Parallelization of Genetic Algorithm for TSP

CSC 415 – Parallel Computing

Javier Vela Tambo

December 2021

https://github.com/javiervela/tsp-genalg-parallel

Contents

1 Introduction

The Traveling Salesman Problem (TSP) is, “given a list of cities, and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?” [8]. The TSP problem can be translated to finding a Hamiltonian cycle with the least weight in a complete weighted graph. TSP is considered a NP-hard problem.

To obtain an exact solution, some algorithms such as branch-and-bound or linear programming have been proposed. Although these methods are much more time expensive compared to algorithms based on heuristics and approximation approaches. For this project the family of Genetic Algorithms is explored to solve the TSP.

Genetic Algorithms [7] are inspired by the process of natural selection, “survival of the fittest.” This family of algorithms uses operators such as ‘selection,’ ‘crossover,’ and ‘mutation’ on the ‘gnomes‘ of each individual of a population which evolves into another one, obtaining fitter individuals, until reaching a solution.

For solving the TSP, a Genetic Algorithm has been implemented that will have a population of individuals (an array of non-repeating city indexes, the path; and the weight of this path, the fitness). The fittest individuals will be selected and mutated to create a predetermined amount of children that will form the next generation population. This procedure will be repeated a fixed amount of generations. The mutation of one gnome is accomplished by swapping two randomly chosen cities a random amount of times (with an upper limit to make similar gnomes).

2 Related Work

The C++ implementation of the Genetic Algorithm module has followed an implementation from an article [5] found online. Also, the C++ implementation of the library, to read TSPLIB problems, [1] has been obtained and customized from GitHub repository [2].

3 Parallelization

Genetic Algorithms are considered “embarrassingly parallel” [6] because each individual of the population in each generation can be modified independently (or almost) from others. Three different parallelization techniques have been considered for this project: distribution of the population of different nodes, distribution of the population evolution in different hardware threads, and GPU parallelization. GPU usage has been discarded due to the nature of the problem. Threading was not completed during the project duration.

The technologies used for implementing the two parallelization techniques are: OpenMP [4] for the threading and MPI [3] for the message passing between nodes.

A predetermined amount of nodes is allocated using MPI. Each of these nodes generates a different, random population of size $\frac{population\ size}{number\ nodes}$. Each node evolves its population for a predetermined series of generations and, if the synchronization flag is activated, the root node will gather all the different populations and distribute a new one formed with the best solutions found in the previous ones. This is repeated till the generation limit is reached, when the best solution of each node is reduced in the root. The node computing distribution and MPI communication is explained using Figure 1.

