Optimizing Traveling Salesman with Branch and Bound

Eliot Atlani[1], Xenia Dela Cueva[1], Alice Cheng[1], and Itamar Rocha Filho[1]

[1]Harvard University

September 5, 2025

Abstract

This project implements and analyzes parallel solutions to the Traveling Salesman Problem (TSP) using the Branch and Bound algorithm. We developed and benchmarked four implementations: a sequential baseline, an OpenMP shared-memory version, an MPI distributed-memory version, and a hybrid MPI+OpenMP approach. Our solutions optimize work distribution and communication patterns to maximize parallel efficiency. We conducted performance analysis including sequential threshold optimization, strong scaling, and weak scaling tests. Our results demonstrate that for moderate problem sizes (15-20 cities), the MPI implementation achieved the best performance. Meanwhile, the hybrid implementation exhibited better scaling potential on larger problem instances (e.g., 24 cities) spanning multiple nodes. We identify bottlenecks in each implementation, including load imbalance, communication overhead, and memory access patterns. This work offers strategies for parallelizing combinatorial optimization problems and applies to other branch-and-bound algorithms in logistics and operations research.

1 Background and Significance

Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is a NP-hard problem in combinatorial optimization, where the objective is to find the shortest possible route that visits each city exactly once and returns to the starting point. Solutions are computationally expensive from the factorial growth of the solution space, making the problem intractable for large instances without advanced optimization techniques.

The Brand-and-Bound algorithm

The Branch-and-Bound (B&B) algorithm offers an exact approach by systematically pruning suboptimal paths in the search tree using bounds. However, it suffers from performance bottlenecks due to the sequential nature of recursive tree expansion and bound computation.

The B&B algorithm can be parallelized at multiple levels: OpenMP can be used to speed up fine-grained operations like branching, bounding, and local search tree exploration within each process, while MPI can distribute subtrees of the search space across nodes as well as sharing updates.

Significance and comparison

Our hybrid approach combines the shared-memory efficiency of OpenMP with the distributed computing power of MPI, enabling high-performance execution on HPC systems. While prior work often adopts

a single parallelization model, we implemented and compared four versions: a sequential baseline, an OpenMP version using task-based parallelism with thresholding, an MPI version with static distribution of root-level branches, and a hybrid MPI+OpenMP version that combines coarse-grained distribution across nodes with fine-grained parallelism within each node. By supporting effective pruning and balanced workload distribution, our approach extends the scalability of exact TSP solvers and offers a flexible foundation for tackling other combinatorial optimization problems.

TSP is crucial for routing logistics in companies like Amazon, UPS, FedEx, Instacart, and DoorDash, which must deliver thousands of packages efficiently. As stop counts grow factorially and real-world conditions change constantly, TSP solutions must be fast and adaptive. Optimizing these problems allows real-time route updates, cutting costs and ensuring on-time service.

2 Scientific Goals and Objectives

In this project, our goal is to build an implementation that can solve larger traveling salesman problems using the branch and bound algorithm. Our two main objectives are:

1. Scale branch-and-bound to larger instances
2. Quantify and optimize parallel efficiency

Since the travelling salesman problem is NP-hard with exponential search complexity, we implemented a serial benchmark of TSP, a parallelized version using only OpenMP, a parallelized version using only MPI, and a hybrid parallelized version using both OpenMP and MPI. We then compared the relative performance of these implementations, as well as the relative performance of different node/thread configurations for a given implementation.

3 Algorithms and Code Parallelization

Branch and Bound TSP implementation

The core algorithm follows a classic recursive B&B structure. A complete distance matrix is first made by computing the shortest path between every pair of TSP nodes using Dijkstra's algorithm, applied on a general graph structure. This matrix acts as the cost function for evaluating partial tours.

