PARALLEL GENETIC ALGORITHMS TO SOLVE OPTIMIZATION PROBLEMS

Girdhar Gopal*

Rakesh Kumar**

Ishan Jawa***

Abstract

Solving the Optimization problem efficiently is an open challenge in computer science. Most of problems in this category are hard to be solved in less time. As the problem size increases the solution becomes worse. Genetic Algorithms are used in past to get a good solution for these problems effectively. Genetic Algorithms are parallel in nature at many steps. In this paper, this parallelism of genetic algorithm is exploited and genetic algorithm is coded in two aspects, simple genetic algorithm and parallel genetic algorithm. Parallel genetic algorithm performs better on a parallel hardware, where all of the processors become busy in execution of algorithm, which leads to quick solutions.

Keywords: Genetic Algorithms, Optimization, PGA, TSP

* **Assistant Professor**

** **Professor**

*** **Research Scholar**

DCSA, KUK, Haryana, India

I. INTRODUCTION

The famous computational problem Traveling Salesman Problem (TSP) is an optimization problem [1] [2]. It belongs to NP-complete class of problems. A basic explanation of TSP is as follows: A salesman with a map, including N cities and the distances between each pair of cities, aims to visit each city exactly once starting from a given city. Meanwhile, he has to find the shortest cycling path between these cities to complete his tour within a minimum time period. The problem ends up with N! Different possible cycles; therefore, the brute force algorithms are not feasible. One possible solution that is proposed is to use intelligent algorithms. In past many evolutionary algorithms are used to solve TSP like Tabu Search, Genetic Algorithm (GA), Ant Colony Optimization (ACO) etc. GA is a population based search consisting of five operators: Initialization, Selection, Crossover, Mutation and Replacement [1] [2] [3] [4]. Initialization is used to seed the initial population randomly. Selection is used to select the fittest from the population. Crossover is used to explore the search space. Mutation is used to remove the problems like genetic drift. Replacement is used to progress generation wise population. Simple GA is very much stochastic in nature when implemented. Simple GA is also parallel in nature as it operates on a set of population instead of a single point. So Parallel Genetic Algorithms are implemented to solve TSP in this paper. In Section II, the implementation details of SGA and PGA are discussed. The implementation results and comparison of PGA with SGA is described in Section III. Section IV represents the conclusions and future scope.

II. PROBLEM FORMULATION

Genetic algorithms are adaptive algorithms proposed by John Holland in 1975 [1] and were described as adaptive heuristic search algorithms [2] based on the evolutionary ideas of natural selection and natural genetics by David Goldberg. They are powerful optimization techniques that employ concepts of evolutionary biology to evolve optimal solutions to a given problem. Genetic algorithm works with a population of individuals represented by chromosomes [5] [6]. Each chromosome is evaluated by its fitness value as computed by the objective function of the problem. The population undergoes transformation using three primary genetic operators – selection, crossover and mutation which form new generation of population. This process continues to achieve the optimal solution. General structure of genetic algorithm is:

Procedure GA (fnx, n, r, m,ngen)

//*fnx* is fitness function to evaluate individuals in population

// n is the population size in each generation (say 100)

// r is fraction of population generated by crossover (say 0.85)

// m is the mutation rate (say 0.001)

//ngen is total number of generations

P := generate n individuals at random

nogen:=1 //denotes current generation

while nogen <=ngen do {

L:= Select(P,n,nogen)

S:= Crossover(L,n)

M:= Mutation(S,m)

P:= M

nogen++;

end proc

Parallel Genetic Algorithms are implemented using the parallel computing toolbox of MATLAB. Parallel Computing Toolbox™ software allows offloading work from one MATLAB session (the client) to other MATLAB sessions, called workers. One can use multiple workers to take advantage of parallel processing. A local worker can be used to keep MATLAB client session free for interactive work, or with MATLAB Distributed Computing Server one can take advantage of another computer's speed [7] [8] [9]. MATLAB Distributed Computing Server software allows you to run as many MATLAB workers on a remote cluster of computers as your licensing allows. MATLAB provides some solutions to as below to do the work in parallel.

