Implementation of a Parallel Genetic Algorithm on a Cluster of Workstations: The Travelling Salesman Problem, a Case Study

Giuseppe Sena[1]*, Germinal Isern[1]**, and Dalila Megherbi[2]

[1] Northeastern University, College of Computer Science,
Boston, MA 02115, U.S.A.
{tonysena,isern}@ccs.neu.edu

[2] University of Denver, Division of Engineering,
Denver, CO 80208, U.S.A.
dmegherb@du.edu

Abstract. A parallel version of a Genetic Algorithm is presented and implemented on a cluster of workstations. Even though our algorithm is general enough to be applied to a wide variety of problems, we used it to obtain optimal/suboptimal solutions to the well known Traveling Salesman Problem. The proposed algorithm is implemented using the Parallel Virtual Machine library over a network of workstations, and it is based on a master-slave paradigm and a distributed-memory approach. Tests were performed with clusters of 1, 2, 4, 8, and 16 workstations, using several real problems and population sizes. Results are presented to show how the performance of the algorithm is affected by variations on the number of slaves, population size, mutation rate, and mutation interval. The results presented show the utility, efficiency and potential value of the proposed algorithm to tackle similar NP-Complete problems.

1 Introduction

In this paper a parallel/distributed version of a *Genetic Algorithm* (GA) is presented and implemented over a cluster of workstations to obtain quasi-optimal solutions to the widely known *Traveling Salesman Problem* (TSP).

Genetic Algorithms are computational models inspired by the idea of evolution [15], and they were introduced by John Holland and his students [7, 3]. GAs encode solutions to a specific problem using a chromosome-like data structure and apply recombination operators to produce new individuals. GAs are often used as function optimizers, and are also considered as global search methods that do not use gradient information. Therefore, they might be applied to problems in which the function to be optimized is non-differentiable or with multiple local optima.

* ON Technology Corp., Cambridge, MA 02142, U.S.A., tsena@on.com.

** On leave from Universidad Central de Venezuela, Caracas, Venezuela.

A GA is inherently parallel [9], and at every iteration individuals are independently selected for crossover and mutation following some probability distribution. The proposed parallel and distributed implementation introduced in this paper uniformly decomposes the population among the available processors (hosts), so that genetic operators can be applied in parallel to multiple individuals. A *master-slave paradigm* is used to implement the Parallel Distributed Genetic Algorithm (PDGA).

The proposed PDGA is applied to the TSP [10, 4, 8]. The problem at hand is an optimization problem which consists of finding a *Hamiltonian* cycle of minimum length. Individuals in the population are Hamiltonian cycles represented as a sequence of vertices, which makes the mutation and crossover operators simpler and efficient. New heuristics as well as decisional rules are introduced to support computationally efficient migration of individuals between sub-populations in order to improve the gene pool.

Our PDGA is implemented on a network of SUN workstations (Ultra Sparc 1) running the Parallel Virtual Machine (PVM) library, and tests are performed using 1, 2, 4, 8, and 16 slave workstations. Several real problems and population sizes are used in our experiments, which are obtained from a library of TSPs called *TSPLIB* from Rice University.

The experimental results obtained are very promising. In particular, we show that as we increase the size of the population the performance of the PDGA improves as we increase the number of slave tasks used. We additionally show that with small population sizes (i.e. $p = 128$) the communication overhead overcomes the advantage of using our PDGA.

The rest of this paper is organized as follows: in section 2 we give an overview of the GA methodology used, present the problem encoding and the fitness function used. In addition, we show the general design of our PDGA, the problem parameters and the GA operators used. Section 3 describes our test environment, and the test results and analysis are presented in section 4. The conclusions and future research is presented in section 5.

2 GA Methodology

Usually, only two components of a GA are problem dependent: the *problem encoding* and the *evaluation (fitness) function*. The following subsections present the encoding, mutation and crossover operators, and the fitness function used for the TSP. In addition, an overview of the proposed PDGA is presented.

Three different ways of exploiting parallelism in GAs can be defined [15]: (a) a parallel GA similar to the canonical GA [7], (b) an *Island Model* that uses distinct sub-populations, and (c) a fine-grain massively parallel implementation where each individual resides in a processor. In this paper the *island model* is used in order to exploit a more coarse-grain parallelism, which can be easily extended to a distributed system. The general idea behind the *island model* is to distribute the total population among the available processors, and to execute a classical GA with each sub-population. Hence every few generations (based on well defined criteria), sub-populations could swap few individuals (chosen at random among the best fit). This migration process allows sub-populations to share "genetic material".

