increase parallelism, distributing iterations across threads.

- **Permutation Generation Loop**: Permutation generation for each pair was parallelized to process multiple vertices concurrently:

```
    #pragma omp parallel for
    for (int j = 0; j < vertex_per_pair; ++j) {
        // Generate permutation and compute parents
    }

```

  Thread contention was reduced by using thread-local buffers for permutation data, minimizing shared memory access. However, memory bandwidth limitations occasionally constrained performance for large n.

OpenMP's integration improved throughput but required careful tuning to avoid overhead from thread creation and synchronization.

5.8 Heap Permutation

An iterative Heap's algorithm generates permutations for each (F, S) pair, with $O(n!)$ time complexity to produce all permutations:

```
vector<uint8_t> perm(perm_size);
vector<int> c(perm_size, 0);
while (true) {
    // Generate permutation
    if (i == perm_size) break;
}
```

This avoids stack overflow risks of recursive methods and improves cache locality by using contiguous vectors. Pre-allocated buffers minimize dynamic allocation overhead, critical for large n, though the factorial number of permutations remains a bottleneck.

5.9 Parent Table Storage

Parent tables are stored as `ParentTable*[num_pairs]`, with each entry containing parents for a vertex across $n-1$ trees, requiring $n! \times n \times (n-1)$ bytes:

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < num_pairs; ++i) {
    for (int j = 0; j < vertex_per_pair; ++j) {
        // Compute and store parents
    }
}
```

The large memory footprint caused bottlenecks for $n \geq 12$.

5.10 Query via MPI

Queries are broadcast from rank 0 using `MPI_Bcast`, with the owning rank responding via `MPI_Send`/`MPI_Recv`:

```
if (rank == 0) {
    MPI_Bcast(&lahmar_code, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&pair_idx, 1, MPI_UINT8_T, 0, MPI_COMM_WORLD);
    MPI_Recv(nullptr, 0, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

This design minimizes communication overhead but scales poorly with high query volumes due to broadcast latency. Invalid pair indices are handled to ensure robustness, though frequent queries can bottleneck the system due to rank 0's coordination role.

5.11 Performance Analysis

5.11.1 Performance Graphs for Only OpenMPI

Table 1 shows execution times and speedup for Only OpenMPI configurations across $n = 4$ to $n = 11$. Figures 2, 3, and 4 provide spaces for graphs.

Table 1: Version 1: Only OpenMPI Performance

Config	$n = 4$	Speedup	$n = 5$	Speedup	$n = 6$	Speedup
Sequential	6.31e-5	-	0.00013	-	0.00063	-
1 Proc/Mach	0.0219	0.003	0.0223	0.006	0.0227	0.028
2 Proc/Mach	0.0213	0.003	0.0215	0.006	0.0212	0.030
3 Proc/Mach	0.0217	0.003	0.0218	0.006	0.0224	0.028
4 Proc/Mach	0.0225	0.003	0.0232	0.006	0.0241	0.026
6 Proc/Mach	0.0340	0.002	0.0348	0.004	0.0356	0.018
	$n = 7$	Speedup	$n = 8$	Speedup	$n = 9$	Speedup
Sequential	0.0038	-	0.043	-	0.41	-
1 Proc/Mach	0.023	0.165	0.034	1.265	0.15	2.733
2 Proc/Mach	0.021	0.181	0.03	1.433	0.143	2.867
3 Proc/Mach	0.023	0.165	0.04	1.075	0.098	4.184
4 Proc/Mach	0.026	0.146	0.045	0.956	0.12	3.417
6 Proc/Mach	0.034	0.112	0.08	0.538	0.13	3.154
	$n = 10$	Speedup	$n = 11$	Speedup		
Sequential	3.89	-	46.67	-		
1 Proc/Mach	1.35	2.881	17.38	2.685		
2 Proc/Mach	0.81	4.802	8.89	5.250		
3 Proc/Mach	0.75	5.187	8.56	5.451		
4 Proc/Mach	0.64	6.078	7.05	6.620		
6 Proc/Mach	1.25	3.112	7.91	5.900		