At each recursive step:

1. The algorithm checks whether a complete tour has been formed.
2. If incomplete, a lower bound is calculated by summing the current path cost and the minimum outgoing edge for each unvisited node (a simple yet effective bounding heuristic).
3. Subtrees with a bound greater than the current best tour cost are pruned.
4. Otherwise, the algorithm explores each unvisited city recursively

Initial Memory Requirements

Each process maintains $N \times N$ distance matrix of `double` values (`distMatrix`), requiring $O(N^2)$ space. Since each `double` is 8 bytes, this is approximately 8 MB for $N = 1000$. The graph is also stored as an adjacency list, using $O(E)$ memory, where E is the number of edges. For dense graphs ($E \approx N^2$), this can take up to ∼12 MB at $N = 1000$. Each process or thread keeps a copy of `bestTour` ($O(N)$), `bestCost` (one `double`), and temporary structures like visited arrays and partial paths. Overall, total memory usage per process is about 20 MB for $N = 1000$. This scales quadratically with N and linearly with the number of MPI processes. All SLURM benchmark scripts (serial, OpenMP, MPI, and hybrid) requested 16 GB of memory per job, which was sufficient for all problem sizes tested (up to 24 cities).

OpenMP implementation

This version uses OpenMP to parallelize the recursive Branch and Bound algorithm for TSP on shared-memory systems. Recursive branching is parallelized using `#pragma omp task` inside a `#pragma omp taskgroup`, with tasks only spawned when the number of remaining cities exceeds a `sequentialThreshold` to avoid excessive overhead. Shared variables (`bestCost`, `bestTour`) are protected by an `omp_lock_t` (`bestLock`), which is initialized via `omp_init_lock()` and managed using `omp_set_lock()` and `omp_unset_lock()`. To minimize locking during reads, `#pragma omp atomic read` is employed for pruning checks, allowing low-overhead access to `bestCost`. The parallel region begins with `#pragma omp parallel`, and a single thread is designated as the root with `#pragma omp single` to coordinate task creation. Within each task, bound computations and recursive calls execute sequentially to reduce complexity and locking, though further vectorization can be used in future optimizations.

OpenMP enables multiple threads to concurrently explore different regions of the search space and benefits from work stealing from the task scheduler. Eliot Atlani was the main contributor for this implementation. While OpenMP works well on single-node, shared-memory systems, it does not scale across multiple machines. This limitation leads us to explore MPI.

MPI

The MPI implementation, with the code developers being Xenia Dela Cueva, Alice Cheng , and Eliot Atlani, runs the TSP Branch and Bound in parallel on multiple machines with separate memory. It splits the search tree across processes to speed up solving TSP, which becomes much harder as the number of cities increases.

The implementation begins with root-level branches (like second city choices) being assigned to processes using $(i-1)$ mod numProcs = rank so that each process explores a different part of the search space. Each rank builds its own distance matrix using the Floyd-Warshall algorithm (`buildDistanceMatrix`) and executes a recursive search with `explore()`.Communication and synchronization is done with `MPI_Allreduce` and `MPI_MINLOC` to identify the process with the lowest-cost tour. This is broadcast to all processes using `MPI_Bcast`. These operations aim to be synchronization points that minimize overhead communication. This implementation supports multi-node execution by using commands like `mpirun -np 8 ./tsp_mpi`. This enables each node to run one or more processes, allowing the algorithm to scale beyond the limits of OpenMP.

MPI reduces runtime by splitting computation at the top-level branches, which is the most expensive level. It does require each process to maintain a full copy of the graph and distance matrix in memory,

but avoids interprocess communication during search. There would be no disk I/O bottlenecks during execution. This pure-MPI solution is ideal for HPC systems with distributed nodes. Although alternatives like dynamic load balancing could improve efficiency for uneven branches, static distribution worked for our experiments.

