Branch and Bound for TSP Problem
Project Report

Angelo Delli Santi
Giulia Grasso

April 2020

1 Introduction

The Branch and Bound is a general technique for solving optimization problems with a finite field of solutions. It is based on the decomposition of the original problem in smaller sub-problems, which are easier to solve. The state-space created by this algorithm could be represented as a binary tree or an nth-tree. The criteria for branching are given by some constraints, which are called branch's constraints that, in this project, are given by the computation of the Lower-Bound.
In order to explore the tree, there are three main strategies that could be used:

1. **Depth Firs (DFS):** in which the exploration starts from the root and goes down from father to child, once the exploration reaches the bottom it backtrack;

2. **Breadth First (BFS):** where first it is examined the father and then all its children, so the exploration goes down the tree when each level is completely explored;

3. **Best Bound First (BBF):** here the criteria are given by the best bound.

In this project, the Branch and Bound Algorithm was used for solving the TSP problem, i.e. the Travelling Salesman Problem, which is the problem of finding the shortest possible route in order to visit each city and go back to the starting one, given a list of cities and the distances from each pair cities.
For this project, it was implemented a sequential version of this algorithm in which it was chosen, as a method for exploring, a mixture between DFS and BBF,

by prioritizing the branch with the Lower-Bound during the DFS, to avoid the overhead of storing older states, and also a binary search tree in order to get a tree which is easier to read and in which the construction of the constraints gives the opportunity of reducing the size of the tree.
In this project, the value of the Lower-Bound is fundamental for the fathoming step, which is the step where it is decided if it is optimal continue the exploration of that precise node or not, in fact when the lower-bound of the current state is worst than the current best solution found, the algorithm prune that branch of the tree. To compute the lower-bound, it was used a method proposed by Beardwood, Halton and Hammersley (1959) [2] that consist in computing it as

$$LowerBound \ = \ \frac{1}{2}\sum_{u} edgeMin(u) \ + \ secondEdgeMin(u)$$

2 Tool used

In this project, C++ is used as programming language for implementing the code and MPI library for making the parallel version of the code and also for managing the communications through processors. Moreover, it was implemented a script in C++ in order to generate test-cases, and it was also used Google or-tool (which is a python library) for comparisons and debugging the solution of the implemented algorithm.

3 Parallel algorithm

The algorithm that is used for the parallel version utilizes three communication layers in order to distribute and SpeedUp the workload under several processors. This algorithm is implemented in such a way that it can be built on top of almost any sequential algorithm for Branch and Bound. This was done to allow us to focus on the parallel implementation, knowing that future improvement in the sequential version would be easily integrated into the parallel algorithm.

3.1 Static Distribution

The first level of parallelization consists of an initial static distribution of the workload. Each processor will take advantage of its rank to start from a different sub-problem; in particular, thanks to the fact that we are using a binary Branch and Bound, each processor will decompose its rank into a binary representation and will use each bit of it to decide autonomously if take the left branch or the right branch. This will ensure that each processor will start from a different sub-problem, without any needs of communications. (Figure 1)

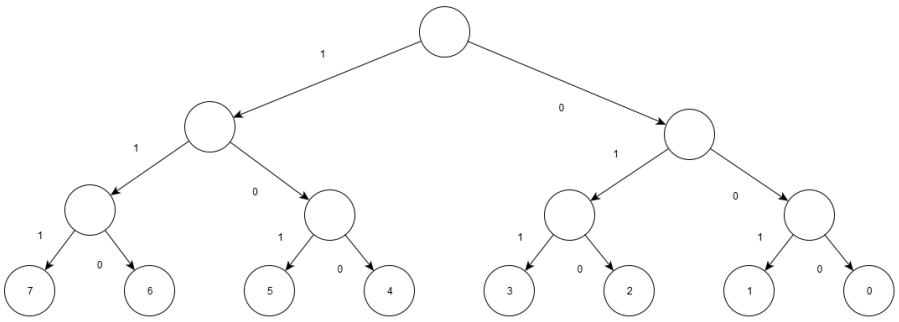

Figure 1: Each processor will use the binary representation of his rank to start from a fixed position in the branch and bound tree. For instance the processor with rank five (101) will follow the branch left (1), right (0) and left(1) to end up in its position.

