Solving the Travelling Salesman Problem Using Genetic Algorithms and MPI

SOKRATIS BARTZIS

University of Crete, Heraklion
Department of Computer Science
sokratisbrtz@gmail.com
Phone number: (+30) 6978598746

ABSTRACT

The Travelling Salesman Problem (TSP) finds application in several fields, ranging from logistics to VLSI chip manufacturing. Due to its categorization in the NP-Hard class, the analytical solution of the problem fails to provide results in a reasonable timeframe for increased problem sizes. In this project, we apply a Genetic Algorithm to the problem, parallelize it using the Message Passing Interface (MPI) and test our implementation on a multicore cluster. The experimental results suggest significant speedup compared to the analytical solution, while sustaining a drop in the accuracy of the output computed best route.

Keywords: Travelling Salesman Problem − Genetic Algorithms − Message Passing Interface

1. INTRODUCTION

The Travelling Salesman Problem is described as traversing through a number of cities, once per city, and return to the origin city, while minimizing the distance travelled. Since the problem is categorized in the NP-Hard class, sequentially calculating the travelled distance for every permutation of the cities graph (analytical solution) is not feasible, in terms of execution time, for an increasing problem size. Parallelizing the analytical solution is also ineffective for large problem sizes due to the fact that the number of different permutations increases exponentially

with the number of cities. Thus, distributing the computation to hundreds or thousands of cores will not provide substantial speedup. Branch and bound Algorithms have also been an attempt at tackling this problem. In this project, we implement a parallel version of a genetic algorithm.

2. PROPOSED METHOD

A genetic algorithm uses the concepts of selection, mutation and crossover, found in nature, to solve the TSP. Starting with a small, randomized, population (subset of the total permutation space), a fitness calculation is performed for each individual of the population. Fitness quantifies the value of a cost function, which is specific to the problem at hand. Individuals that ranked higher in terms of fitness (selection) are then used to create a new generation, a new population based on the previous one, that, hopefully, is more fit than their parents' generation. This new generation can be created in two ways: perform random changes in a member's genome (mutation), or exchanging parts of the genome of two fit individuals in the population (crossover).

In the case of TSP, fit individuals are considered to be routes with short distance travelled. Mutating an individual corresponds to exchanging the order of visiting of two or more cities in a route. Finally, crossover between two routes is performed by keeping part of a route intact and filling the rest of the path with cities found in a different route, in the order of appearance and skipping the cities that are already found in the unchanged part.

The random nature of genetic algorithms makes it possible that the true shortest distance is never found during program execution. Thus, genetic algorithms do not consistently provide the optimal solution to the examined problem and such is the case for TSP.

3. METHODOLOGY

In order to achieve parallelism for the genetic algorithm implementation, the Message Passing Interface (MPI) was used. Each process is assigned a role, a singular master and multiple slaves. All processes create their own initial population of routes and, for each individual of the population, a fitness score is calculated. Using the three methods described above, selection, mutation and crossover, a new generation is created and the distance calculations are repeated.

For every individual examined, a comparison with the current best route is performed. In the case of the master process, if a new best has been found, the best route is updated with the new one. In the case of the slave processes, apart from the best route update, a message is sent to the master process with the newfound route. The master process then performs another comparison with its own best route and updates it if needed. On program termination, the master's best route is considered as solution.

The algorithm terminates if a number of generations has passed and a new best route is not found. An upper limit to the generation count could also be set, but it is not used in our implementation. In order to meet the termination condition faster in the parallel version, all slaves send a *Generation Increment* message to the master if a number of generation (less than the

termination condition count) have passed without a new best route. Increasing the number of processes should also tweak the value of those counters for better accuracy.

4. EXPERIMENTAL RESULTS

4.1 *Platform and settings*

For our evaluation, a problem size of 25 cities was used. The population size of each generation was set to 10.000 individuals and program termination occurs 2.000 generations after the last (total) best route was found. Slaves send a *Generation Increment* message every 100 generation of no new best route. Finally, the program was run on a 3-node dual-core cluster provided by ICS FORTH.

It is worth noting that the analytical solution to this problem is not shown in the figures since its execution with problem size 25 is beyond the capabilities of our system.