Suppose the code includes a loop to create a sine wave and plot the waveform:

for i=1:1024

*A(i) = sin(i*2*pi/1024);*

end

To interactively run code that contains a parallel loop, firstly MATLAB pool must be opened. This reserves a collection of MATLAB worker sessions to run loop iterations. The MATLAB pool can consist of MATLAB sessions running on local machine or on a remote cluster:

matlabpool open local 3

With the MATLAB pool reserved, the code can be modified to run loop in parallel by using a parfor statement:

parfor i=1:1024

*A(i) = sin(i*2*pi/1024);*

end

The only difference in this loop is the keyword parfor instead of for. After the loop runs, the results look the same as those generated from the previous for-loop. Because the iterations run in parallel in other MATLAB sessions, each iteration must be completely independent of all other iterations. The worker calculating the value for A(100) might not be the same worker calculating A(500). There is no guarantee of sequence, so A(900) might be calculated before A(400). The only place where the values of all the elements of the array A are available is in the MATLAB client, after the data returns from the MATLAB workers and the loop completes. When you are finished with your code, close the MATLAB pool and release the workers:

matlabpool close

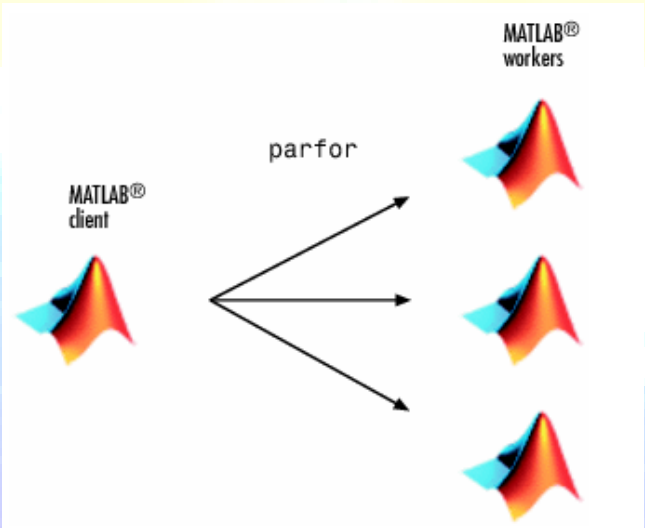

Figure 1: MATLAB parfor looping workers

In implementation of PGA this parfor is used instead of simple for loop to loop for various generations.

III. RESULTS AND DISCUSSION

This section will focus on experiment that uses the GA in two aspects, Simple GA and Parallel GA. The algorithms are coded in MATLAB 2014a with Intel Xeon Quad Core Processor. The performance of GA is tested at three TSP instances: the known optimal solution TSP instances taking from TSPLIB. The objective of the experiment is to investigate the performance of SGA and PGA in terms of number of generations and iteration time to come out with the optimal solution for TSP.

One of the main difficulties in building a practical GA is in choosing suitable values for parameters such as population size, probability of crossover (Pc), and probability of mutation (Pm). In this experiment, De Jong's guidelines, which is to start with a relatively high Pc (≥ 0.8), relatively low Pm (0.001-0.1), and a moderately sized population is used. The selections of parameter values are very depend on the problem

to be solved. Noted that the larger the population size, the longer computation time it takes [10-15]. In this experiment, the GA parameters were obtained from the screening experiment and trial run. For each experiment, the algorithms were run ten times and the lowest travelling distance is taken as a final result. For all experiments in this study, termination is performed when number of generation reached the maximum number of generation. The maximum number of generation is entered at runtime of program. The following parameters are used in this implementation:

- Population size (N): 500
- Number of generations (ngen) : 1000
- Selection method: Roulette Wheel Selection (RWS)
- Crossover Operator: Partial Matched Crossover (PMX) with probability 0.85.
- Mutation: Inversion with mutation probability 0.01.
- Replacement: Simple Replacement Strategy
- Termination criteria: Execution stops on reaching maximum number of generations

Genetic algorithm is implemented in PGA form to optimize tsp problem instances. Following are the two plots to show the CPU positions when the program is in execution.

