Architectures and Platforms for Artificial Intelligence

Module 1

Project Work

Barnabé Sellier - barnabe.sellier@studio.unibo.it

Introduction

In this report I will describe the implementation of Bellman-Ford algorithm that works on parallel architecture using OpenMP.
The Bellman-Ford algorithm is used to compute the minimal distance in a graph from a root vertice to all other vertices in the graph. It is an amelioration of the Dijkstra as it also works with negative-valued edges. In an oriented graph negative cycles can be present and the Bellman-Ford is not adapted to such cycles and produces false results when applied on it.
Then we also have to detect such patterns. As it computes each distance independently, it is very suited for a parallelization.

Explanation of the algorithm

Given a root vertice, we want to compute the minimal distance from it to all the others ones. The algorithm assumes that the shortest past between a node and another can be computed within at most N - 1 iterations, N being the number of vertice inside the graph. It guarantees that the algorithm has explored all possible paths of length up to N-1 which is the maximum possible length of a shortest path in a graph with N nodes. Each iteration is based on the principle of relaxation. During relaxation, the algorithm examines each edge of the graph and checks whether there's a shorter path to the destination vertex through the source vertex. If it finds such a path, it updates the tentative distance to the destination vertex with the new, shorter distance and updates the predecessor of the destination vertex accordingly
First we initialize all the distances to infinite, then we iterate N - 1 times to evaluate the distance from the root node to all other nodes using relaxation. Then we iterate again over the graph to check the existence of a negative cycle. This works by checking if after N - 1 iterations, then it can still be a shorter path than the one found.

Implementation of the algorithm

As we have seen before, several parts of the code use loops that we could parallelize. First we have to calculate which nodes go to which threads to calculate. We use two arrays of size available threads, *local_start* and *local_end* that indicate what nodes each thread has to process. We parallelize this part with the instruction *#pragma omp parallel for*.
Once each thread knows which nodes they treat we can proceed to the initialization of the distance array, as we have to do it for each node we once again use a ‘“for parallelization”.

Then we proceed to the computation of the shortest distance within the N - 1 iterations. To do that we use the instruction *#pragma omp parallel* and we use a variable called *my_rank* that stores the number of the active thread thanks to the function omp_get_thread_num(). For each iteration, for each node assigned to the active thread we check if the path would be shorter by passing through a parent node. We test each node as a parent and this is the part of the process that we parallelize. Then, when an iteration has ended we wait each thread with a barrier, and we check on a single one if a change has been made in the last iteration. If so we continue, otherwise we break the loop.

When every iteration has ended, we can then check for the existence of a negative cycle. I recall that we detect it by doing one more iteration and if a change has been made, it means that a negative cycle exists in the graph. Thus, we have a variable *has_change* that we will reduce with parallelizing by using the instruction *#pragma omp parallel for reduction(|:has_change)*. We use the | (or) operator so if a change is made by one thread, *has_change* will have the value *true*.

Then, if there is no negative cycle we output the results of the algorithm otherwise we output that the graph contains one.

Experimental Setup and Result

In order to test my algorithm, I use 5 different inputs with 2, 4, 8 and 16 threads. The fourth input has a negative cycle, and the fifth is a very large graph in order to test several possible cases. The results are shown in the following table. We can see that the more there are nodes, the more it takes advantage of parallelization, except for very large numbers. We can also see that increasing the number of threads doesn’t necessarily increase the speed of the algorithm.

Table 1: Time elapsed during Bellman-Ford algorithm

Inputs \ N° Threads	4 nodes	5 nodes	10 nodes	4 nodes & negative cycle	1500 nodes
2	0.000116	0.000072	0.000080	0.000107	0.094516
4	0.000273	0.000283	0.000212	0.000496	0.099431
8	0.000527	0.000391	0.000467	0.000510	0.124738
16	0.001070	0.000652	0.000707	0.001045	0.142142

Conclusion

In conclusion, we have seen that the Bellman-Ford algorithm can take advantage of parallelization and we implemented the algorithm in C with OpenMP. We tested it on several inputs and with different numbers of threads and we have seen that a larger graph can take more advantage of parallelization than smaller ones.