4.2 *Figures*

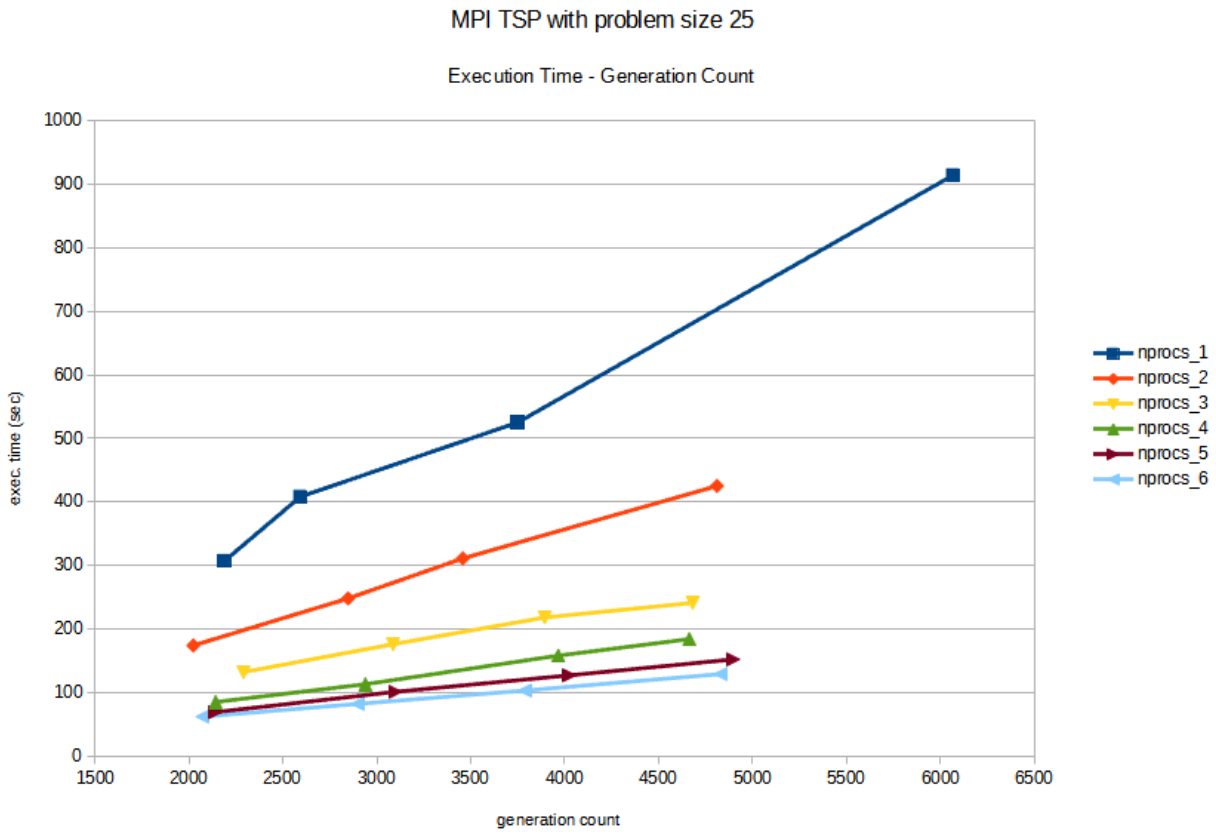

Figure 1. Execution time vs generation count of Parallel TSP run with 1 to 6 processes.

