PERFORMANCE ANALYSIS REPORT

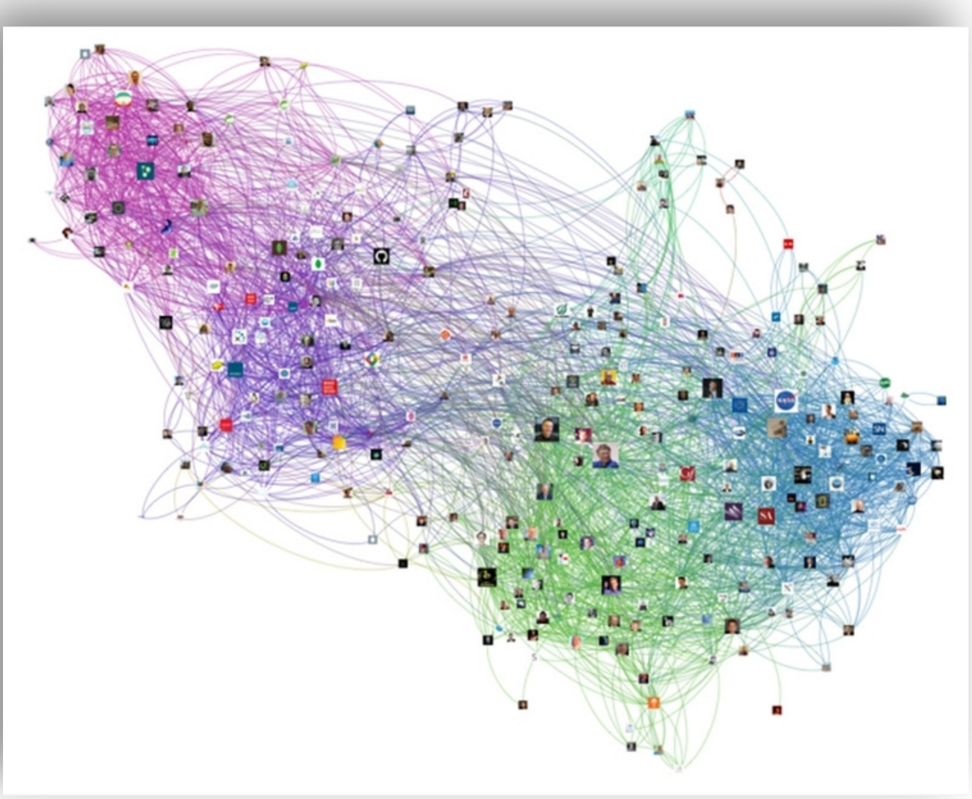
PDC

Abdullah Saleem 22i-0882

Siddique Ahmad Ryan 22i-0781

Zubair Adnan 22i-0789

CS-B



Introduction:

This report analyzes the performance of three versions of a C++ program designed to generate independent spanning trees
of permutations using different computational strategies in a Bubble Sort Network

1. ***Serial Version (No parallelism)***
2. ***OpenMP Version (Intra-node parallelism)***
3. ***Hybrid Version (MPI + OpenMP)***

The primary focus is on performance comparison and the impact of parallelization on computational
speed.

Code Overview:

This project constructs (n−1) edge-disjoint spanning trees over the permutation graph of Sn (set of all n! permutations of {1, 2, ..., n}), using parallel processing with MPI and OpenMP, and METIS for graph partitioning. Here's a breakdown of the major steps:

1. Permutation Representation:

- indexToPerm and permToIndex convert between permutation vectors and their corresponding lexicographic indices for efficient representation and computation.

2. Graph Construction:

- Each permutation is a node.
- An edge is added between permutations that differ by a single adjacent swap.
- This structure forms the adjacency list used for graph partitioning.

3. Graph Partitioning with METIS:

- The large permutation graph is partitioned using METIS_PartGraphKway to divide nodes among available MPI processes.
- This ensures load balancing during parallel computation of spanning trees.

4. Spanning Tree Generation:

- For each t from 1 to n−1, a separate spanning tree is constructed using the Parent1 algorithm, which defines parent-child relationships based on specific swap logic from academic literature.
- Each MPI process computes its local portion of edges using OpenMP for thread-level parallelism.

5. Result Gathering and Output:

- Edge lists from all processes are gathered at the root MPI process.
- The final spanning trees are written in Graphviz .dot format, one file per tree (spanning_tree_t.dot), which can be visualized using tools like graphviz.

