OpenMP implementation of TSP

Group 15

93230 Catarina Bento

95600 João Pereira

95618 Leonor Barreiros

1 Approach used for parallelization

Our program begins with the master thread sequentially expanding the search tree until there are $n\_threads^2$ nodes in the priority queue. Note that if the problem is small enough, it will ultimately be solved during this work, so no parallelization overhead is introduced.

Then, the master thread distributes those nodes between *n_threads* priority queues. It does so in an interleaving manner because the first distributed nodes will have a higher priority (since they're ordered in ascending cost in the queue), hence will be processed first by each thread. If the nodes weren't distributed in an interleaving manner, the first priority queue would have all nodes with lowest lower-bound, which would automatically give the other threads worse sub-problems to explore.

After this point, each thread is responsible for exploring its own part of the tree, by expanding the nodes in its private priority queue. Each tree will solve an independent search problem.

Periodically, threads will share their best solution. This will make threads with worse best solutions (i.e., threads that are exploring uninteresting parts of the tree) update their own and clear their priority queues, thus, pruning the tree.

2 Decomposition used

Our decomposition consists of dividing the search problem between the threads, i.e., each thread will explore a sub-tree. This is ensured by the sequential part of the code, where the master thread divides the initial nodes between all threads (including itself). More concretely, our decomposition consists of assigning the processing of each node to a different thread.

Furthermore, as we will explain in more detail soon, it may change during run-time if any thread has concluded processing their sub-tree while others still have work to do. Threads whose priority queues have become empty will signal that they need work, and threads with non-empty priority queues who notice that will give them work. The idea behind this came from (1).

3 Synchronization concerns (and why)

Our program achieves very good performance due to its low need for synchronization. More explicitly:

- each thread explores different nodes, and each thread has its own queues. Since they're private, they will not interfere with each other. So, they can run independently without waiting for each other or having critical sections;
- when threads share their best solution, it's possible for a thread to be reading an obsolete cost and its respective (or not) tour. If the cost matches the tour, this isn't a problem: eventually, the right tour/value will be set. If it doesn't, this could possibly give us a wrong solution. However, the probability of this happening is so small (in over 200 tests we ran, it never happened), that we concluded the overhead wasn't worth it.

However, we did synchronize access to the *shared_nodes* vector. This ensures that threads aren't able to add/remove nodes from the vector at the same time. This also has a low probability of happening, however, it has a risk of there being a Segmentation Fault. So, in this case, we concluded the overhead was worth it.

4 Load Balancing

As threads explore independent parts of the tree, it's possible that some of them will be exploring nodes with more (and better neighbors), and thus have more work to do than others. This may lead to some tasks having no processing left to do (when they have explored all nodes in their sub-problem, their queues will be empty), while others carry most of the load.

To deal with this, we implemented a load-balancing method that ensures that threads will not be without work for long. When a thread's queue is empty, it signals it via the corresponding entry in the *free_queue* array (and waits). Simultaneously, threads with non-empty queues check if there are signaled entries in the *free_queue*. When a thread notices someone's queue is empty, it transfers 4 nodes to the *shared_nodes* vector. When the waiting thread notices there are nodes in *shared_nodes*, it will push one into its queue. This ensures it will do more processing.

It's important to note that our load-balancing doesn't ensure a perfect division of work between the threads, it just ensures there aren't threads without work while others still have nodes to process. If we were to achieve that, we would need more synchronization, and we concluded that this was the best tradeoff achieved.

5 Performance results (vs expected)

A parallel application's goal is to obtain a speedup that grows linearly with the number of threads. For example, with 2 threads, we're targeting a speedup of 2, with 4 threads, 4, etc. However, performance is limited by (1) the number of cores, as it's not the number of threads that translates to computational power; (2) the overhead introduced by synchronization; and (3) the impossibility to achieve perfect load balancing (without using too much synchronization).