Hybrid

To take advantage of both process-level and thread-level parallelism, we combined our two previous approaches to create a hybrid approach using both OpenMP and MPI. Alice Cheng was the primary contributor for this implementation. We first enhanced the branch-and-bound approach with a precomputed all-pairs shortest-path matrix. We loaded the input graph via a custom `Graph` class and computed its distance closure with the Floyd–Warshall algorithm, storing an $N \times N$ matrix of doubles (time complexity $\mathcal{O}(N^3)$, memory $\mathcal{O}(N^2)$). This approach ensures tight lower bounds for pruning. An alternative is to run Dijkstra's algorithm from each node (N times, $\mathcal{O}(N\,E \log N)$) or to use heuristic relaxations (e.g. 1-tree), but Floyd–Warshall was chosen for its conceptual simplicity and uniform cost structure for moderate N.

For the branch-and-bound algorithm, we used a recursive approach; from a partial path we compute a bound by adding the minimum outgoing edge from the current city plus, for each unvisited city, its cheapest exit edge (or return edge to the depot). Subtrees whose bounds exceed the current global best are pruned. On completion of each full tour, the best cost and tour are updated atomically under an OpenMP lock.

The core dependencies for this implementation are the C++17 standard library, OpenMP (for intra-node task parallelism), and MPI (for inter-node coordination). An overview of the two parallelism levels is:

- **MPI:** Each rank receives a disjoint subset of branches of the first step. After a local search, the ranks perform an `MPI_Allreduce` with `MPI_MINLOC` to identify the best global cost and its owner, followed by an `MPI_Bcast` of the winning tour.
- **OpenMP:** Within each MPI process, the remaining search tree is explored through OpenMP tasks organized by a user-tunable 'sequential threshold' on subtree size. A global `omp_lock_t` serializes updates to the shared bestCost variable.

Our implementations do not use GPU or CUDA/OpenACC acceleration. Memory usage is dominated by the $\mathcal{O}(N^2)$ distance matrix (e.g. for $N = 1000$, $\approx$8 MB) and the sparse adjacency list (proportional to edge count). In addition to `OpenMP` and `MPI` for parallelism, we used standard C++ STL libraries like `vector`, `queue`, `algorithm`, `limits`, and `chrono` for data structures and built-in functions. The input instances were from `TSPLIB`, a benchmark dataset for Traveling Salesman Problem (TSP) instances, as well as a custom connected graph. I/O is limited to a single file read of the input graph; all subsequent data resides in RAM.

This hybrid design balances the coarse-grain MPI distribution of independent subtrees with fine-grain OpenMP tasking, yielding scalable performance on CPU-based clusters without specialized accelerators.

Validation, Verification

To ensure the correctness of our parallel implementations, we employed a multi-faceted validation strategy. First, we verified our solutions against the TSPLIB benchmark library, which contains reference TSP

instances with known optimal solutions. For smaller instances, we cross-validated the tour costs produced by all four implementations (sequential, OpenMP, MPI, and hybrid) to ensure they all converged to identical optimal costs.

For validation on randomly generated instances, we used the sequential implementation as our ground truth reference. We implemented deterministic pruning strategies to ensure that different execution orders across parallel runs would still produce identical optimal tour costs, even if the exact path representation differed due to equivalent-cost alternatives.

Our validation process included:

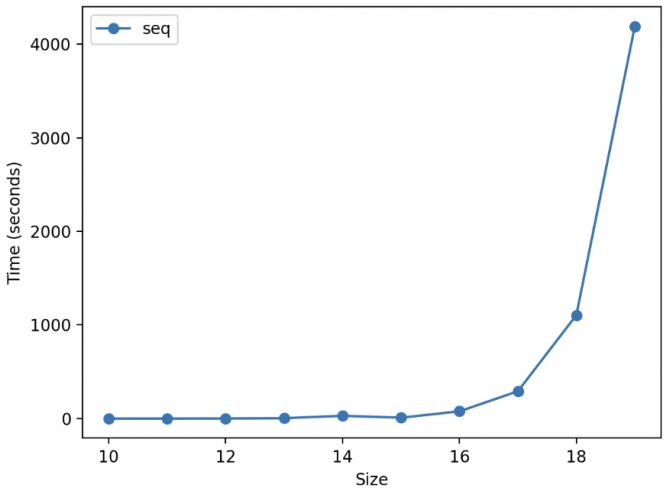
- Consistency checks ensuring all parallel implementations produce identical optimal costs
- Verification that the final tours are valid (visit each city exactly once and return to start)
- Validation of bound calculation correctness to ensure proper pruning