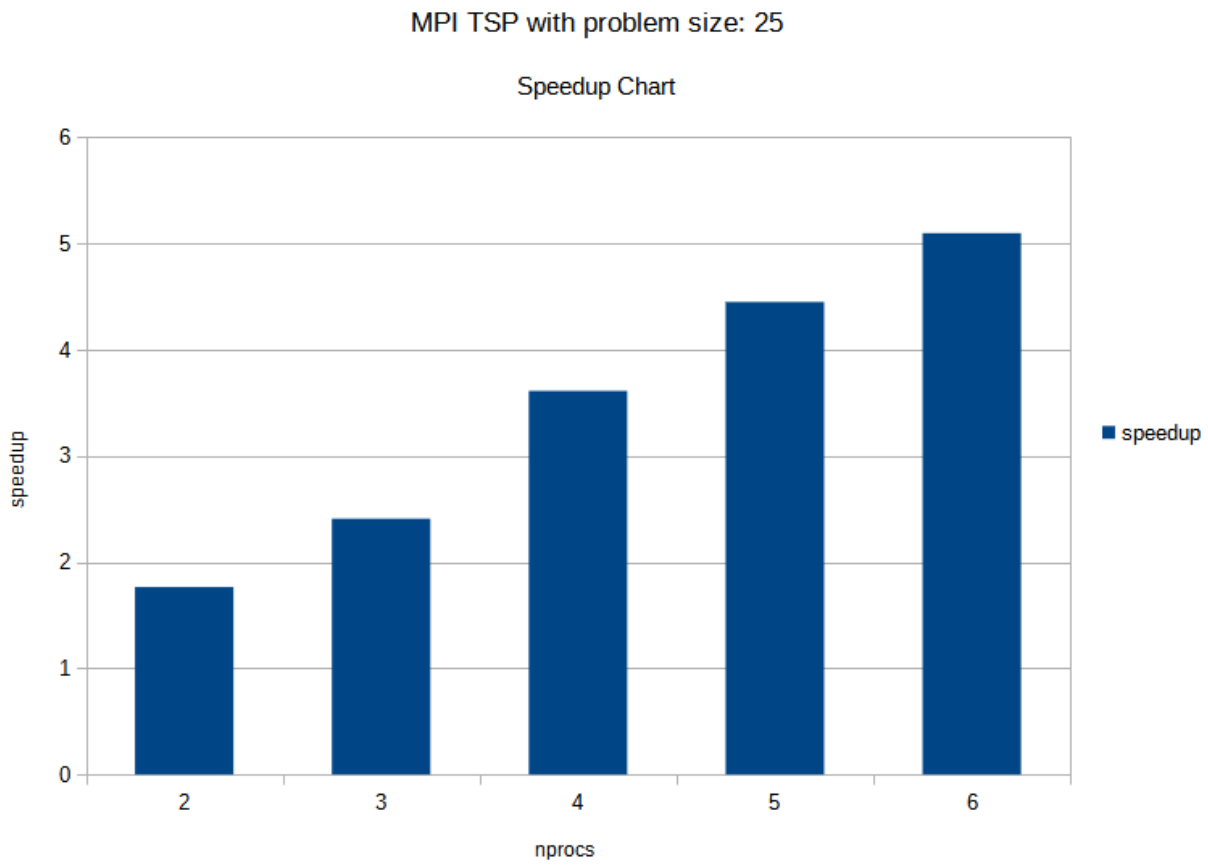

Figure 2. Speedup chart of different number of processes.

As *Figure 1* suggests, the execution time decreases with the addition of processes, but the speedup is not linear (*Figure 2*). Since the communication overhead in our implementation is low (new best routes are found more often at the start of the program and less often toward the middle/end of it, when the best route is already short enough), a possible explanation of *Figure 2* is the speedup calculation itself. To obtain a speedup value:

$$speedup = \frac{\text{exec\_time (nprocs=1)}}{\text{exec\_time (nprocs=N)}}, \text{ where } N \epsilon [2, 6].$$

This division is performed on executions of the same total generation count. While the generation count of a single process is accurate, for multiple processes the actual generation count is greater than *Figure 1* suggests. This is attributed to the fact that the program termination can be triggered by a *Generation Increment* message, without taking into consideration the generation counters of the processes that did not manage to send such a message. The greater the number of processes is, the greater the potential error in the total generations' calculation will be.

5. FUTURE WORK

To improve the evaluation of our study, the total generation count described in the previous section needs to change. A message from all the slaves to the master, just before program termination, containing their total generation count, as well as the summation of these counters by the master seems like a viable solution. As an extra evaluation metric, the accuracy of the given solution could be compared to a known solution, in order to verify if the calculated solution's distance is in fact close to the actual best.

6. CONCLUSIONS

In this study we have shown that the Travelling Salesman Problem can be solved within a reasonable timeframe, even for increased problem sizes. By selectively mutating a subset of permutations, the genetic algorithms provide a near optimal solution to TSP, while, with the use of MPI, we managed to achieve considerable speedup compared to the sequential genetic algorithm.