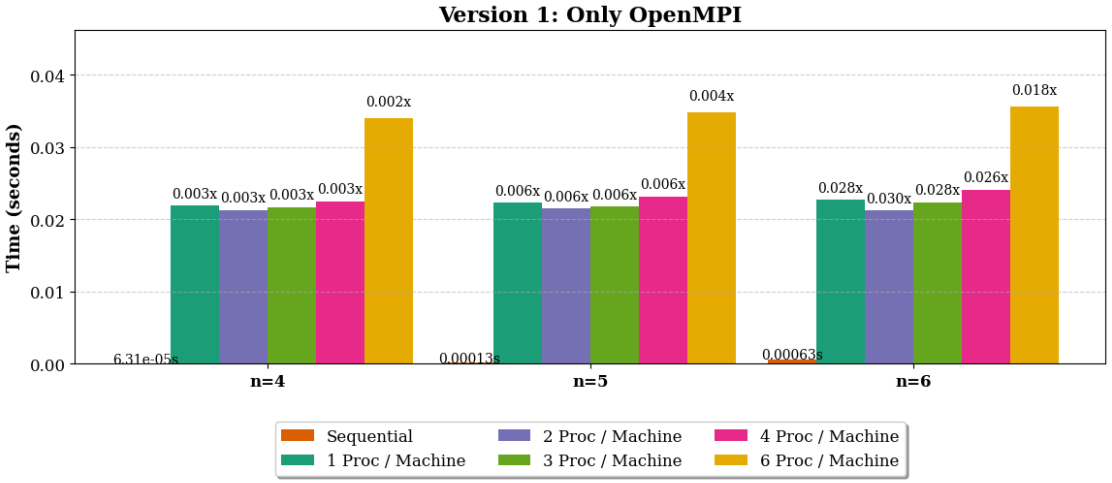
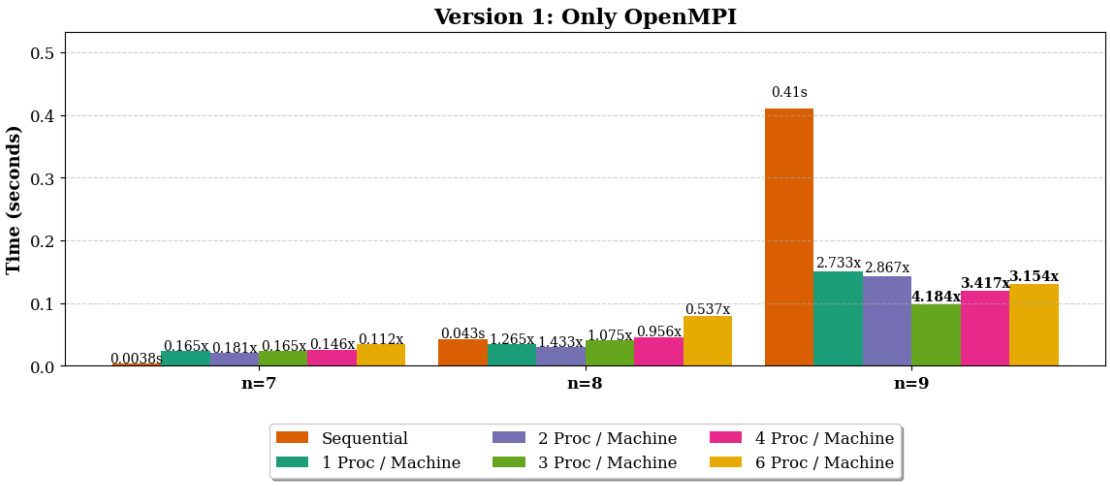
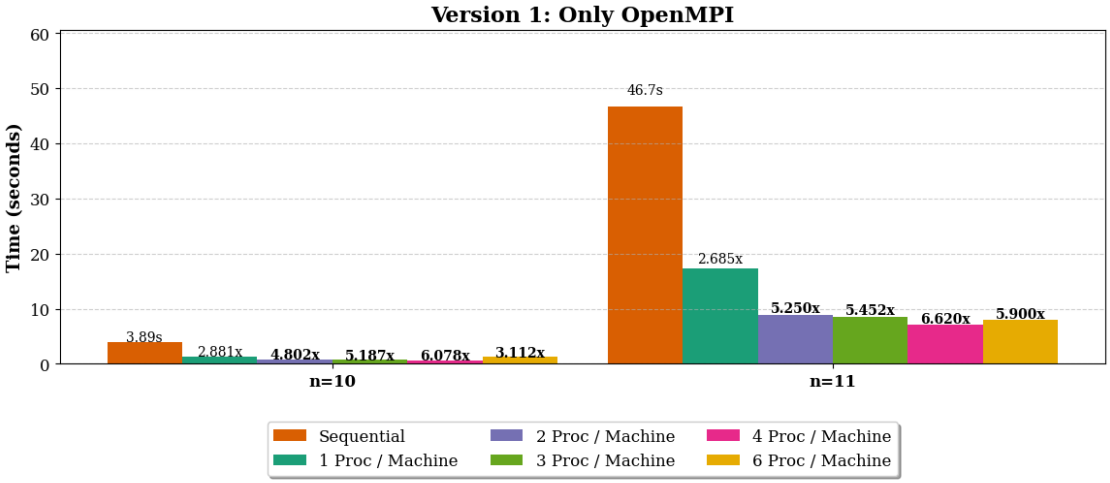


