

I. Parallelization Approach

This project phase started with testing the direct parallelization of the default algorithm proposed in the project guidelines using a single shared queue. However, upon using the VTune profiler, it was realized that the program did not run faster due to synchronization issues arising from the concurrent insertion and removal of elements to and from the shared queue. When considering the overheads, this approach was slower than the serial version.

To address the synchronization limitation, domain decomposition was employed, dividing the problem domain into smaller sub-domains, which different threads can independently solve. In this project, the search space of the TSP problem was divided into multiple sub-problems, with each sub-problem representing a partial tour of the TSP or a node on a given queue. Each thread was then assigned a sub-problem, and the best result was shared among all threads, leading to the overall minimum cost path at the end of the computation.

II. Software Implementation

Using domain decomposition and OpenMP allowed for efficiently distributing the workload among the threads and effectively exploited the TSP Branch and Bound algorithm's parallelism. These techniques are commonly used in parallel computing and are well-suited for large search space problems.



Implementing the approach described in Sect. I., a queue was created for each thread, with a preamble initializing all the queues using the neighbors of node 0, as seen in Alg. 1.

Algorithm 1 TSP Branch and Bound Parallel Version - Preamble.

```

1: procedure TSPBB(Distances, N, BestTourCost)
   #pragma omp parallel
2:   idx  $\leftarrow$  omp_get_thread_num() ■[idx]
   #pragma omp for
3:   for each neighbor v of 0 do
4:     newBound  $\leftarrow$  updated lower bound on tour cost
5:     if newBound > BestTourCost then  $\triangleright$  BestTourCost starts with the limit argument
6:       Continue
7:     end if
8:     newTour  $\leftarrow$  0 to v
9:     newCost  $\leftarrow$  Distances(0, v)
10:    Queue[idx].push(newTour, newCost, newBound, 2, v)
11:  end for ■[idx]
12:  (...)
13: end procedure

```



The parallel algorithm utilizes individual locks  and unlocks  in each thread queue during Popping and Pushing operations. OpenMP lock mutexes (`omp_set_lock()` and `omp_unset_lock()`, respectively) are used for locking and unlocking, with each queue having its own lock indexed by its thread id. This asynchronous behavior ensures that there will not be any data races or threads locked waiting, allowing every thread that has nodes in its queue to work independently of the others. This approach successfully addresses the main problem encountered in the project's early stages.

Data races and threads locked waiting may occur when multiple threads attempt to simultaneously balance their workloads and update shared variables such as *BestTour* and *BestTourCost*. In the worst-case scenario, waiting times can be as long as a single push or the updating time, $\mathcal{O}(N)$, respectively. As explained previously, mutexes are used to address the first case, and for the second case, `#pragma omp critical` is utilized. Moreover, although not presented in Alg. 3, a shared flag keeps track of the number of stopped threads, and updating this flag requires a `#pragma omp atomic`.

This implementation redistributes nodes through stopped threads to balance the workload, visible in Alg. 2. For example, when a thread encounters a node with a lower bound over the limit or the *BestTourCost*, it stops, clears its queue, and notifies all threads that it is stopped (since all nodes in the queue will not yield the solution, this will relieve memory usage). When searching for neighbors in a given tour, working threads will interleave work between stopped threads and themselves. If, in a given problem, node 0 does not have enough good neighbors for all the queues during the preamble, when entering in the core algorithm (Alg. 3), those threads will wait until other thread can fill their queues.

Algorithm 2 TSP Branch and Bound Parallel Version - Work distribution core.

```

1: for each neighbor  $v$  of  $Node$  and  $v \notin Tour$  do
2:   procedure IndexChoosing( $idx, waiting$ )
3:     find waiting threads other wise choose itself as target
4:     return target
5:   end procedure
6:   [target]  $Queue[target].push(newTour, newCost, newBound, Length + 1, v)$ 
7:    $waiting[target] \leftarrow false$  [target]
8: end for

```

A more active mechanism could be implemented for better performance and load balancing. For instance, if a given thread were to be stopped by the lower bound it is searching because it is considerably larger than the lower bound that other threads are searching, that given thread could then assist the other threads.

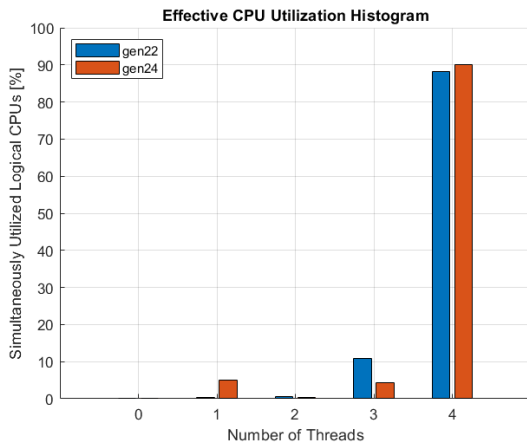
Several issues were encountered during program and algorithm testing, with the most significant problem being threads not getting stuck in the while loop on line 13 of Alg. 3 while waiting for work. The optimization flag `-O3` was optimizing away the crucial part that holds the threads in place, waiting until they could carry on working. As the attempt to use volatile casting did not work, the problem was resolved by adding a `#pragma omp atomic` inside the loop. As previously indicated, this led to the final implementation presented in the pseudo-code shown in Alg. 3.

Algorithm 3 TSP Branch and Bound Parallel Version - Core algorithm.