2.1 Problem Encoding

Traditional GAs use binary bit-strings to encode individuals in the population. However, many researchers [2, 10, 8] use real-valued or non-binary encodings instead of binary encodings, and employ *recombination operators* or *specialized operations* that may be problem or domain specific. For the TSP, individuals in the population are Hamiltonian cycles. If we try to represent Hamiltonian cycles as bit-strings we will soon find out that the mutation and crossover operators can generate invalid individuals with a very high probability. Hence the need for repairing algorithms to convert invalid bit-strings into valid individuals. Therefore, we represent a cycle as sequence of vertices, so that each Hamiltonian cycle could be one of the possible $n!$ permutations of the vertices of $G = (V, E)$.

2.2 Fitness Function

Let us assume that $G = (V, E)$ is the graph representing the problem, with $V = \{v_0, v_1, \ldots, v_{n-1}\}$. Each individual in the population is a Hamiltonian cycle HC_k $(k = 1, 2, \ldots, n!)$ in G. Let us define $V_n = \{p_1, p_2, \ldots, p_{n!}\}$ as the set of all possible permutations of the elements of V. The *fitness* function F is defined as:

$$F(HC_k) = \sum_{i=0}^{n-1} Cost(v_{p_k(i)}, v_{p_k((i+1) \bmod n)}) \quad \forall k \in [1, n!] \ . \tag{1}$$

where $HC_k = \langle v_{p_k(0)}, v_{p_k(1)}, \ldots, v_{p_k(n-1)} \rangle$ and $Cost(v_i, v_j)$ is the distance between each pair of cities v_i and v_j. Therefore, $F(HC_k)$ would be the tour-length of the Hamiltonian cycle HC_k.

2.3 PDGA: Distributed-Memory Approach

The basic idea of a GA is simply to generate a random population of individuals and apply mutation and crossover operators [4, 17] to a sub-population in each generation until an individual is found to satisfy some criteria for *fitness* or a maximum number of generations (iterations) is reached. It should be clear at this point, that there is an implicit parallelism in this basic GA and one could apply this basic GA to sub-populations in parallel in order to improve the performance of the algorithm.

Several *Parallel Genetic Algorithms* (PGA) have been proposed in the literature [14, 16, 12, 10, 6, 13]. In this paper we present the design and implementation of a *Parallel-Distributed Genetic Algorithm* (PDGA) based on a distributed-memory approach. Figure 1 shows the general architecture of the proposed PDGA, and the interaction between the MASTER and SLAVE tasks. The idea is to start-up a MASTER task whose main purpose is to control the

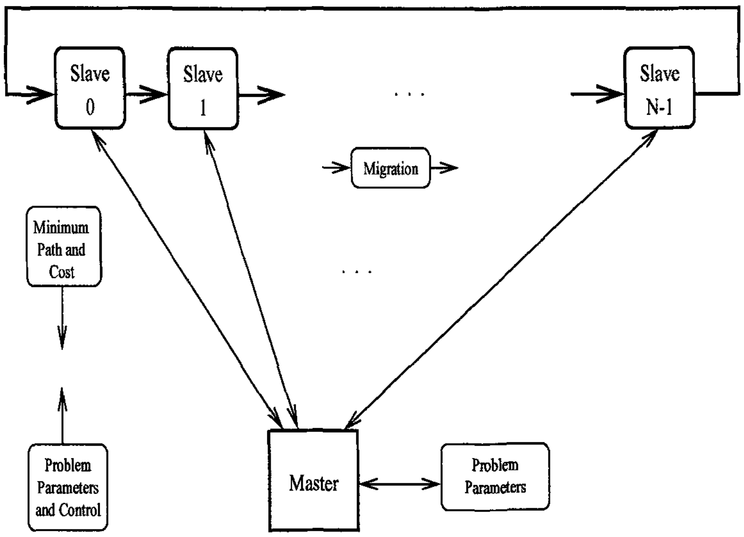


Fig. 1. PDGA: General Design

operations of the system by broadcasting the problem parameters and by synchronizing the operation of the SLAVE tasks. The master task spawns as many slave tasks as specified by the user and uniformly decomposes the population among those slaves. Every slave task receives the problem parameters from the master task, generates a random sub-population and starts executing a basic GA [9] within its own sub-population. During every generation and after each slave task has performed the crossover and mutation operations (and probably migration), each slave task sends back to the master task the minimum-cost Hamiltonian cycle found in that generation. Based on the results received, the master decides if it has reached the goal or if it is necessary to compute the next generation.