To verify numerical consistency, we implemented double-precision arithmetic throughout the codebase and ensured identical computation of distance matrices across all implementations. The Floyd-Warshall algorithm was validated independently to confirm accurate distance calculations between city pairs. This aspect of the implementation was handled by Itamar Rocha Filho, who ensured consistency and correctness across versions.

4 Performance Benchmarks and Scaling Analysis

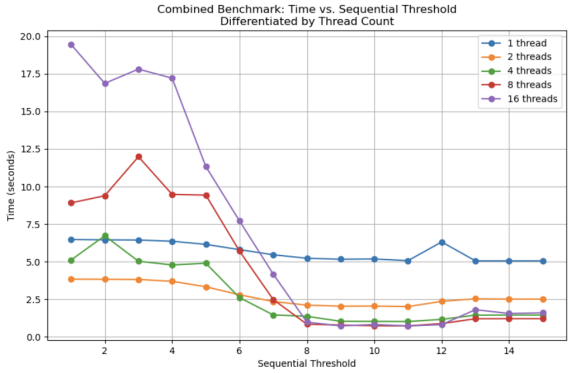
We first ran our sequential TSP solver to find optimal paths for up to 19 cities. Figure 1 shows an exponential increase in runtime as the number of cities to visit increases. To begin our parallelization

Figure 1: Runtime of Sequential Implementation for Varying TSP Problem Sizes



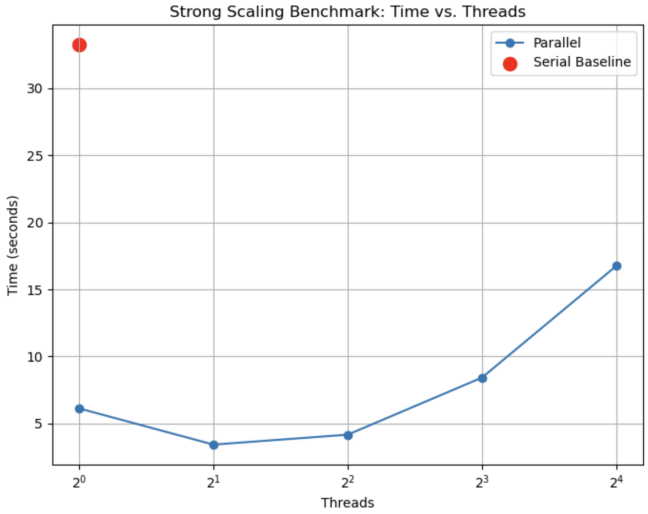
efforts, we focused on using only OpenMP. Our first step was to parallelize the search step for the first city, leaving the rest of the algorithm to use the sequential implementation. We observed an improvement, and proceeded to run experiments varying the sequential threshold, or the maximum number of cities along the path to use the parallel algorithm to search before switching back to sequential. We also varied

Figure 2: Sequential Threshold Analysis for OpenMP Implementation



the number of threads. In figure 2, we show the results of this experiment for a 15 city tour. The optimal sequential threshold is around 8 to 12 cities, with the best performance from using 8 or 16 threads. We also performed a strong scaling analysis for this implementation as shown in figure 3. The plot shows an

Figure 3: Strong Scaling of OpenMP Implementation



initial improvement in runtime with an increase in number of threads but then the runtime increases for higher number of threads, likely due to added overhead from thread coordination/communication.