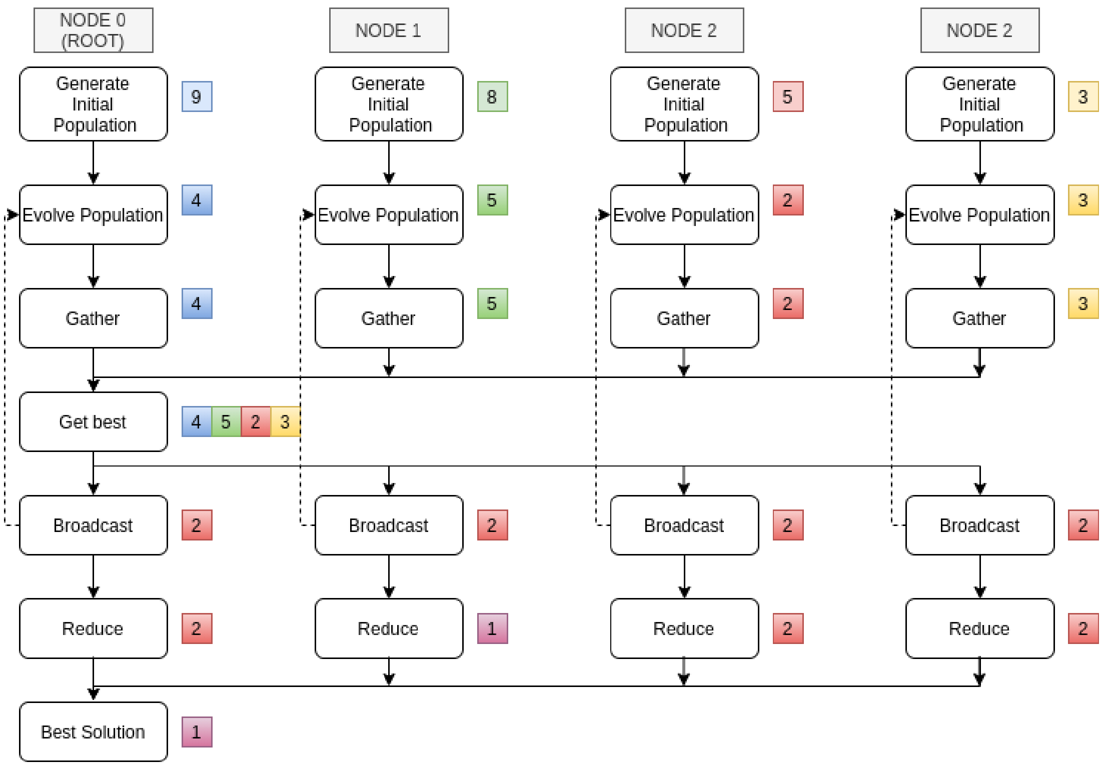


Figure 1: Node computing distribution and MPI communication diagram

The populations are serialized manually from a `std::vector` to two arrays of integers (`int`), the first one with all the fitness values of the population and the second one with all the sequences of cities in the gnomes. This serialized version is sent through the MPI interface.

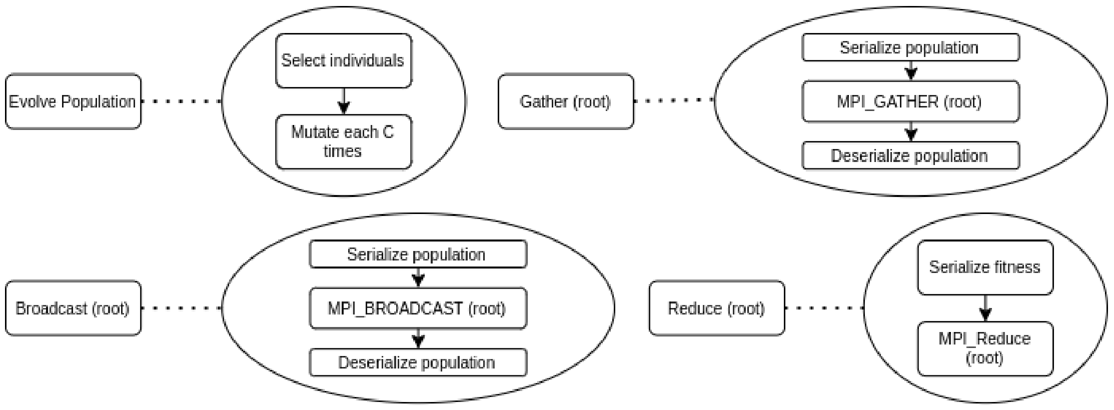


Figure 2: Definition of some Figure 1 terms

The mutation of each of the selected individuals is computed in a separate thread, allowing its parallelization. This calculation is completely independent of the others, so when each child of an individual is generated, it is added to a population private to the thread and later combined with the total population in a critical section. As noted, threading has not been completed in time, so it is not taken in account in the performance analysis.

4 Performance Analysis

The implementation debugging, and later performance analysis, has been carried out using the TSPLIB [1] problem and solution files. In this report the `gr202` TSP map, which represents a 202-city sub-problem of 666-city TSP, is used. Also `att48`, which represents 48 capitals of the US, has been used for more detailed debugging due to its low number of cities.

In the analysis, for the hyperparameters of the algorithm the following have been selected:

- Population size: 10000
- Child per gnome selected: 10
- Maximum number of mutations per gnome: 20
- Number of total generations: 10000
- Generations per batch (before synchronization and output): 50

Note: When tuning these parameters a better suboptimal solution could be achieved.

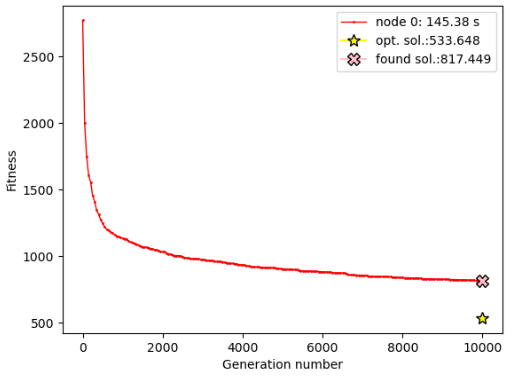
Figure 3 shows the plots generated with the outputs from the execution nodes. Each plot shows the evolution of the fittest (best) solution in each node's population. It also shows the execution time of each of the nodes. The time values from the plots are synthesized in Table 1 for the speedup analysis.