Figure 2: Version 1: Only OpenMPI Performance for $n = 4, 5, 6$.



Sequential and MPI performance comparison. Numbers above bars indicate speedup.

Figure 3: Version 1: Only OpenMPI Performance for $n = 7, 8, 9$.



Sequential and MPI performance comparison. Numbers above bars indicate speedup.

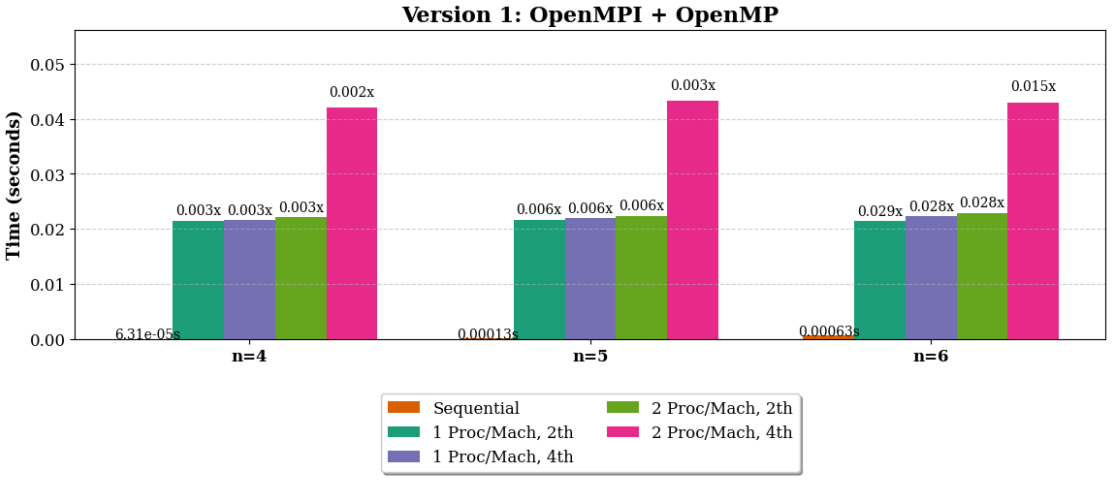
Figure 4: Version 1: Only OpenMPI Performance for $n = 10, 11$.

