Analysis of Parallel Bubble-Sort Spanning Tree Construction

Muhammad Salman Javed

Abstract

This report analyzes a C++ implementation of a parallel algorithm for constructing independent spanning trees (ISTs) in bubble-sort networks, based on the work by Kao et al. (2021). The code generates permutations, computes parent relationships for $n-1$ ISTs, and leverages OpenMP for parallelization. We examine the algorithm's functionality, correctness, and performance, using timing data for serial and parallel executions across $n = 4, 6, 8, 10$. The parallel implementation shows significant speedups for larger n, with a notable reduction from 94.757 seconds to 34.548 seconds for $n = 10$. Recommendations for further optimization are provided.

1 Introduction

Bubble-sort networks (B_n) are Cayley graphs where vertices represent permutations of $\{1, 2, ..., n\}$, and edges connect permutations differing by adjacent swaps. These networks are used in parallel computing for sorting and broadcasting. Independent spanning trees (ISTs) are $n-1$ trees rooted at the identity permutation (e.g., 123 for $n = 3$), where paths from any vertex to the root in different trees are vertex-disjoint except at endpoints. Constructing ISTs enhances fault tolerance and secure message distribution.

The analyzed code implements a parallel algorithm from Kao et al. (2021), which constructs $n-1$ ISTs in B_n. It generates all permutations, computes parent relationships for each permutation in each tree using OpenMP, and allows user queries for specific permutations. This report details the code's functionality, evaluates its performance, and suggests improvements.

2 Algorithm Overview

The algorithm constructs $n-1$ ISTs in B_n, rooted at the identity permutation. Key components include:

- **Permutation Generation**: Generates all $n!$ permutations of $\{1, 2, ..., n\}$ recursively.
- **Parent Computation**: For each permutation v and tree $t \in \{1, 2, ..., n-1\}$, computes the parent $p(v)$ using:

  - `is_root(v)`: Checks if v is the identity permutation.
  - `compute_inverse(v)`: Computes the inverse permutation for efficient position lookup.
  - `compute_r(v)`: Finds the rightmost position where the element is not in its correct place.
  - `find_parent(v, t)`: Determines the parent by swapping elements based on conditions involving t, v_n, and v_{n-1}.

- **Parallelization**: Uses OpenMP to parallelize parent computation across permutations.

The parent computation follows rules to ensure the resulting trees are independent, leveraging precomputed inverses and positions for efficiency.

3 Code Analysis

The C++ code is structured as follows:

3.1 Key Functions

- `generate_permutations`: Recursively generates all permutations, storing them in a vector.
- `find_parent`: Computes the parent of a permutation for tree t, handling special cases (e.g., root, $v_n = n$).
- `swap_elements`: Swaps elements at positions i and $i + 1$, used in parent computation.

3.2 Parallel Implementation

The loop computing parents for all permutations is parallelized using OpenMP:

```
#pragma omp parallel for
for (size_t i = 0; i < permutations.size(); i++) {
    const auto& perm = permutations[i];
    vector<vector<int>> parents(n - 1, vector<int>(n));
    for (int t = 1; t <= n - 1; t++) {
        find_parent(parents[t - 1], perm, t);
    }
    #pragma omp critical
    {
        parent_map[perm] = parents;
    }
}
```

A critical section ensures thread-safe insertion into the shared `parent_map`.

3.3 Correctness

The code correctly implements the algorithm for $n-1$ ISTs, producing one parent per tree t. However, the `find_parent` logic has complex conditionals that may need validation against a reference table to ensure the resulting trees are vertex-disjoint.

4 Performance Evaluation

The code measures execution time for both serial and parallel implementations, focusing on the time to generate permutations and compute parents. The following table summarizes the results:

Table 1: Serial vs. Parallel Execution Times for Permutation Parent Computation

Implementation	n	Serial Time (s)	Parallel Time (s)	Speedup
Serial	4	0.000 32		
	6	0.005 27		
	8	0.482 99		
	10	94.757		
Parallel	4		0.002 60	0.123
	6		0.005 88	0.896
	8		0.360 60	1.339
	10		34.548	2.743

4.1 Analysis

- **Small** n: For $n = 4$, the parallel version is slower (0.00260s vs. 0.00032s) due to OpenMP overhead (thread creation, synchronization).
- **Larger** n: For $n = 10$, the parallel version achieves a 2.74x speedup (94.757s to 34.548s), demonstrating scalability as the number of permutations ($10! = 3,628,800$) increases.
- **Speedup Trends**: Speedup improves with n, from 0.123x ($n = 4$) to 2.743x ($n = 10$), reflecting better utilization of parallel resources for larger workloads.

5 Recommendations

- **Parallel Permutation Generation**: The `generate_permutations` function is sequential. Implementing a parallel permutation generation algorithm (e.g., using OpenMP tasks) could further reduce total execution time.
- **Reduce Synchronization Overhead**: The critical section for `parent_map` insertion limits scalability. Using thread-local maps and merging results could minimize contention.

- **Memory Optimization**: Storing all $n!$ permutations is memory-intensive. Generating permutations on-the-fly or in chunks could improve scalability.
- **Dynamic Scheduling**: Adding `schedule(dynamic)` to the OpenMP directive may improve load balancing if parent computation times vary.

6 Conclusion

The analyzed code effectively constructs $n-1$ ISTs in bubble-sort networks, leveraging OpenMP for parallelization. It correctly computes parents for all permutations, with the parallel version showing significant speedups for larger n. However, overhead for small n and synchronization bottlenecks suggest areas for improvement. Future work could focus on parallelizing permutation generation and optimizing memory usage to enhance performance further.

References

- Kao, S.-S., et al. (2021). A parallel algorithm for constructing multiple independent spanning trees in bubble-sort networks. *Journal of Parallel and Distributed Computing*.