The expected speedup is $1 \times N$, N being the number of nodes involved in the execution. The speedup result is unexpected, achieving a $1.2 \times N$ speedup. The reason is that the bigger the population is in each node the more time takes to order it by fitness. When dividing the population, this ordering cost is reduced.

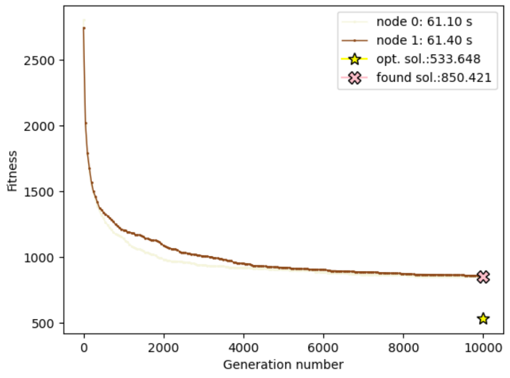
In Figure 4, the plots show the same executions with the synchronization activated between solutions in different nodes. The execution time increases slightly, but this allows for a better suboptimal solution to be achieved. The lines in the plots keep parallel due to the synchronization between populations each batch computed. Also, the execution times of the nodes are the same due to the synchronous communication by the MPI calls.

N	Avg. exec. time	Speedup	$\frac{Speedup}{N}$
1	145.38	1	1
2	61.25	2.37	1.18
4	26.42	5.50	1.37
10	11.58	12.54	1.25