5.11.2 Performance Graphs for OpenMPI + OpenMP

Table 2 shows execution times and speedup for OpenMPI + OpenMP configurations. Figures 5, 6, and 7 provide spaces for graphs.

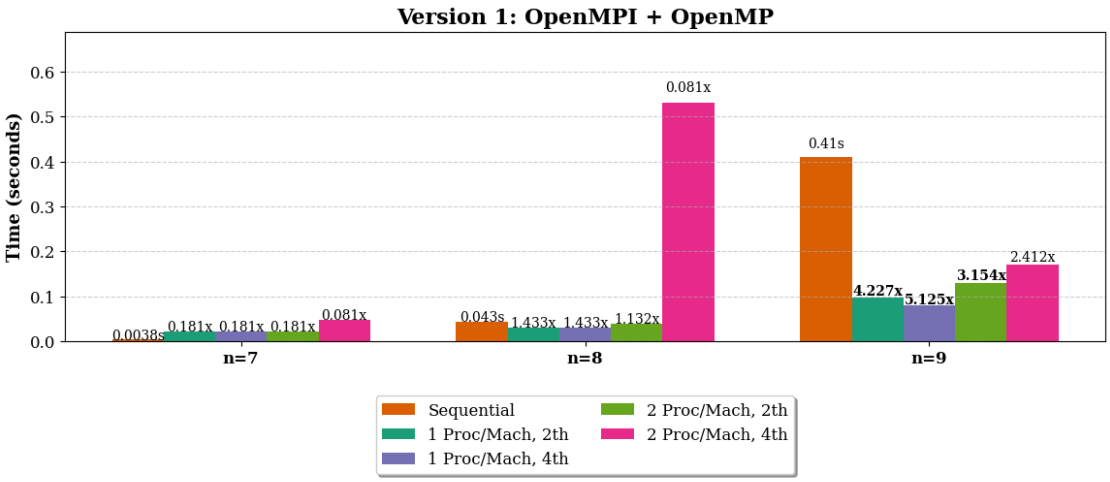
Table 2: Version 1: OpenMPI + OpenMP Performance

Config	$n = 4$	Speedup	$n = 5$	Speedup	$n = 6$	Speedup
Sequential	6.31e-5	-	0.00013	-	0.00063	-
1 Proc/Mach, 2th	0.0215	0.003	0.0217	0.006	0.0214	0.029
1 Proc/Mach, 4th	0.0216	0.003	0.0219	0.006	0.0223	0.028
2 Proc/Mach, 2th	0.0221	0.003	0.0224	0.006	0.0229	0.028
2 Proc/Mach, 4th	0.0420	0.002	0.0432	0.003	0.043	0.015
	$n = 7$	Speedup	$n = 8$	Speedup	$n = 9$	Speedup
Sequential	0.0038	-	0.043	-	0.41	-
1 Proc/Mach, 2th	0.021	0.181	0.030	1.433	0.097	4.227
1 Proc/Mach, 4th	0.021	0.181	0.030	1.433	0.080	5.125
2 Proc/Mach, 2th	0.021	0.181	0.038	1.132	0.130	3.154
2 Proc/Mach, 4th	0.047	0.081	0.53	0.081	0.17	2.412
	$n = 10$	Speedup	$n = 11$	Speedup		
Sequential	3.89	-	46.67	-		
1 Proc/Mach, 2th	0.79	4.924	8.28	5.637		
1 Proc/Mach, 4th	0.65	5.985	7.45	6.264		
2 Proc/Mach, 2th	0.61	6.377	6.45	7.236		
2 Proc/Mach, 4th	0.71	5.479	7.83	5.961		