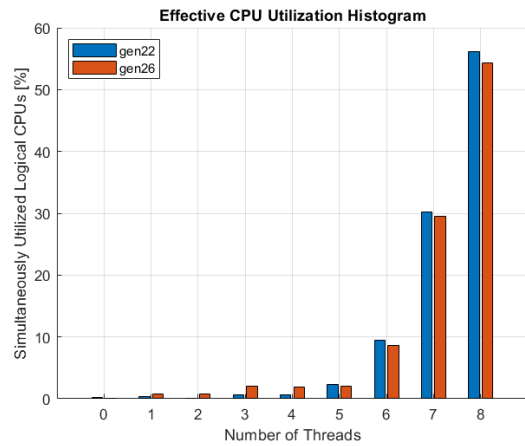
```
1: procedure TSPBB(Distances, N, BestTourCost)
  #pragma omp parallel
2:   idx  $\leftarrow$  omp_get_thread_num()
3:   (...)
4:   do
5:     if Queue[idx]  $\neq$  {} then
6:       run the algorithm provided in the guidelines while locking and unlocking queues
7:     else
8:       waiting[idx]  $\leftarrow$  true
9:     end if
10:    if Every thread is locked then    ▷ The last thread to finish has the responsibility to do it
11:      unlock all threads
12:    end if
13:    while waiting[idx] do    ▷ The loop that gets threads stuck waiting for nodes to process
14:      Continue
15:    end while
16:  while At least one thread has work
17:  return BestTour, BestTourCost
18: end procedure
```

III. Results

The results of running the VTune profiler on this implementation indicated that the queue control remained the main bottleneck, and there was a potential data race within the algorithm function. Although data races can occur elsewhere in the algorithm code, they primarily arise within the while loop on line 13 of Alg. 3 as false positives, as mentioned earlier. In particular, data races within the while loop tend to occur in problems that converge to a single optimal route, causing the algorithm to appear pseudo-sequentially towards the end of execution. Fig. 1 shows the effective CPU utilization histogram extracted from the VTune profiler, which illustrates the percentage of execution time utilizing different numbers of threads.



(a) *gen22* vs *gen24* tests, using 4 threads.



(b) *gen22* vs *gen26* tests, using 8 threads.

Fig. 1 – VTune profiler comparison output.

The results presented in Fig. 1 indicate that, in the case of four threads, for both tests, the code utilizes all the available threads approximately 90% of the execution time. However, with eight threads, some performance is being left out, primarily due to the scenario previously explained where the problem lacks valuable nodes for every queue/thread.

Besides the values obtained through the VTune Profiler, the project was also tested against several input files in order to obtain benchmark values for all possible thread configurations. The results were obtained by running a total of 10 times for each input and each thread configuration leading to a total of 240 runs. The final results comprise the average of the runs and are shown in Fig. 2 with the total speedup (in reference to the serial version) on the left side and the memory usage on the right side.

On Fig. 2a, it is possible to observe an almost pseudo-linear improvement for tests *gen20* and *gen22*, and a pseudo-logarithmic improvement for the others. On test *gen24* there was a speedup plateau'd starting from the 4 threads configuration.

The lack of speedup of test *gen24* can be partially explained by the high memory utilization, leading to the absence of space in the system and, consequently, the queue's search slowdown.

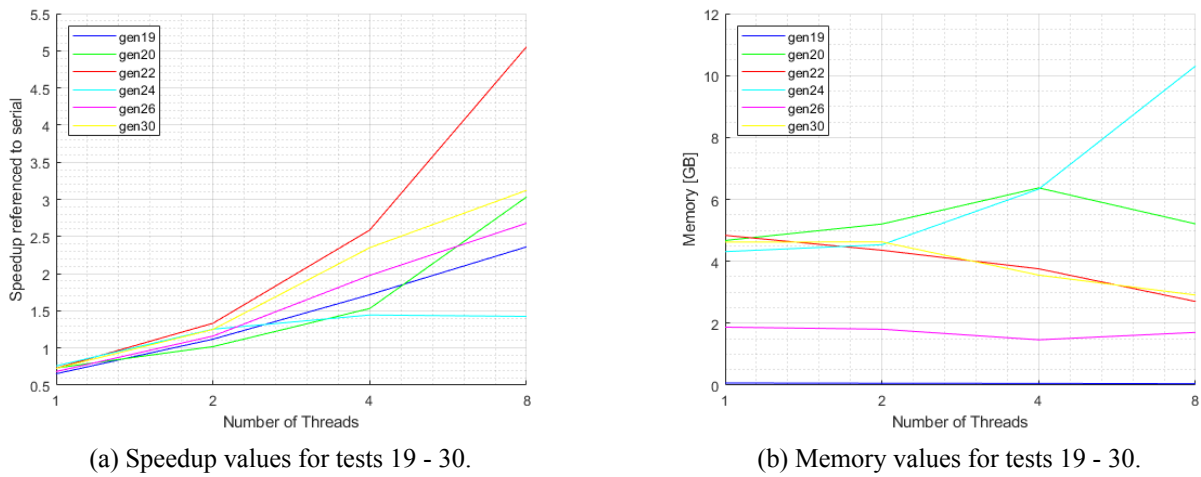


Fig. 2 – Results for the OMP implementation.

As expected and visible in Tab. 1, the mean of the speedup for one thread is lower than 1 (meaning slower than the serial version), this occurs due to the overhead introduced by the OpenMP directives utilized to allow parallelization. For the other thread configurations, the obtained speedup was higher than 1 meaning faster execution than serial, reaching a max of 5.0532 for 8 threads on test *gen24*.

Tab. 1 – Speedup analysis.

Speedup by thread	1	2	4	8
Max	0.7584	1.3328	2.5869	5.0532
Mean	0.6117	1.0190	1.6582	2.5247
Min	0.6538	1.0186	1.4441	1.4252