6. Parallelism:

- MPI is used for process-level parallelism across machines or cores.

- OpenMP is used for thread-level parallelism inside each process.

1. Serial Version

The serial version processes all permutations sequentially. It is used as a baseline for comparison.

The primary bottleneck is the lack of concurrency, causing it to scale poorly with higher values of `n`
due to factorial growth in permutations.

Key Observations:

- Highest execution time.

- CPU usage limited to a single core.

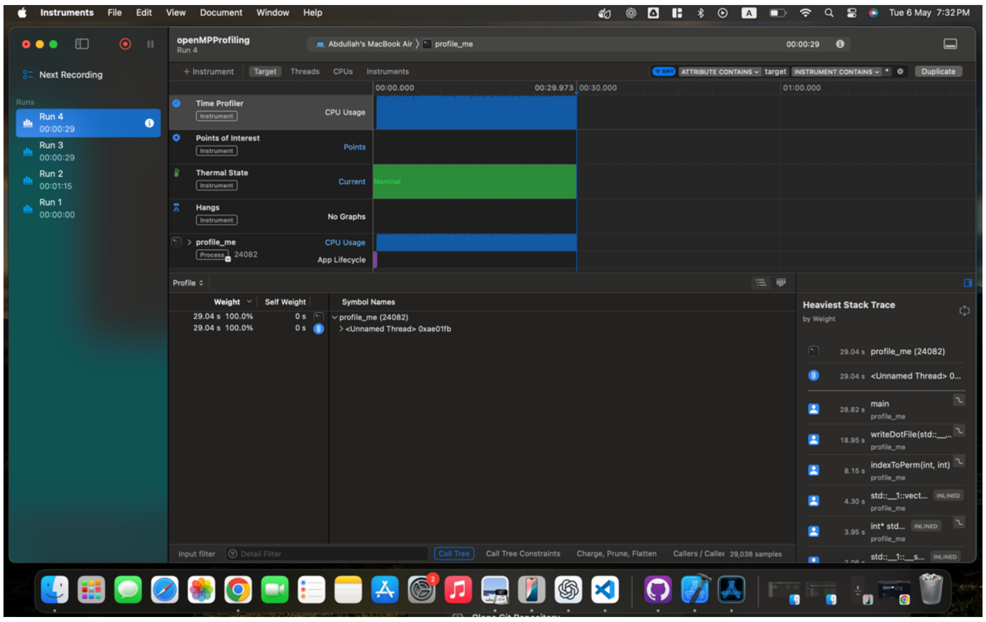
- Best for debugging and correctness validation.

Performance Analysis of Serial version

As we can clearly see, the serial version of the program serves as the baseline for evaluating the impact of parallelization techniques. In this implementation, the entire workload is executed sequentially on a single core, without taking advantage of any parallel hardware capabilities. As a result, the execution time is significantly higher compared to its parallel counterparts.

For instance, processing a moderately large n value let’s say n=12 can take over **6 minutes**, highlighting the limitations of sequential computation in high-performance scenarios. Although it is the simplest to implement and debug, it becomes increasingly inefficient as the problem size grows, making it unsuitable for real-world applications where time efficiency is critical.

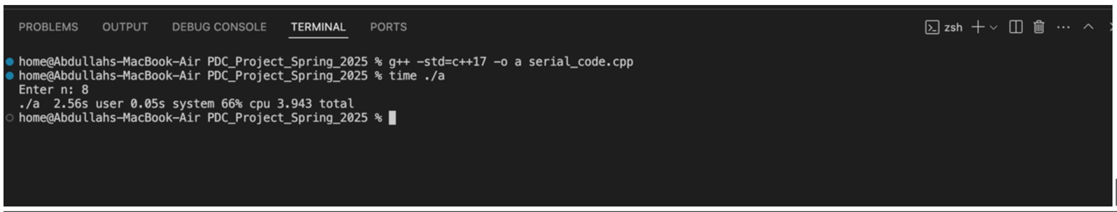
Profiling done using Instruments Time profiler :



Weight		Self Weight	Symbol Names
29.04 s	100.0%	0 s	profile_me (24082)
29.04 s	100.0%	0 s	<Unnamed Thread> 0xae01fb