We then created a parallel implementation using only MPI, with runtimes as shown in figure 4. We observe a significant improvement in runtime when using more MPI processes, with diminishing returns beyond 8 processes. This is likely due to increasing overhead resulting from collective MPI operations, such as MPI_Allreduce, whereby processes need to communicate essential information and updates with each other. Since our problem size is relatively small (15 cities), the overhead is more significant relative to the actual runtime. Finally, we combined our OpenMP and MPI implementations for a hybrid approach. We tested a few configurations and compared their runtimes, as shown in figure 5. Again, we see that there is a significant improvement in runtime when increasing the number of processes, but that there are diminishing returns beyond 16 processes. In addition, when using 1 process and 4 threads, we would

Figure 4: Runtime of MPI-only Implementation

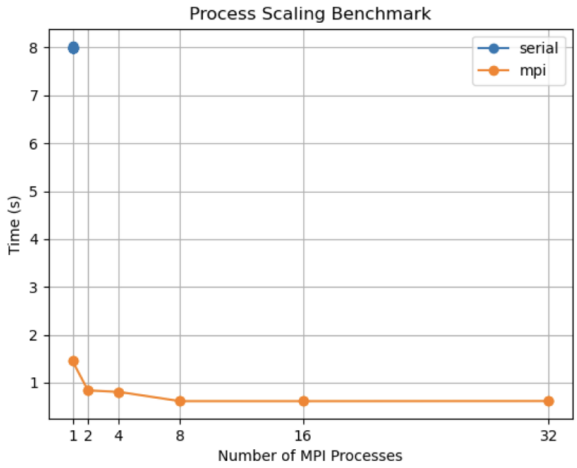
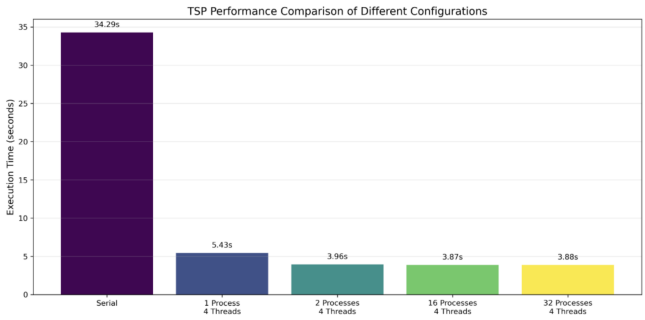


Figure 5: Comparison of Runtimes for Different Configurations of Hybrid Implementation



expect a runtime of $\frac{34.29}{4} = 8.57$ seconds in the ideal case, but our runtime is significantly quicker. Some potential explanations include:

1. **Cache effects**: Since each thread works on a smaller subset of the problem, more of the working set for each thread fits into the cache, leading to better cache utilization.

2. **Hardware architecture benefits**: Modern CPUs often have shared caches and other architectural features that parallel code can leverage more efficiently than serial code.

3. **Compiler optimizations**: The compiler might apply different optimization strategies to parallel code, resulting in more efficient machine code.

We also performed a similar sequential threshold analysis for the hybrid implementation in figure 6. Again, we see an optimal sequential threshold of around 7 to 10 cities with the best performance from 32 processes and 4 threads per process. We also performed strong-scaling analysis for the hybrid implementation, as shown in figure 7. In this plot, we see a speedup in runtime as the total number of threads (multiple processes and multiple threads per process) increases. Using the optimal configurations for each implementation, we compared the runtimes of each implementation for problem sizes up to 19 cities in figure 8. Our MPI-only implementation performed the best, followed by the hybrid approach and then OpenMP-only. Some possible explanations for why the hybrid approach was worse than the MPI-only approach are:

