Traveling salesman – OpenMP implementation report
PDS - Parallel and Distributed Computing

Group n. 38
Jan Sáblík - ist1107873
David Bilnica - ist1107874

We were working on this project since last weekend every day, but unfortunately, we were not able to implement OpenMP parallelization with desired speedup. We were not sure how the parallelization should be handled, like that it should use multiple queues. We tried many different versions of parallel implementation, and our latest version is dividing the tree between the threads and their queues and is using work stealing as load balancing strategy.

1) Approach used for parallelization

The parallelization is done using OpenMP, which is a shared-memory parallel programming model. The algorithm parallelizes the exploration of the solution space, where each thread starts from a different node and explores possible tours. Work stealing is used to keep threads busy when they run out of work in their queue.

2) What decomposition was used

The algorithm creates a tree structure, where each node represents a partial tour, and the tree's depth corresponds to the number of visited cities. The root node is the initial city, and the first layer of the tree is created by connecting the initial city to its nearest neighbors. Each thread starts by exploring a different node from this first layer. As the threads explore the tree, they generate new nodes (partial tours) by adding cities to the current tour. Each new node corresponds to a subproblem of visiting the remaining cities.

This decomposition allows the parallel exploration of the solution space, where each thread investigates different parts of the tree concurrently. The decomposition is dynamic and adapts to the problem's structure, as the threads generate new nodes while exploring the search space. This approach leads to an efficient parallel exploration of the solution space, as it ensures that the workload is distributed among the threads, and the work stealing mechanism addresses load balancing issues that may arise during the search.

3) Synchronization concerns and why

Updating the *best tour* and its *cost*: These variables are shared among the threads, and they need to be updated atomically to avoid race conditions. The *#pragma omp atomic write* directive is used for this purpose.

Adding a new node to the thread's queue: The code uses a critical section *(#pragma omp critical*) to protect the access to the thread's queue when adding a new node. This ensures that only one thread can add a new node to the queue at a time, avoiding potential data corruption.

Comparing and updating the best tour at the end of the search: A critical section is used when updating the best tour and its cost to avoid race conditions.

4) How was load balancing addressed:

Load balancing is addressed using work stealing. When a thread's queue is empty, it tries to steal work from other threads' queues. This ensures that threads do not remain idle when there is still work to be done.

5) Performance results and if they are what you expected

Since we were not able to successfully implement parallelization and the code ends up in a segmentation fault the performance is definitely not what it should be. We tried our best to make it work.
However, the parallelization approach using OpenMP and work stealing should provide a decent speedup compared to a sequential version.