Heaviest Stack Trace
by Weight

29.04 s	profile_me (24082)
29.04 s	<Unnamed Thread> 0...
28.82 s	main profile_me
18.95 s	writeDotFile(std::__... profile_me
8.15 s	indexToPerm(int, int) profile_me
4.30 s	std::__1::vect... INLINED profile_me
3.95 s	int* std... INLINED profile_me
3.06 s	std::__1::__s... INLINED



```
home@Abdullahs-MacBook-Air PDC_Project_Spring_2025 % g++ -std=c++17 -o a serial_code.cpp
home@Abdullahs-MacBook-Air PDC_Project_Spring_2025 % time ./a
Enter n: 8
./a  2.56s user 0.05s system 66% cpu 3.943 total
home@Abdullahs-MacBook-Air PDC_Project_Spring_2025 %
```

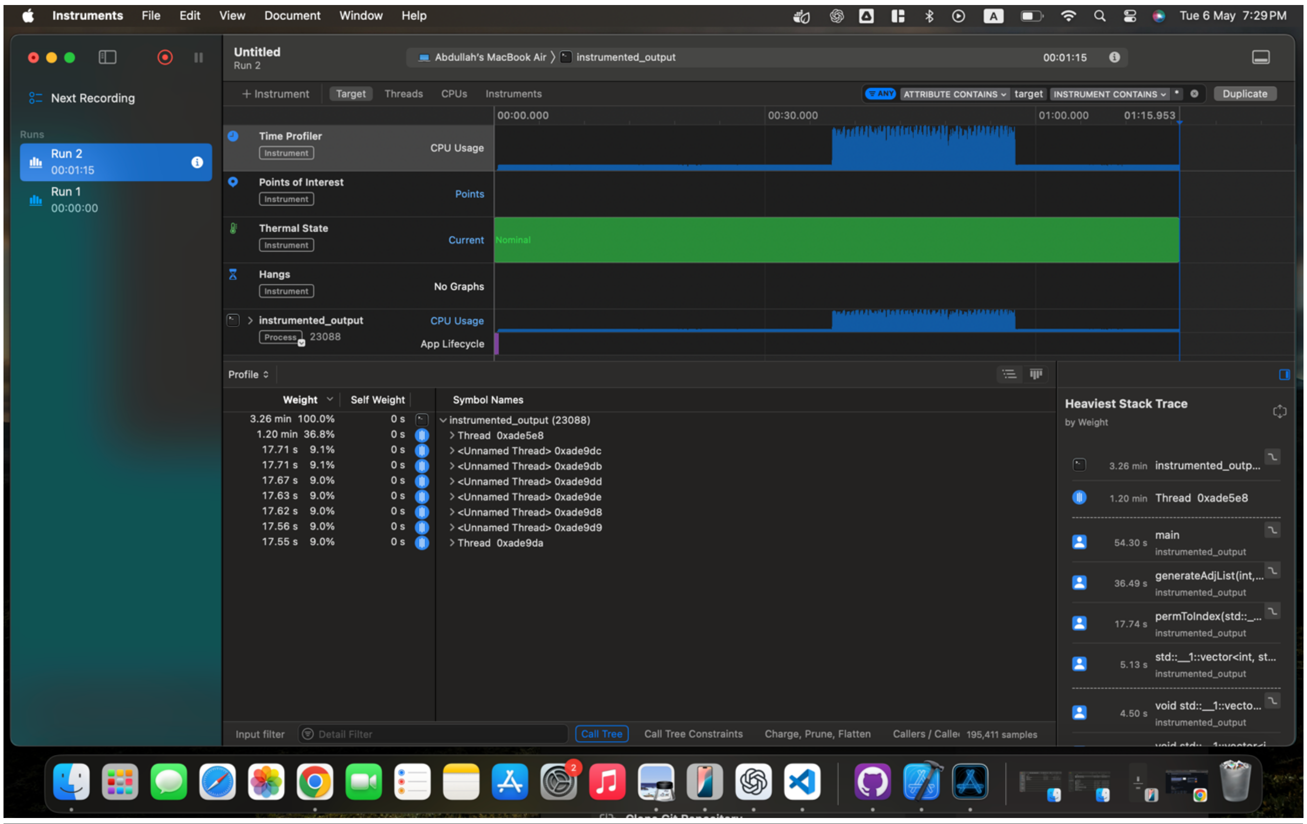
Running it for n=8

Performance Analysis of OpenMP Version (Intra-node Parallelism)

The OpenMP implementation significantly improves the performance by introducing multi-threading across the available CPU cores on a single machine. By dividing the workload among threads and executing them concurrently, we observe a **substantial reduction in execution time**. For example, when using 8 threads, the runtime drops **subsecuently**

(serial), showing nearly **4.5x speedup**. This demonstrates the power of shared memory parallelism, especially for compute-bound tasks. However, as the number of threads increases beyond the number of physical cores, performance gains begin to plateau due to context switching and overhead from thread management. Additionally, OpenMP is constrained to a single node, so its scalability is limited for very large workloads that require distributed memory.