Although this solution is simple, on its own is not effective enough. In fact, it will not take into account, because it is not possible to know a priori, how deep each branch will be, and therefore, the workload could result unbalanced. Also, because each processor will work with a sub-tree, they may not have knowledge about some optimal solution that other processors found during their exploration, and this lack of information may result in less pruning and slower execution time. For this reason, two more layers of communication will be used.

3.2 Increase the chance of pruning - Communication of the optimal solution

To allow each processor to maximise the number of branch pruned, It will be necessary to make all of them communicate with each other when a new best solution is found. To do so, non-blocking communications are exploited; each processor will open a communication channel (MPI_Irecv) in order to be ready to receive a new solution from any other processor. When a processor finds a new solution, which is better than all those previously found, it sends (MPI_Isend) the cost of the new solution to all the other processors. Each processor will receive it and will check if it is better than its current best solution to avoid situations of data-race. If a processor receives a solution cost which has a lower value than the cost of its best solution, it will update the best cost and discard its solution.

3.3 Dynamic work-load balancing

Thanks to the communication of optimal solutions, some of the processors may receive a cost which is much better than all (or most of) the Lower-Bound of its sub-tree. This could result in some processors to finish much earlier than the other. Besides, from experimental evidence, this phenomenon occurs very often, and at the end only the processor that starts from the "lucky sub-tree" will do most of the work. Therefore a new communication layer will be used. Each processor will have to know the state (work/idle) of all the other processors; to do so, each processor opens a new communication channel that will be used to receive the state information of the other processors. When a processor ends its sub-tree, it will enter in the idle state. In this state, it will send a message to all the other processors to inform them that it is free; then, it will open a new communication channel to receive a new sub-tree to work with.
On the other side, when a processor detects that the processor at its left (with respect to the rank enumeration) is free, it will send half of its sub-tree to it. The reason why only the processor at the right of an idle processor is allowed to send to it new work is to avoid situations where multiple processors send problems to the same receiver, and this would require more work to coordinate them in order to not lose sub-problems due to data race. Once the idle processor receives a new problem to work with, it will send its new status to all the other processors and will start to explore that sub-tree.
Once all processor are in idle, the program can terminate.

4 Implementation details

Several different types of communication have to coexist in this algorithm, and each processor does not know who will send the message.
Besides, processors have to continue their work even if they could receive some messages. Therefore non-blocking communications are used.
To make the distinction between different type of messages simpler, TAGS are used to distinguish communication, in particular, TAG=0 is used to communicate a new best solution, TAG = 1 is used to communicate processor status (working/idle/exiting), TAG=2 is used to communicate sub-problems.

```
M //initial Problem
checkWorkState //it contains status of all the processors
M_sub = TspSplitProblemGivenRank(M,rank);

 //Initialize communications
Irecv(&bestTmp, ANY_SOURCE, TAG=0, &requestCostIn);
Irecv(&tmpStatus, ANY_SOURCE, TAG=1, &requestInfoIN);
Tsp(M_sub); //Solve TSP problem for sub-problem M.
sendStatusToAll(1); //Send to all idle status


Irecv(M_sub, ANY_SOURCE, TAG=2, &requestProblemIn)
while(1){
    Test(&requestProblemIn);
    if (WorkReceived) {
        sendStatusToAll(0); //Status busy
        Tsp(M_sub);
        sendStatusInfoToAll(1);
        Irecv(M_sub, ANY_SOURCE, TAG=2, &requestProblemIn)
    }

    Test(&requestInfoIn);
    if (InfoReceived) {
        checkWorkState[statusInfoIn.SOURCE] = tmpStatus;
        Irecv(tmpStatus, ANY_SOURCE, TAG=1, &requestInfoIn);
        //Open new input channel
    }
    if (workState()){ //check if all processor are idle.
        sendStatusInfoToAll(2);
        // extra status to inform that processor is exiting.
        return;
    }
}
```