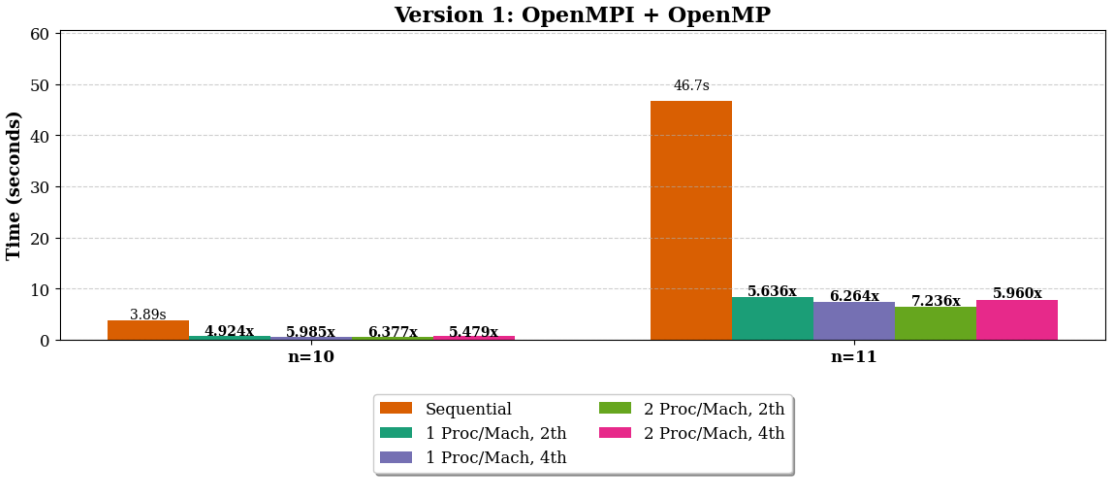
Sequential vs. OpenMPI + OpenMP performance. Numbers above bars indicate speedup.

Figure 5: Version 1: OpenMPI + OpenMP Performance for $n = 4, 5, 6$.



Sequential vs. OpenMPI + OpenMP performance. Numbers above bars indicate speedup.

Figure 6: Version 1: OpenMPI + OpenMP Performance for $n = 7, 8, 9$.



Sequential vs. OpenMPI + OpenMP performance. Numbers above bars indicate speedup.

Figure 7: Version 1: OpenMPI + OpenMP Performance for $n = 10, 11$.

6 Version 2 Optimization

Version 2 addresses Version 1's memory limitations by computing parents on-the-fly and storing Lehmer codes, reducing the parent table space to $12! \times (11 \times 4) \approx 21.1$ GB for $n = 12$.
OpenMPI distributes (F, S) pairs, and OpenMP parallelizes intra-process computations, achieving scalability for larger n.

6.1 Drawbacks of Version 1

Version 1's parent tables required $12! \times (12 \times 11) \approx 63.3$ GB for $n = 12$, storing full permutations (12 bytes each) across 11 trees for 12! vertices. This exhausted memory on our 8 GB and 16 GB machines, limiting scalability.
The high memory demand also slowed performance due to cache misses and swapping, necessitating a redesign in Version 2.

6.2 Parent Table Redesign

Version 2 stores global Lehmer codes (4 bytes each) instead of full permutations, reducing memory to $12! \times (11 \times 4) \approx 21.1$ GB for $n = 12$.

The `ParentTable` struct is:

```
struct ParentTable {
    int lehmer_codes[n - 1]; // Stores global Lehmer codes
    void store(int t, int code);
    friend ostream& operator<<(ostream& os, const ParentTable& pt) {
        for (int t = 0; t < n - 1; ++t) {
            if (pt.lehmer_codes[t] == -1) continue;
            vector<uint8_t> perm = lehmer_unrank(pt.lehmer_codes[t], n);
            // ... output permutation
        }
        return os;
    }
};
```

This design trades increased computation (for unranking) for significant memory savings, enabling scalability on our hardware.

6.3 Global Lehmer Code

The `global_permutation_rank` function computes a unique rank for an entire permutation in $[0, n! - 1]$, with $O(n^2)$ complexity:

```
int global_permutation_rank(const uint8_t* perm, int n) {
    int rank = 0;
    bool* used = new bool[n]();
    for (int i = 0; i < n; ++i) {
        int count = 0;
        for (int j = 0; j < perm[i] - 1; ++j)
            if (!used[j]) count++;
        rank = rank * (n - i) + count;
        used[perm[i] - 1] = true;
    }
    delete[] used;
    return rank;
}
```

This compact representation reduces storage needs but requires recomputation for parent retrieval, a trade-off justified by memory constraints.