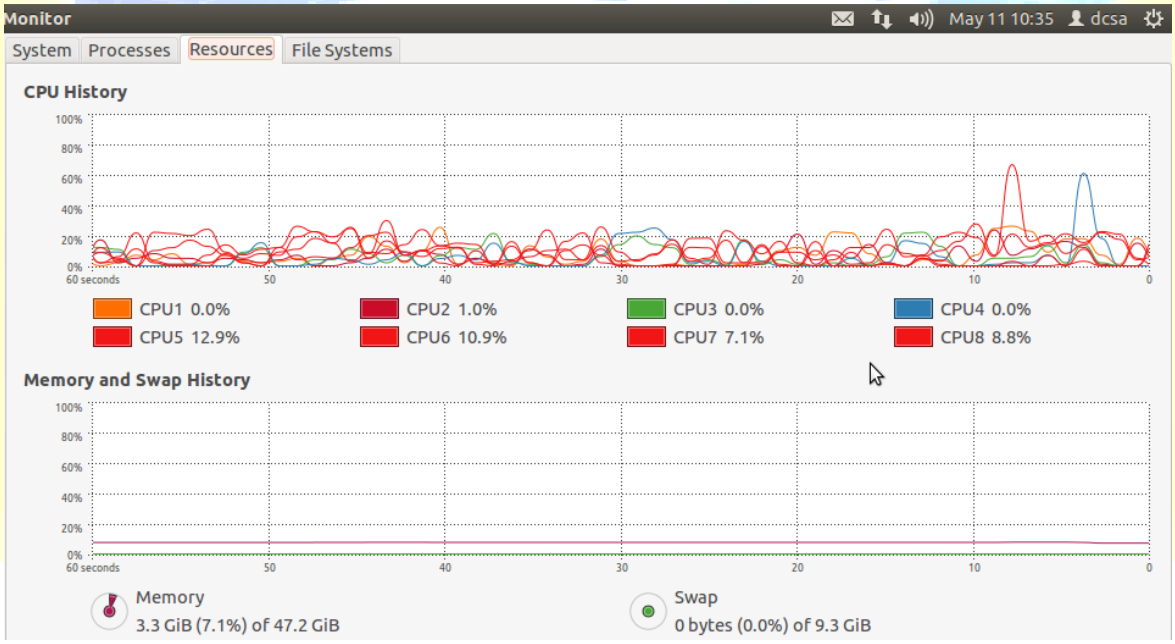

Figure 2: Simple GA in execution

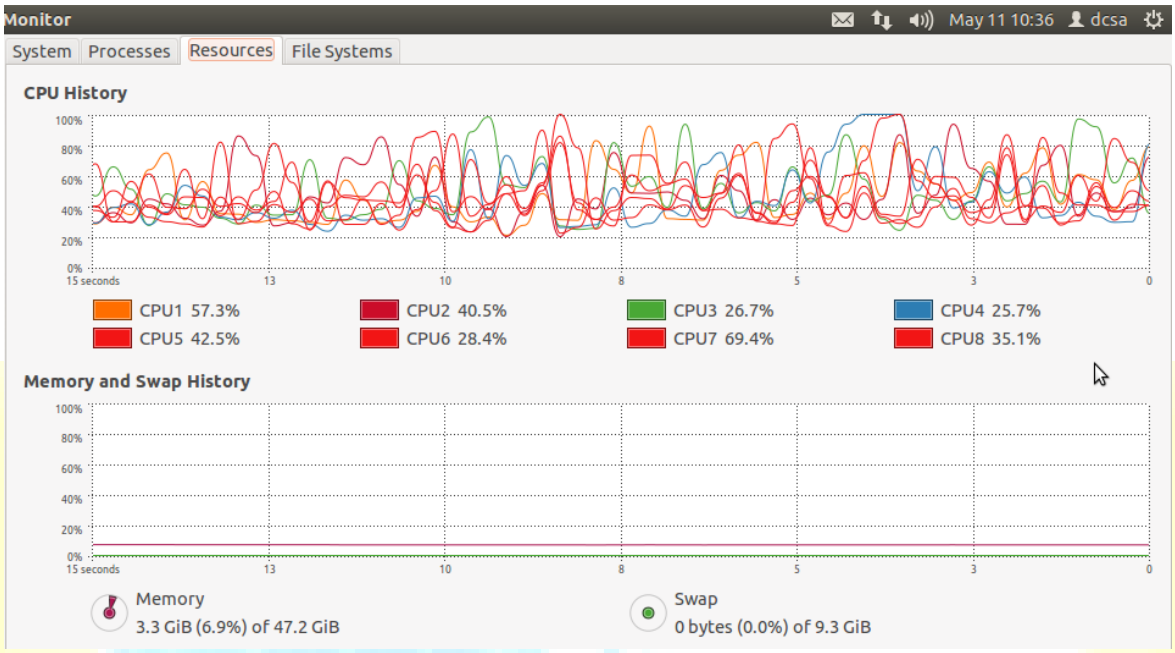

Figure 3: Parallel GA in execution

From these figures it can be observed that when Simple GA runs only one of the available CPU is busy in execution, whereas in Parallel GA graph all of the CPU in the hardware are busy to execute the program. At the end of the experiment the time taken by the SGA is far more in comparison to the PGA. Hence PGA performs better in terms of execution time as well as the solution quality.

IV. CONCLUSION

While solving any optimization problem one must have to keep in mind that one solution for the entire optimization problems may never be found. It depends on the problem that one has to tune his solutions slightly according to the nature of the problem. No doubt global search has very few chances to get stuck in local optima. So there is always option in global searches to look for better results. When Simple genetic algorithm generates new child for next generation then one can also exploit the parallelism in hardware to get more in same time. In this paper SGA and PGA are implemented and their execution time and performance are compared on a quad core processor.

REFERENCES

1. Holland J., (1975), *Adaptation in natural and artificial systems*, University of Michigan Press, Ann Arbor.
2. Rakesh Kumar, Girdhar Gopal, Rajesh Kumar, (2013), "*Hybridization in Genetic Algorithms*",

International Journal of Advanced Research in Computer Science and Software Engineering (IJARCSSE), Vol-3, Issue-4, pp 403-409.

3. Goldberg D. E., (1989), *Genetic algorithms in search, optimization, and machine learning*, Addison Wesley Longman, Inc., ISBN 0-201- 15767-5.
4. Rakesh Kumar, Girdhar Gopal, Rajesh Kumar, (2013), "*Novel Crossover Operator for Genetic Algorithm for Permutation Problems*", International Journal of Soft Computing and Engineering (IJSCE), Vol 3, Issue-2, pp 252-258.