We noticed (1) when we tested with $n\_threads = 12$ (on a 6-core machine), since in some tests performance became worse. The machine supports hyper-threading, so in theory we would be able to have 2 threads running per core. However, since they're still competing over the same cache and the same memory, this may lead to worse results (which it did, in some cases).

We noticed (2) when we tested with $n\_threads \geq 4$: when there are more threads, it's more possible that they will compete for the same memory positions, and thus more possible that there will be idle threads. In fact, Amdahl's Law tells us that speedup will be limited by the fraction of the problem that's not parallelizable.

In summary, although the obtained speedup isn't the theoretically optimal for a shared-memory system, we obtained very satisfactory results which were only harmed by aspects we cannot avoid (for this particular problem).

All tests were performed on a machine with the following specifications:

- Intel Core i5-10500 3.10GHz (Hexa-core)
- 16GB RAM

#threads	ex1	ex2	gen10	gen15	gen19	gen20	gen22	gen24	gen26	gen30
1	0.0	0.0	0.0	0.1	1.5	79.4	111.5	52.8	55.1	187.9
2	0.0	0.0	0.0	0.1	0.9	49.3	57.1	27.5	30.7	100.6
4	0.0	0.0	0.0	0.0	0.6	22.6	38.9	24.8	19.1	94.1
6	0.0	0.0	0.0	0.0	0.6	21.7	31.3	20.3	15	93.9
8	0.0	0.0	0.0	0.0	0.5	17.9	25.5	19.7	13.6	59.6
12	0.0	0.0	0.0	0.1	0.5	15.9	37.2	12.5	16	46.6

Table 1: Execution times when running parallel version on input data (for 1, 2, 4, 6, 8, and 12 parallel threads).

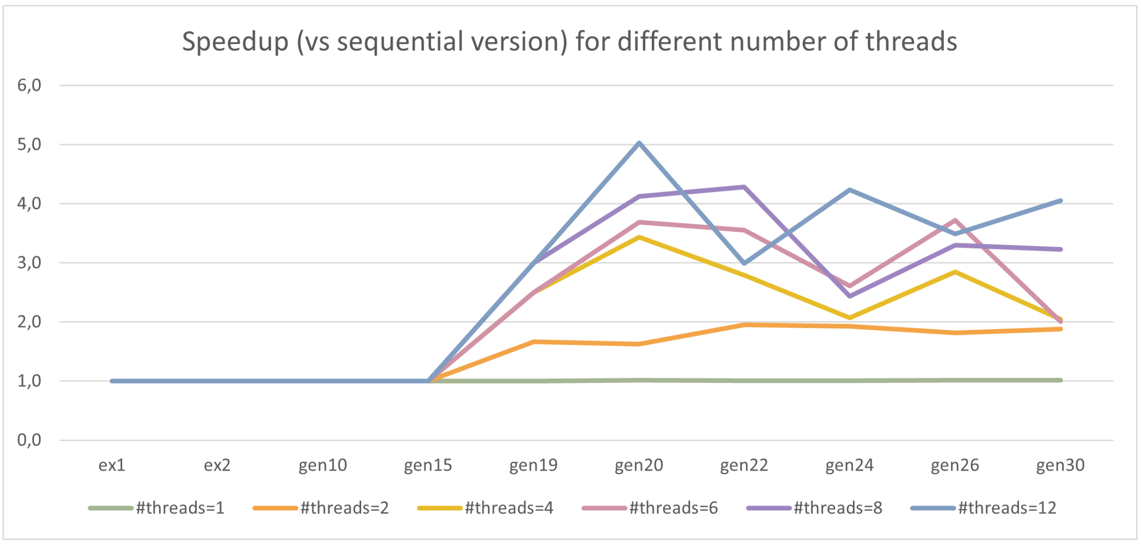

Figure 1: Speedup obtained for each test, with a varying number of threads.

References

S Tschoke, R Lubling, and Burkhard Monien. Solving the traveling salesman problem with a distributed branch-and-bound algorithm on a 1024 processor network. In *Proceedings of 9th International Parallel Processing Symposium*, pages 182–189. IEEE, 1995.