Although a broadcast would have been more efficient, the MPI implementation requires to know which processor will be the sender. In our case, we do not know who will be the processor that will need to communicate first; therefore, we use a simple MPI_Isend / MPI_Irecv to broadcast information with MPI_ANY_SOURCE.

```
void sendStatusToAll(int status) {
    for (int i = 0; i < size; i++) {
        if (i != p) {
            Isend(status, i, TAG=1, &requestInfo);
        }
    }
}
```

Finally, the TSP algorithm is modified in order to send the best solution when it found a new optimal one.

```
void TSP(M_sub){
    Test(&requestCostIn);
    if (CostReceived ) {
      if (bestTmp < bestCost) {
        bestCost = bestTmp;
        bestIsMine = 0;
      }

      Irecv(bestTmp, ANY_SOURCE, TAG=0, &requestCostIn);
      //p open a new communication channel, to wait for new sol
    }

  if (checkEndState(M, V) == 1) {  //END STATE
      int tmp_cost = checkCost(M, C, V);

      if (tmp_cost < bestCost && checkFinalSol(M, V)) {
      //check if finalSol is correct
        bestCost = tmp_cost;
        saveBestSol(Bestsolution, M, V);
        bestIsMine=1;
         for(int i=0;i<size;i++){
            if(i!=p){
             Isend(bestCost, i, TAG=0, &requestCostOut[i]));
            }
         }
      }
    }
       return;
}
```

5 Complexity analysis

5.1 General Algorithm

Thinking about the worst-case analysis of Branch and Bound, the algorithm should explore each node of the state space; hence the complexity is the size of the space case. However, if the Lower-Bound cost function used is exact, the complexity is linear in the length of the path from the starting node to the goal node. Unfortunately, these two cases are really rare; thus, they do not give much information about the actual performance of the algorithm. As a consequence, average-case complexity gives a more realistic performance. The aim of an average-case analysis is to search for the relationship between the average-case complexity and the accuracy Lower-Bound cost functions used.
Let p_o be the probability that an edge cost is zero and let b be the mean branching factor. Then, bp_o is the expected number of zero-cost branches leaving a node, or the expected number of children of a node that have the same cost as their parent. We call these nodes same-cost children. It turns out that the expected number of same-cost children of a node determines the expected cost of optimal goal nodes. Intuitively, when $bp_o > 1$, a node should expect to have at least one same-cost child, so that the optimal goal cost should not increase with depth. On the other hand, when $bp_o < 1$, most nodes should not have a same-cost child, which causes the optimal goal cost to increase with depth.
It has been shown, in the article *Performance of linear-space search algorithms* [9] in lemma 2.8, that the expected number of nodes expanded by any algorithm that is guaranteed to find an optimal solution is exponential in n, the number of nodes, when $bp_o < 1$, and the expected number of nodes visited is quadratic in n when $bp_o = 1$, and linear in d when $bp_o > 1$.
Practically p_0 can also be considered the probability that edges take the minimum cost, given that this cost is known a priori.

As previously described, our algorithm makes use of a binary Branch and Bound. Thus, the branch factor is $b = 2$; if we consider the distribution of weights of the graph normally distribute as $\mathcal{U} = \{1, maxW\}$, the probability p_0 can be easily computed as $\frac{1}{maxW}$.
Hence, we are in the case where $bp_0 < 1$, therefore we can expect that the average time complexity of our algorithm will be exponential.