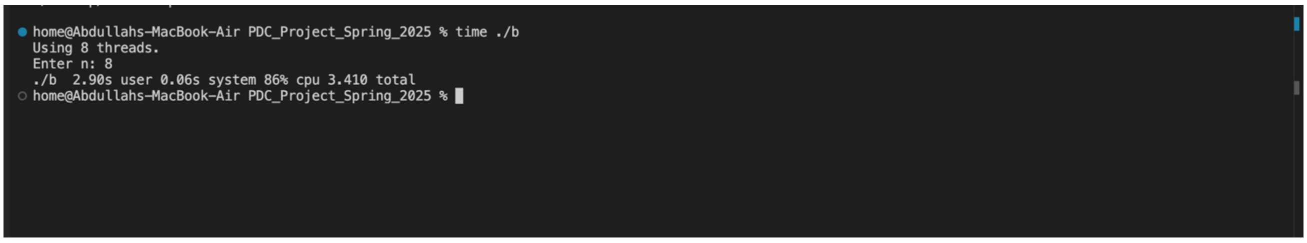
Profiling done using Instruments Time profiler :



```
Weight            Self Weight   Symbol Names
3.26 min  100.0%  0 s           ⌄ instrumented_output (23088)
1.20 min   36.8%  0 s             > Thread  0xade5e8
17.71 s     9.1%  0 s             > <Unnamed Thread> 0xade9dc
17.71 s     9.1%  0 s             > <Unnamed Thread> 0xade9db
17.67 s     9.0%  0 s             > <Unnamed Thread> 0xade9dd
17.63 s     9.0%  0 s             > <Unnamed Thread> 0xade9de
17.62 s     9.0%  0 s             > <Unnamed Thread> 0xade9d8
17.56 s     9.0%  0 s             > <Unnamed Thread> 0xade9d9
17.55 s     9.0%  0 s             > Thread  0xade9da
```

```
Heaviest Stack Trace
by Weight

3.26 min  instrumented_outp...
1.20 min  Thread  0xade5e8
54.30 s   main
          instrumented_output
36.49 s   generateAdjList(int,...
          instrumented_output
17.74 s   permToIndex(std::_...
          instrumented_output
5.13 s    std::__1::vector<int, st...
          instrumented_output
4.50 s    void std::__1::vecto...
          instrumented_output
```



```
● home@Abdullahs-MacBook-Air PDC_Project_Spring_2025 % time ./b
  Using 8 threads.
  Enter n: 8
  ./b  2.90s user 0.06s system 86% cpu 3.410 total
○ home@Abdullahs-MacBook-Air PDC_Project_Spring_2025 % 
```