Figure 6: Sequential Threshold Analysis of Hybrid Implementation

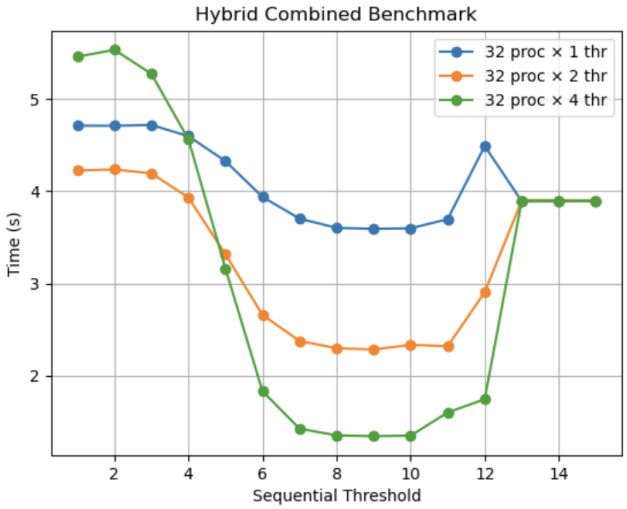
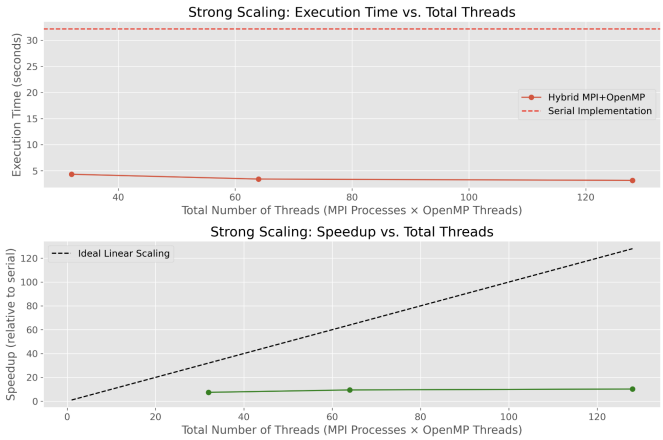


Figure 7: Strong Scaling Analysis of Hybrid Implementation



1. **Overhead costs**: The hybrid approach accumulates overheads from both systems - MPI's message passing and process management plus OpenMP's thread creation and synchronization. These combined overheads can reduce performance.

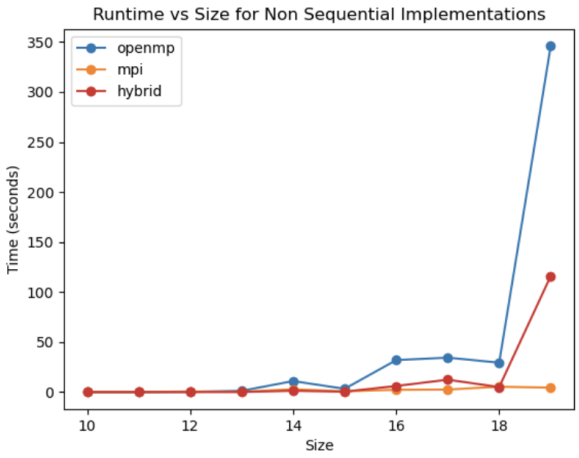
2. **Load balancing complications**: Balancing work becomes more complex when considering two levels of parallelization. If either level has imbalances, the overall performance suffers.

3. **Memory hierarchy utilization**: Pure MPI might make better use of distributed memory resources, while the hybrid approach has to manage both shared memory (OpenMP) and distributed memory (MPI) models.

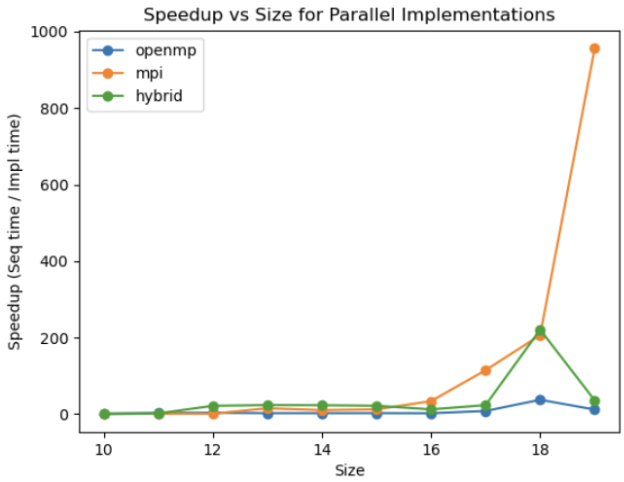
4. **Implementation quality**: The pure MPI implementation might simply be more optimized than the hybrid one, which is often more complex to tune perfectly with the extra considerations of the interaction between OpenMP and MPI.