5.2 Parallel algorithm complexity

As explained in section 3, the parallel algorithm will try to distribute the workload between P processors. It means that the number of states to visit will be at most $\frac{2^V}{P}$ where V is the number of edges. The communication time has also to be taken into account. In the worst-case scenario, for each state explored, processors will have to communicate to share: sub-problems, new solutions, and new states of processors.
To share these data, it is required a direct communication between two processors; this will require a linear number of communications equal to P, where P is the number of processors. A possible improvement could be made by making use of recursive doubling broadcasting to SpeedUp this part of the communication.
We can define t_d as the time required to send a word of data; a sub-problem is represented as a matrix NxN where N is the number of nodes. Therefore the worst time complexity can be express as

$$Tp = \frac{2^V}{P} \cdot ((t_s + N^2 t_d) + 2P(t_s + t_d)$$

5.3 SpeedUp

The SpeedUp is defined as $\frac{T^*}{T_P}$ where T^* is the best sequential algorithm.
Even though our implementation of the sequential algorithm is far from the "best", the strength of most of the best algorithms make use of linear programming or statistical methods as heuristics to find more precise lower bounds.[5] [6] [7]
These features can be transposed in both our sequential and parallel implementation without any overhead; for this reason, we think that the SpeedUp estimated both analytically and empirically can be meaningful, if the problem is compared with a sub-optimal implementation of the sequential branch and bound.
Considering that, the worst case of the sequential version of the algorithm is $O(2^V)$

$$S_p = \frac{T^*}{T_P} = \frac{P}{t_c}$$

where t_c is the communication time.
However, as explained before, for this type of problem, the worst-case scenario is very unlikely to happen and considered too pessimistic, while the SpeedUp may look better than the reality.

Since it cannot be predicted without the support of statistical tools, the pruning component of the Branch and Bound is completely neglected. Hence, also the SpeedUp anomalies described in section 5.4 are difficult to be considered.
For these reasons, normally, the quality of Branch and Bound algorithms are evaluated with empirical performance tests [4] [1] [3] that will be discussed in section 6.

5.4 SpeedUp anomalies

When p processors are operating independently on separate downward path, it may be that one processor very quickly finds a feasible solutions. In that case, also thanks to the pruning, it is possible for the parallel SpeedUp to be greater than p (super-linear speedup). This is known as an *acceleration anomaly*. Conversely, a feasible solution may be in a position in the tree where it cannot be reached in 1/p of the time that it would be reached with a single processor. Then the SpeedUp becomes less than p. This is known as a *deceleration anomaly*. [8], [4]

6 Experimental results

6.1 Experimental setup

To test the correctness and the performance of the algorithm, several test cases have been generated and solved with another tool. In particular, 100 test has been generated for each size from 4 to 25 (cities). An ulterior script has been implemented in order to automatize the testing; it was used to automatically run the program with all these test cases.

6.2 Analysis of Outcome

The first comparison was made by running the sequential version and the parallel version against all the tests, to see the average time elapsed (Figure 1).

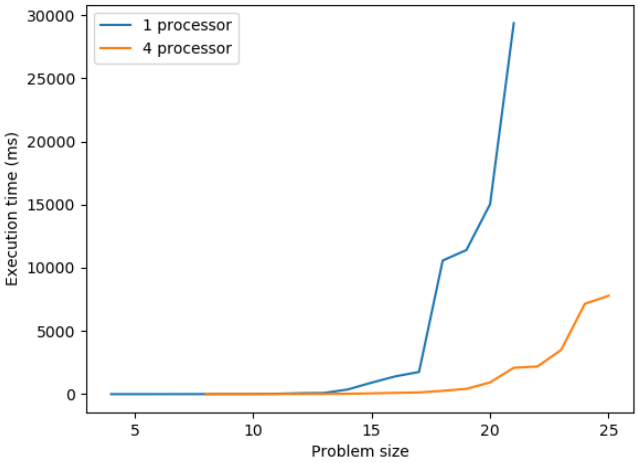

Figure 2: Average execution time

Problem size	Sequential version	Parallel version	Cost	Efficiency
8	5.27	1.52	6.08	0.87
9	9.65	2.53	10.12	0.95
10	14.70	3.42	13.68	1.07
11	29.22	6.30	25.20	1.16
12	72.68	10.30	41.20	1.76
13	96.07	14.48	57.92	1.66
14	374.54	31.91	127.64	2.93
15	912.29	64.31	257.24	3.55
16	1408.99	101.22	404.88	3.48
17	1755.02	135.84	543.36	3.23
18	10579.41	262.54	1050.16	10.07
19	11409.05	418.95	1675.80	6.81
20	15041.15	931.09	3724.36	4.04
21	29370.24	2088.01	8352.04	3.52