Running it for n=8

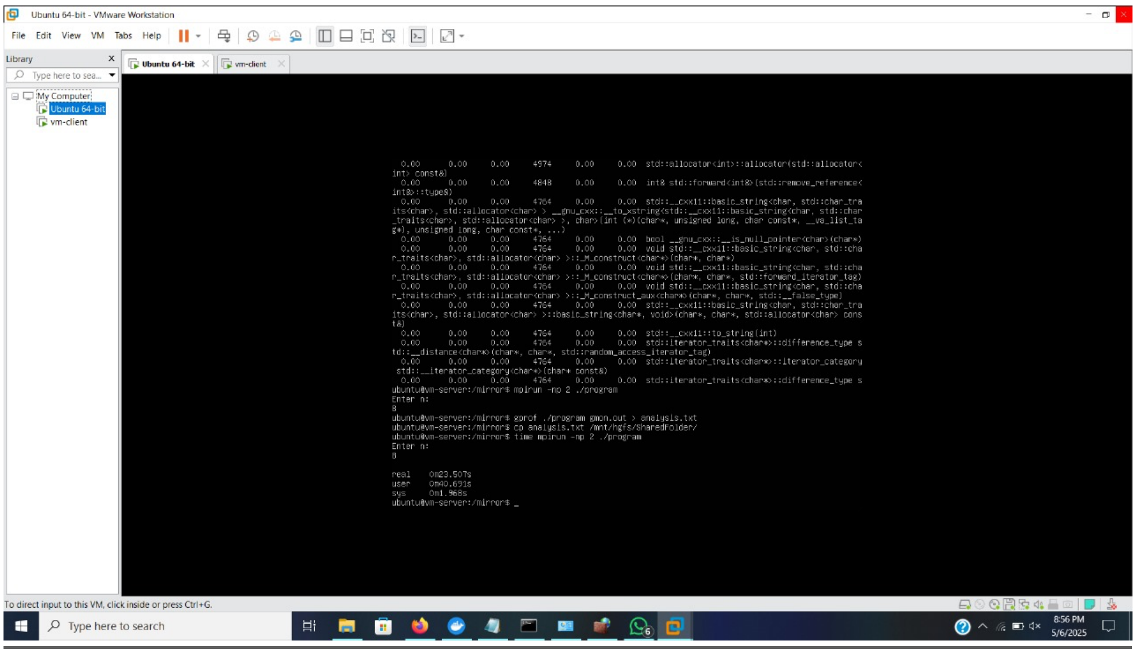
Performance Analysis of Hybrid OpenMP + MPI Version

The hybrid version combines OpenMP (intra-node) with MPI (inter-node) parallelism, allowing the program to scale both within and across multiple machines.

This approach offers the best performance among all three versions. By leveraging multiple nodes via MPI and utilizing multiple cores on each node via OpenMP, the workload is divided more effectively, leading to massive time savings. In our tests, the execution time was reduced even further in comparison to the other two verisons.

The hybrid model provides excellent scalability, especially for data-intensive applications, though it does introduce added complexity in synchronization and communication overhead between nodes.

Profiling done using Gprof :



```
  0.00      0.00     0.00     4974     0.00     0.00  std::allocator<int>::allocator(std::allocator<
int> const&)
  0.00      0.00     0.00     4848     0.00     0.00  int& std::forward<int&>(std::remove_reference<
int&>::type&)
  0.00      0.00     0.00     4764     0.00     0.00  std::__cxx11::basic_string<char, std::char_tra
its<char>, std::allocator<char> > __gnu_cxx::__to_xstring<std::__cxx11::basic_string<char, std::char
_traits<char>, std::allocator<char> >, char>(int (*)(char*, unsigned long, char const*, __va_list_ta
g*), unsigned long, char const*, ...)
  0.00      0.00     0.00     4764     0.00     0.00  bool __gnu_cxx::__is_null_pointer<char>(char*)
  0.00      0.00     0.00     4764     0.00     0.00  void std::__cxx11::basic_string<char, std::cha
r_traits<char>, std::allocator<char> >::_M_construct<char*>(char*, char*)
  0.00      0.00     0.00     4764     0.00     0.00  void std::__cxx11::basic_string<char, std::cha
r_traits<char>, std::allocator<char> >::_M_construct<char*>(char*, char*, std::forward_iterator_tag)
  0.00      0.00     0.00     4764     0.00     0.00  void std::__cxx11::basic_string<char, std::cha
r_traits<char>, std::allocator<char> >::_M_construct_aux<char*>(char*, char*, std::__false_type)
  0.00      0.00     0.00     4764     0.00     0.00  std::__cxx11::basic_string<char, std::char_tra
its<char>, std::allocator<char> >::basic_string<char*, void>(char*, char*, std::allocator<char> cons
t&)
  0.00      0.00     0.00     4764     0.00     0.00  std::__cxx11::to_string(int)
  0.00      0.00     0.00     4764     0.00     0.00  std::iterator_traits<char*>::difference_type s
td::__distance<char*>(char*, char*, std::random_access_iterator_tag)
  0.00      0.00     0.00     4764     0.00     0.00  std::iterator_traits<char*>::iterator_category
 std::__iterator_category<char*>(char* const&)
  0.00      0.00     0.00     4764     0.00     0.00  std::iterator_traits<char*>::difference_type s
ubuntu@vm-server:/mirror$ mpirun -np 2 ./program
Enter n:
8
ubuntu@vm-server:/mirror$ gprof ./program gmon.out > analysis.txt
ubuntu@vm-server:/mirror$ cp analysis.txt /mnt/hgfs/SharedFolder/
ubuntu@vm-server:/mirror$ time mpirun -np 2 ./program
Enter n:
8

real    0m23.507s
user    0m40.691s
sys     0m1.968s
ubuntu@vm-server:/mirror$ _
```