Table 2 shows the workflow parameters for slurm jobs used to run our experiments and simulations for all implementations, since we ran multiple implementations in the same job for comparison purposes.

Figure 8: Runtime and Speedup Comparisons of Parallel Implementations



(a) Runtime comparison of parallel implementations



(b) Speedup comparison of parallel implementations

Implementation	**Threads / Processes**	**Time (s)**
Serial	1	32.142100
OpenMP	16	16.744800
MPI	16	0.614629
Hybrid (OpenMP + MPI)	16	0.614712

Table 1: TSP runtime comparison across implementations using 16 threads/processes. Values taken from respective benchmark CSVs.

	Values
Typical wall clock time (minutes)	3
Typical job size (nodes)	16
Memory per node (GB)	16
Maximum number of input files in a job	1
Maximum number of output files in a job	1
Library used for I/O	fstream, iomanip

Table 2: Workflow parameters of slurm jobs used during project development.

5 Resource Justification

To estimate computational needs, we benchmarked our Traveling Salesman Problem (TSP) on Harvard's cluster. The hybrid MPI + OpenMP implementation took approximately 160 seconds to solve a 24-city instance using 2 MPI ranks (1 per node) and 32 OpenMP threads per process. This corresponds to:

$$\text{node hours} = 2\,\text{nodes} \times \frac{160\,\text{seconds}}{3600\,\frac{\text{seconds}}{\text{hour}}} \approx 0.0889\,\text{node hours}$$

Each production run solves one TSP instance. We tested different algorithms (serial, OpenMP, MPI, hybrid) and varied parameters like random seed, threshold, and city count. We define two simulation classes: smaller graphs (19 cities) for comparison, and larger graphs (24 cities) for testing the hybrid approach as seen in Table 3.

	Simulations	Node hours / simulation	Total node hours
Small Graphs (19 cities)	20	0.0889	1.778
Large Graphs (24 cities)	50	0.2667	13.335

Table 3: Estimated node hour usage based on actual TSP benchmarks

The estimated usage is 15.11 node hours (15.11 hours on one node, or about 3.78 hours if run across 4 nodes). Each simulation is short but offers a baseline for scaling to larger or more complex TSP applications.The hybrid implementation uses MPI to split top-level search branches across processes in a distributed-memory setup, and OpenMP for task-based recursion within each process using shared-memory parallelism. We do not use GPU acceleration (CUDA/OpenACC). Memory usage increases with problem size due to 3 main factors: the $\mathcal{O}(N^2)$ distance matrix, recursive stack depth from branch-and-bound, and overhead from OpenMP task scheduling. For problems with more than 20 cities, memory becomes a limiting factor— especially when many branches are explored concurrently. Effective pruning is essential not only for computational speed but also for managing memory. Our benchmarks in the 19–24 city range (using 16–32 threads or processes) showed significant memory usage, with requirements expected to grow exponentially for larger instances. Future scaling will require careful resource planning to manage both computation and memory demands.

References

[1] Harvard University. *Lecture 8: OpenMP Basics*. Cambridge, Massachussetts, 2025.

[2] Harvard University. *Lecture 13: MPI: Beyond the Basics*. Cambridge, Massachussetts, 2025.

[3] Heidelberg University. *TSPLIB: A library of sample instances for the TSP*. Heidelberg, Germany, 1993.

[4] Indian Institute of Technology *Travelling Salesman Problem: Parallel Implementations Analysis*. Bombay, Mumbai, India, 2021.