Table 1: Execution time for sequential version and parallel version with 4 processors

It is very interesting to see that for problems large enough, the SpeedUp is much larger than P. This phenomenon is the speedup anomaly described in section 5.4. Further test has been made to see how well the algorithm scale as the number of

processor increases. This time, the test case was fixed, taking as one the most time demanding with 20 cities, and it was tested with different numbers of processors. (Fig. 1)

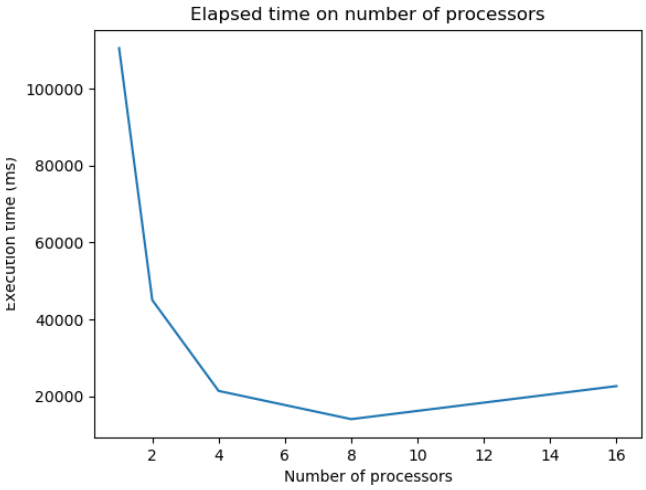

Figure 3: Average execution time versus the number of processors

Also in this case, the elapsed time decreases rapidly. Nevertheless, once it reaches 16 processors, it looks like the communication overhead becomes too heavy.

7 Summary and conclusion

Overall, the parallelization algorithm resulted in being quite successful, having an efficiency greater than 1 in a wide range of tests. Even though it does not seems to scale too well with a high number of processors, we think that some of these criticism given by the communication overhead could be mitigated, as suggested in section 5.2. It would be crucial to allow the algorithm to scale better for a larger number of processors. We could also test that efficiency gets better as the problem gets bigger. However, this was hard to be demonstrated for a number of processors greater than 8, because it would require a problem size too big to be feasible with the current performance limited by the quality of the Lower-Bound. However, as explained in section 5.3, the performance boosts that the parallel implementation provided to our sequential Branch and Bound could be easily applied to more powerful implementation of the sequential Branch and Bound.

References

[1] T. S. Abdelrahman and T. N. Mudge. Parallel branch and bound algorithms on hypercube multiprocessors. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications - Volume 2*, C3P, page 1492–1499, New York, NY, USA, 1989. Association for Computing Machinery.

[2] Jillian Beardwood, J. H. Halton, and J. M. Hammersley. The shortest path through many points. *Mathematical Proceedings of the Cambridge Philosophical Society*, 55(4):299–327, 1959.

[3] Raphael Finkel and Udi Manber. Dib—a distributed implementation of backtracking. *ACM Trans. Program. Lang. Syst.*, 9(2):235–256, March 1987.

[4] Ten-Hwang Lai and Sartaj Sahni. Anomalies in parallel branch-and-bound algorithms. *Commun. ACM*, 27(6):594–602, June 1984.

[5] A N Letchford and A Lodi. The traveling salesman problem: a book review. *4OR: A Quarterly Journal of Operations Research*, 5(4):315–317, dec 2007.

[6] Manfred W. Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33:60–100, 1991.

[7] Stefan Steinerberger. New bounds for the traveling salesman constant. 2013.

[8] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition)*. Prentice-Hall, Inc., USA, 2004.

[9] Weixiong Zhang and Richard E. Korf. Performance of linear-space search algorithms. *Artificial Intelligence*, 79(2):241 – 292, 1995.