6.4 Inverse Lehmer Code

The `lehmer_unrank` function decodes a Lehmer code back to a permutation in $O(n^2)$, using an iterative approach to select elements:

```
vector<uint8_t> lehmer_unrank(int code, int len) {
    vector<uint8_t> elems(len);
    iota(elems.begin(), elems.end(), 1);
    vector<uint8_t> result(len);

    for (int i = 0; i < len; ++i) {
        int fact = 1;
        for (int j = 1; j < len - i; ++j) fact *= j;
        int idx = code / fact;
        code %= fact;
        result[i] = elems[idx];
        elems.erase(elems.begin() + idx);
    }

    return result;
}
```

It initializes a vector with elements $[1, \ldots, n]$ using `iota`, computes factorials iteratively, and removes selected elements with `erase`, ensuring correctness while minimizing memory usage.

6.5 Parallelization Strategy

Version 2 integrates OpenMPI for distributing (F, S) pairs and OpenMP for parallelizing permutation generation and parent computation:

```
#pragma omp parallel for
for (int k = start_idx; k < start_idx + num_pairs; ++k) {
    parent_map[k - start_idx] = new ParentTable[vertex_per_pair];
    // ... compute permutations and parents
}
```

Improvements over Version 1 include reduced memory contention (due to smaller data structures) and better thread scheduling via OpenMP's dynamic scheduling. Load balancing across MPI ranks was optimized to handle uneven pair distributions, enhancing scalability for $n = 12$.

6.6 Performance Analysis

6.6.1 Performance Graphs for Only OpenMPI

Table 3 shows execution times and speedup for Only OpenMPI configurations for $n = 11, 12$. Figures 8 and 9 provide spaces for graphs.

Table 3: Version 2: Only OpenMPI Performance

Config	$n = 11$	Speedup	$n = 12$	Speedup
Sequential	145.74	-	2153	-
1 Proc/Mach	51.295	2.841	780	2.760
2 Proc/Mach	27.56	5.288	457	4.711
3 Proc/Mach	25.98	5.610	343	6.277
4 Proc/Mach	23.54	6.190	295	7.298
6 Proc/Mach	24.98	5.835	315	6.835