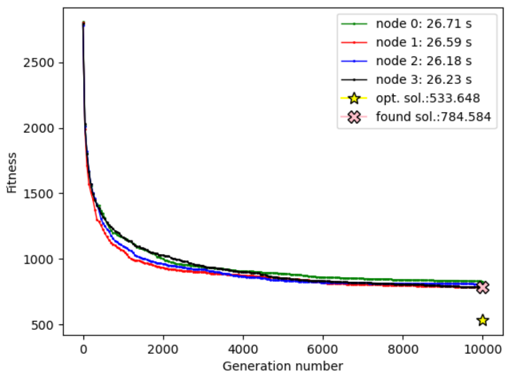
Table 1: Speedup analysis



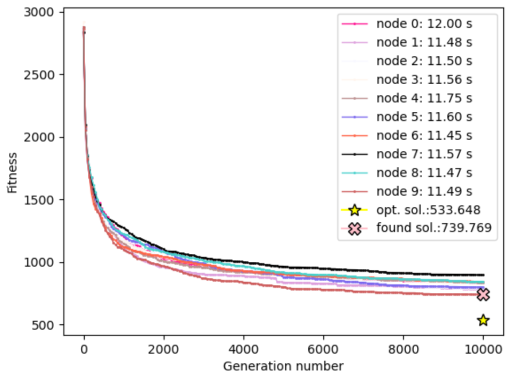
1 Node



2 Nodes



4 Nodes



10 Nodes

Figure 3: Speedup analysis increasing number of nodes

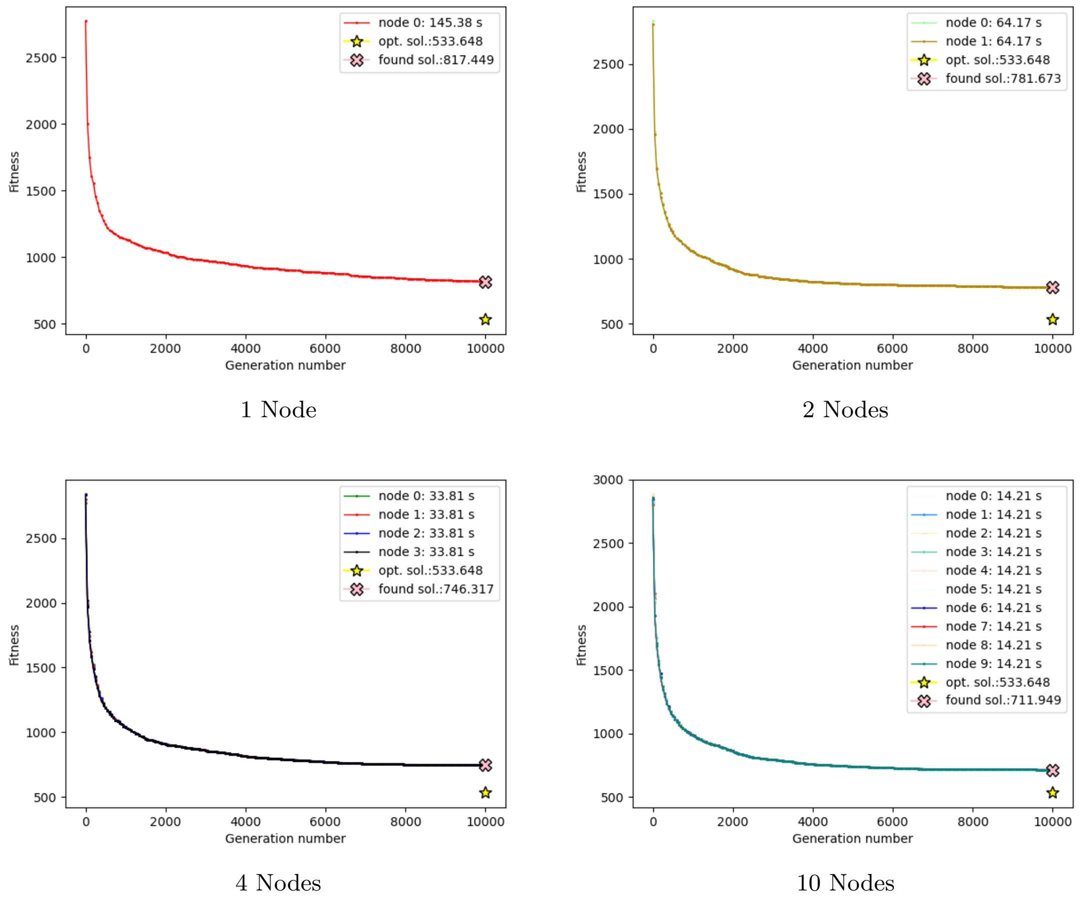


Figure 4: Execution with synchronization between nodes activated

5 Conclusion

Distributing the population in different nodes achieves a significant speedup because of the nature of the algorithm to parallelize. The implementation takes advantage of some capabilities of the MPI interfaces such as 'broadcast', 'gather', and 'reduce', letting the underlying library optimize the communications. Although, the need to serialize the populations was noted after the initial and base implementation, so a better storage and ordering of the individuals could have been developed to erase this need. Also, the ordering of the individuals by fitness could have been omitted using a priority queue. With these optimizations a faster serial implementation would have led to a faster parallel algorithm.

Although the synchronization between nodes to update the best individuals is useful to achieve a better suboptimal solution, this solution is not substantially better. The synchronization increases the execution time, but not enough to consider an algorithm without synchronization better.

The threading improvement has not been accomplished either due to the complication in the serial code, which shadows a basic problem with the OpenMP interface; due to an error in the allocation of resources or passing of environment values for threads; or due to a thread overhead much larger than the thread speedup. This situation has forced the implementation to revert to a previous version with only the MPI interface.

The genetic algorithm uses two basic operators for selection and mutation. For this reason, the suboptimal solution achieved is not comparable to the optimal. Also, the algorithm is parameterized to allow for tuning to achieve the best solution. The serial implementation is not the most sophisticated to allow for an easier parallelization and debugging. Although, once the development of the code is finished, other genetic solutions can be added to the algorithm, to favor a better suboptimal solution. Moreover, a fine-tuning of the hyperparameters of the current algorithm would boost finding a better solution.

References

[1] URL: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/.

[2] Annihil. *Annihil/Little-TSP-Solver: A C++ TSP solver using Little Algorithm (TSPLIB compatible).* URL: https://github.com/Annihil/Little-TSP-solver/.

[3] *MPI - Tutorials.* URL: https://mpitutorial.com/tutorials/.

[4] *OpenMP - Home.* Sept. 2021. URL: https://www.openmp.org/.

[5] *Traveling salesman problem using genetic algorithm.* July 2021. URL: https://www.geeksforgeeks.org/traveling-salesman-problem-using-genetic-algorithm/.

[6] Wikipedia. *Embarrassingly parallel — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=Embarrassingly%20parallel&oldid=1057796455. [Online; accessed 11-December-2021]. 2021.

[7] Wikipedia. *Genetic algorithm — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=Genetic%20algorithm&oldid=1051184365. 2021.

[8] Wikipedia. *Travelling salesman problem — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=Travelling%20salesman%20problem&oldid=1058458511. 2021.