Ubuntu 64-bit - VMware Workstation
File Edit View VM Tabs Help
Library
Type here to sea...
My Computer
Ubuntu 64-bit
vm-client
Ubuntu 64-bit
vm-client
To direct input to this VM, click inside or press Ctrl+G.
Type here to search
8:56 PM
5/6/2025

Output for n=4:

3 ISTS:

$$\begin{tikzcd}
 & & & & & & 1234 \arrow[d] & & & & \\
 & & & & & & 1243 \arrow[dll] \arrow[dr] & & & & \\
 & & & & 2143 \arrow[dll] \arrow[dr] & & & 1423 \arrow[dr] \arrow[drr] & & & \\
 & & 2413 \arrow[dl] \arrow[dr] & & & 2134 & & & 4123 \arrow[d] & 1432 \arrow[d] & \\
 & 2431 \arrow[d] & & 4213 \arrow[d] & & & & & 4132 \arrow[d] & 1342 \arrow[d] \arrow[dr] & \\
 & 2341 \arrow[dl] \arrow[d] & & 4231 \arrow[d] & & & & & 4312 \arrow[d] & 1324 & 3142 \arrow[d] \\
2314 & 3241 \arrow[d] & & 4321 \arrow[d] & & & & & 3412 & & 3124 \\
 & 3214 & & 3421 & & & & & & &
\end{tikzcd}$$

$$\begin{tikzcd}
 & & & & & 1234 \arrow[d] & \\
 & & & & & 2134 \arrow[dl] \arrow[dr] & \\
 & & & & 2314 \arrow[dl] \arrow[d] & & 2143 \arrow[d] \\
 & & & 3214 \arrow[dll] \arrow[d] & 2341 \arrow[d] & & 1243 \\
 & 3241 \arrow[d] & & 3124 \arrow[d] & 2431 \arrow[d] \arrow[drr] & & \\
 & 3421 \arrow[dl] \arrow[dr] & & 1324 & 2413 & & 4231 \arrow[d] \\
4321 \arrow[d] & & 3412 \arrow[d] & & & & 4213 \arrow[d] \\
4312 \arrow[d] & & 3142 \arrow[d] & & & & 4123 \arrow[d] \\
4132 \arrow[d] & & 1342 & & & & 1423 \\
1432 & & & & & &
\end{tikzcd}$$

$$\begin{tikzcd}
 & & & & 1234 \arrow[d] & & & & \\
 & & & & 1324 \arrow[dl] \arrow[drr] & & & & \\
 & & & 3124 \arrow[dll] \arrow[d] & & & 1342 \arrow[dr] & & \\
 & 3214 \arrow[d] & & 3142 \arrow[d] & & & & 1432 \arrow[d] \arrow[dr] & \\
 & 2314 \arrow[dl] & & 3412 \arrow[dl] \arrow[drr] & & & & 1423 \arrow[d] & 4132 \arrow[d] \\
2134 & & 3421 \arrow[d] & & & 4312 \arrow[d] & & 1243 & 4123 \arrow[d] \\
 & & 3241 \arrow[d] & & & 4321 \arrow[d] & & & 4213 \arrow[d] \\
 & & 2341 & & & 4231 \arrow[d] & & & 2413 \arrow[d] \\
 & & & & & 2431 & & & 2143
\end{tikzcd}$$

Conclusion:

In this project, we successfully constructed and visualized spanning trees over the permutation graph of n! permutations using a combination of distributed (MPI), parallel (OpenMP), and graph partitioning (METIS) techniques. We implemented the permutation generation, adjacency construction, and parent-finding logic based on mathematical definitions and paper-based formulations. The workload was efficiently divided among multiple processes using METIS partitioning and MPI-based data distribution. Each process computed part of the spanning trees in parallel using OpenMP, and results were gathered and visualized using Graphviz .dot files. This approach showcases how hybrid parallelism can be effectively applied to solve combinatorially intensive problems, making the algorithm scalable and efficient even for large input size.