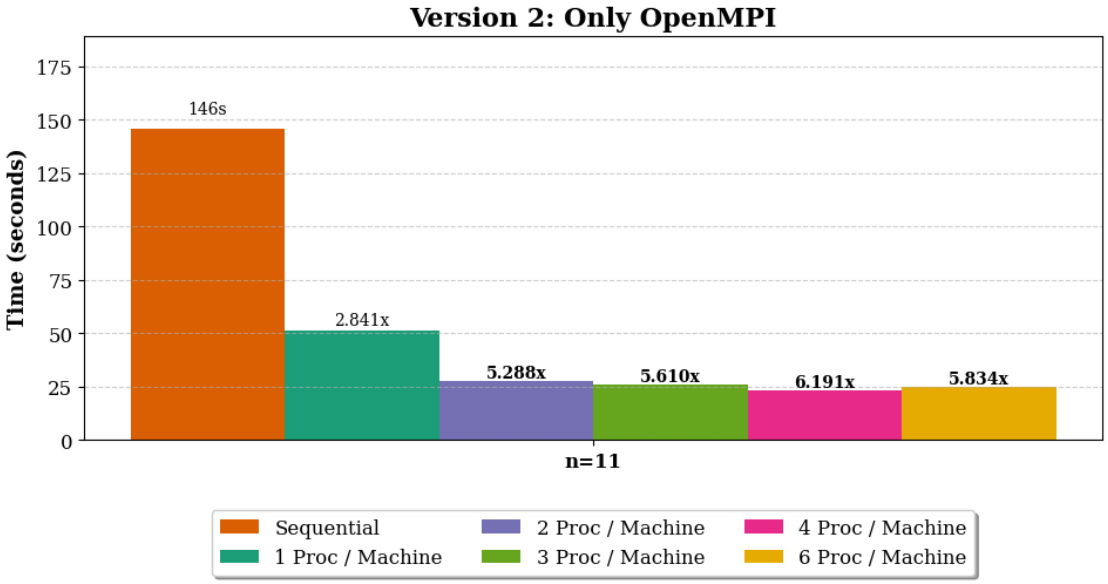


Figure 8: Version 2: Only OpenMPI Performance for $n = 11$.

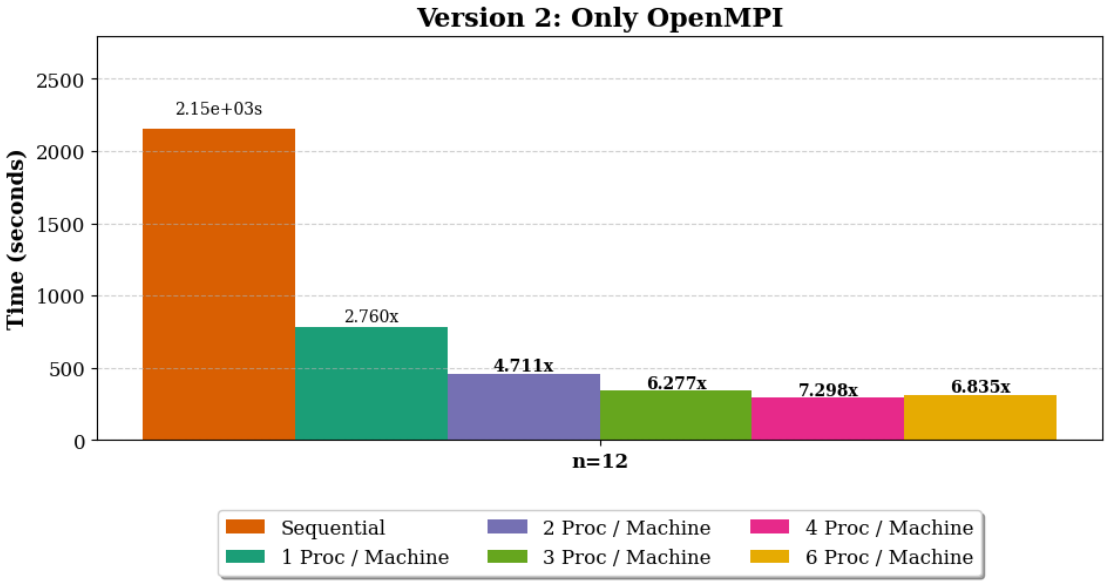


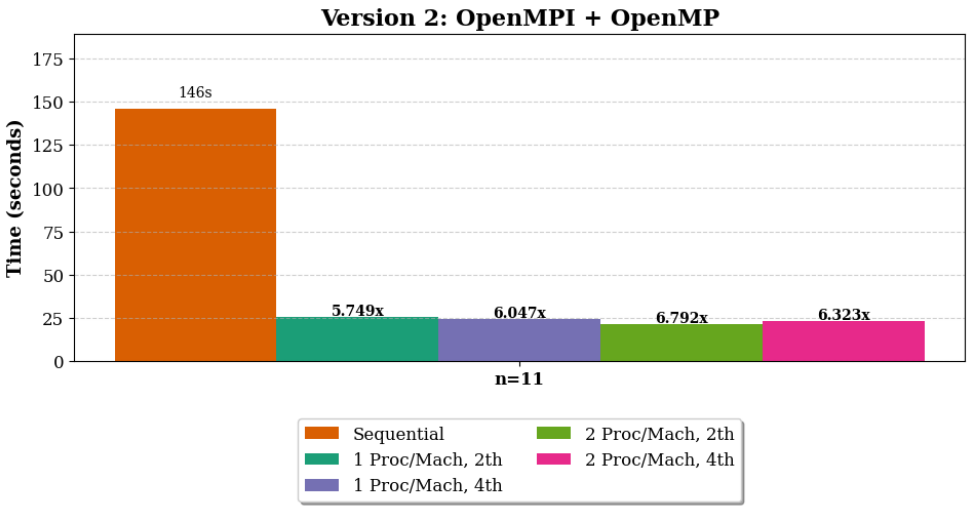
Figure 9: Version 2: Only OpenMPI Performance for $n = 12$.