5. Moscato P., Cotta C., (2003), "*A gentle introduction to memetic algorithms*", Handbook of metaheuristics, pp 105-144.
6. Bosworth Jack, Foo Norman, and Zeigler Bernard P. (1972), "*Comparison of Genetic Algorithms with Conjugate Gradient Methods*". Technical Report 00312-1-T, University of Michigan: Ann Arbor, MI, USA.
7. Bethke Albert Donally. (1980), "*Genetic Algorithms as Function Optimizers*". PhD thesis, University of Michigan: Ann Arbor, MI, USA.
8. Brady R. M. (1985), "*Optimization Strategies Gleaned from Biological Evolution*." Nature, 317(6040): 804–806, doi: 10.1038/317804a0.
9. Sinha Abhishek and Goldberg D.E. (2003), "*A Survey of Hybrid Genetic and Evolutionary Algorithms*". IlliGAL Report 2003-2004, Illinois Genetic Algorithms Laboratory (IlliGAL), Department of Computer Science, Department of General Engineering, University of Illinois at Urbana-Champaign: Urbana-Champaign, IL, USA.
10. Radcliffe Nicholas J. and Surry Patrick David. (1994), "*Formal Memetic Algorithms*". International Workshop on Evolutionary Computing, Selected Papers, pages 1–1.
11. Digalakis J.G. and Margaritis K.G.. (2002), "*An Experimental Study of Benchmarking Functions for Genetic Algorithms."* International Journal of Computer Mathematics, 79:4, 403-416.
12. Gorges-Schleuter Martina. (1989), "*ASPARAGOS: An Asynchronous Parallel Genetic Optimization Strategy*". Proceedings of the 3rd International Conference on Genetic Algorithms , pages 422–427.
13. Brown Donald E., Huntley Christopher L., and Spillane Andrew R.. (1989), "*A Parallel Genetic Heuristic for the Quadratic Assignment Problem*". Proceedings of the 3rd International Conference on Genetic Algorithms, pages 406–415.
14. Weise Thomas, (2011), "*Global Optimization Algorithms – Theory and Application –", 3rd Edition.*
15. Digalakis Jason and Margaritis Konstantinos. (2001), "*A Parallel Memetic Algorithm for Solving Optimization Problems.* "4th Metaheuristics International Conference, pages 121–125.