Instituto Superior Técnico

2022/2023 - Third Quarter

Parallel and Distributed Computing

OpenMP Implementation

1. **No: 93178 Name: Rui Ferreira**
2. **No: 105348 Name: Hubert Budny**
3. **No: 107985 Name: Maria Ramacciotti**

March 24, 2023

1 Introduction

OpenMP is one of the most used technologies for parallelization. This is due to the fact the it extends an existing programming language. Giving it little learning overhead for the programmer and the possibility of using already established and popular compilers.

In this project we will attempt to parallelize the implementation of the traveling salesperson problem using OpenMP. In the report we will focus on the approach used, the decomposition of the problem, our synchronization concerns and how the load balancing was addressed. After discussing the implementation we will analyse the results.

2 Implementation

2.1 Parallelization approach

Our approach involves breaking down a large problem into smaller, more manageable sub-problems that can be solved independently. These sub-problems can then be solved concurrently by multiple threads, resulting in faster overall computation time.

With this mindset, instead of parallelizing the operations executed in each node, we decided to split the tree in sub-trees so that each task can operate individually. With this approach we have to keep in mind the load balancing so one thread doesn't finish the work before the others and sits idle too much time while there are still nodes to be analysed. We also had to keep in mind that all of the tasks should be aware of the best overall results so the program doesn't keep running when the best result is already found.

2.2 Decomposition

As explained before the tree search was parallelized by splitting the tree in smaller trees so that each task can run without much need for inter-task communication. To do this, we decide to start the algorithm initializing a set of initial nodes for each thread, and placing them in their respective priority queues. Each thread then repeatedly pops a node from its priority queue and explores its child nodes. Two main concerns were raised. The first one is how can the private queues get more nodes when they finish their queues and the second concern is how can the tasks know when there is still work to be done or leave the loop.

To solve the first concern we decided that each thread should run a fixed number of iterations of the main loop on its private queue. The number of iterations was chosen by trial and error and we arrived at 50 iterations. This ensures that each thread does a good amount of work before trying to balance the queues and that, in case one of the queues is emptied, the threads don't stay idle for to long. After all threads run for the chosen set of iterations, they synchronize. After that, if one queue does not have enough elements to run another set of iterations, the thread gets new elements from the other queues.

To solve the second concern we changed the while loop condition for a public flag. At the beginning of the loop the threads change that flag to true. At the end of the while loop, inside a reduction section, each thread checks if their queue is empty. If it is not the threads change the flag to false. The reduction section has the operator &, making sure that if at least one of queues still has elements the flag will have the value false.

2.3 Synchronization concerns

One of the main synchronization concerns in this project is ensuring that the priority queue is accessed safely by multiple threads. We know that in the first part of the loop, where the threads run the algorithm for a set of iterations, there will not be any problem accessing the individual queues. Because of this, that part is inside a omp for that ensures that all threads are done computing before trying to balance the load.

In the load balancing section, we need to add a critical section to make sure that no threads pop and push the queues at the same time, creating a race condition.

The other critical section added was to when we update the best tour and best tour cost so that no thread reads or writes values in those variables at the same time.

At the end of the while loop we added a omp barrier to make sure that all threads had checked their queues and the reduction of the flag was completed. Allowing for all threads to continue or leave the loop correctly.

2.4 Load balancing

Load balancing is an important consideration as some of the threads may receive a smaller amount of work and finish before others leading to a thread being idle while the others are still working. Another consideration we needed to have was that when the first element of the queue is bad the queue is emptied, leading to an idle thread. To solve this we divided the work in sets of fixed iterations as explained above. When the tasks finish those iterations they check if their queue has enough elements for a new loop of iterations. If they don't, they steal elements from the other queues. To do this a thread checks every other queue and, if that queue has enough elements, they steal some elements to their own queue.

We decided to go with this approach, because it eliminates the necessity for accessing a shared queue which would need to be in a critical section slowing the program significantly. Our solution also ensures that queues are almost never empty without the need for any more data structures besides the thread queues.

3 Results

In Table 1 it's possible to see the execution times for the files provided for testing and the average speedup for 1, 2, 4 and 6 threads. The program was also tested for the files gen10 and gen15 but, since the number of cities is too small, the overhead of adding threads takes a disproportional amount of time.

The results are as expected and the speedup increases as the number of threads increase. It's also noticeable that the smallest files have lesser speedup since, as mentioned above, the overhead added with the parallelization is disproportional to the amount of time gained in the computation. It's also possible to see that gen 24 does not follow the trend. This happens because the load balancing is not the most efficient for the organization of this map. Making it so the program spend to much time passing elements from one queue to another.

Instance	1 thread	2 threads	4 threads	6 threads
gen 19	1.42	0.85	0.70	0.58
gen 20	40.6	27.9	24.87	17.13
gen 22	76.01	56.72	39.25	35.73
gen 24	21.95	16.37	17.61	12.09
gen 26	39.56	24.80	19.12	15.11
gen 30	123.35	86.88	62.64	45.37
Average Speedup	1	1.5	1.8	2.4

Table 1: results for the instances provided and average speedup for 1, 2, 4 and 6 threads

4 Conclusion

The parallelization approach used in this project is based on the divide-and conquer algorithm. Because each nodes can find his best solution on his path, we can divide the problem in other sub-problem calculated simultaneity, and this improves the execution time compared to the sequential version. In this way, the main concerns are the synchronization and the load balancing. For the first, we decided to use critical section and barrier to make sure that the threads work cooperatively to find the best solution. For the second, our approach was not the most efficient, since only one task can run at a time and there is a large amount of push and pops that use a big amount of time.

The performace results show that the speedup achieved through parallelization can be particularly beneficial for large-scale instances, where the sequential version would require excessive computation time to find the solution. However these results could be improved using a more efficient load balancing method.