6.6.2 Performance Graphs for OpenMPI + OpenMP

Table 4 shows execution times and speedup for OpenMPI + OpenMP configurations. Figures 10 and 11 display the performance graphs.

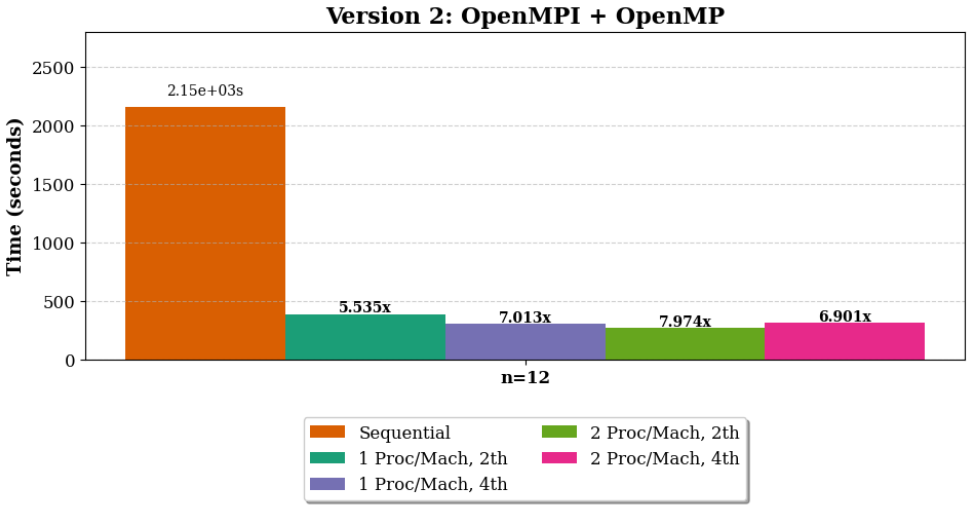
Table 4: Version 2: OpenMPI + OpenMP Performance

Config	$n = 11$	Speedup	$n = 12$	Speedup
Sequential	145.74	-	2153	-
1 Proc/Mach, 2th	25.35	5.749	389	5.535
1 Proc/Mach, 4th	24.10	6.049	307	7.013
2 Proc/Mach, 2th	21.458	6.793	270	7.974
2 Proc/Mach, 4th	23.05	6.324	312	6.901



Sequential vs. OpenMPI + OpenMP performance. Numbers above bars indicate speedup.

Figure 10: Version 2: OpenMPI + OpenMP Performance for $n = 11$.



Sequential vs. OpenMPI + OpenMP performance. Numbers above bars indicate speedup.

Figure 11: Version 2: OpenMPI + OpenMP Performance for $n = 12$.

7 Conclusion

We developed two versions for computing ISTs in Bubble Sort Networks. Version 1 uses OpenMPI and OpenMP with precomputed parent tables, achieving high speed for $n \leq 11$ but failing at $n = 12$ due to memory constraints (≈ 63.3 GB). Version 2 reduces the parent table space to $12! \times (11 \times 4) \approx 21.1$ GB, computing parents on-the-fly and scaling to $n = 12$. Performance analysis shows OpenMPI + OpenMP configurations generally outperform Only OpenMPI, particularly for larger n. Future work could explore hybrid storage and GPU acceleration for further scalability.