2.4 Problem Parameters

The input parameters required by the PDGA to solve the TSP are: a) **nslaves**: number of slave tasks, b) **n**: number of vertices (cities) in the graph, c) **p**: total number of Hamiltonian cycles in the population, d) **r**: crossover rate, e) **m**: mutation rate, f) **mut_int**: mutation interval, g) **mig_rate**: migration rate, h) **mig_int**: migration interval, and i) **min_threshold**: minimum tour-length accepted to stop the algorithm.

2.5 GA Operators

In this section we present the GA operators used for mutation and crossover, as well as the way we perform migration of sub-populations. The proposed PDGA is an example of what is called a *steady state GA* [2], which means that offsprings do not replace parents, instead they replace some less fit members of the population.

Many mutation operators for the TSP have been proposed in the literature [4, 10, 17, 8], and we selected one that is simple to implement when cycles are represented as a sequence of vertices: choose two of the vertices in the cycle at random and swap them. A mutation operation is performed as often as indicated by the mutation interval parameter (**mut_int**).

There are also several crossover operators that have been studied for the TSP [4, 10, 17, 8]. In order to apply a GA to permutation problems (like the TSP), it is necessary to use special operators to ensure that the recombination of two permutations (individuals) also yields another permutation [11]. Again, due to its simplicity, we chose here a crossover operator called *OX operator* [17], also known as *order crossover operator.* The crossover operation (mating) is performed every generation (iteration) with some percentage of the population (**r**) selected at random.

As we mentioned before, we use an *island model* to exploit differences among sub-populations. Each sub-population represents an island and we move the genetic material between islands in a specific way. The migration of a percentage of the population (**mig_rate**) is performed every **mig_int** generations between consecutive slave tasks as if they were connected in a logical ring topology. Each slave task S_i chooses the best individuals in their sub-population and sends them to its successor $S_{(i+1) \bmod nslaves}$ in the ring, and receives those individuals from its predecessor task $S_{(i-1) \bmod nslaves}$ and replaces the worst fit individuals with the ones just received. After several migrations one may observe that every slave task will have most of the best individuals found by the others. In addition this process will provide more diversity to the gene pool used for the search.

3 Test Environment (Experiments)

The proposed PDGA was implemented using PVM [5], which is a message passing library that allows programmers to write parallel/distributed applications using a NoW [1]. Figure 1 shows the message exchange that occurs between the MASTER and SLAVE tasks.

Tests were performed on a cluster of twenty (20) SUN SparcStations (Ultra Sparc 1) running *Solaris 2.6* and *PVM 3.4b.* For debugging and virtual machine configuration we used *XPVM 1.1*, an X Windows interface for the PVM console program. We tested our algorithm using 1, 2, 4, 8 and 16 slave tasks running in parallel, and different population sizes (p =128, 512, 1024, and 2048) changing only the number of slave tasks in order to observe and study the increase in performance (speedup). In addition, we studied the effect of variations in the mutation rate (**m**) and mutation interval (**mut_int**) on the total execution time and convergence rate of the proposed algorithm.

We used a library of TSP problems (TSPLIB) from Rice University that contains real data for graphs of different sizes. Vertices in the graphs represent cities in countries around the world, and the cost associated with every edge represents distances between each pair of cities. Test runs were performed with data from cities in Germany and Switzerland. We present the results obtained from data in Bavaria, Germany (**bavaria29.data**), represented by a complete undirected graph of $n = 29$ vertices, and with optimal tour-length of 2020 (problem parameters: $r = m = 0.5$, $mut\_int = 20$, $mig\_int = 25$, and $mig\_rate = 0.1$).

In order to determine the effectiveness of the proposed PDGA we selected TSP problems for which the minimum tour-length cost was known. In addition, we required the PDGA to stop if and only if the derived solution was within 1% off the optimal solution provided (for **bavaria29.data** $min\_threshold = 2040$).

4 Test Results and Analysis

4.1 Effect of Population Size and Number of Slave Tasks

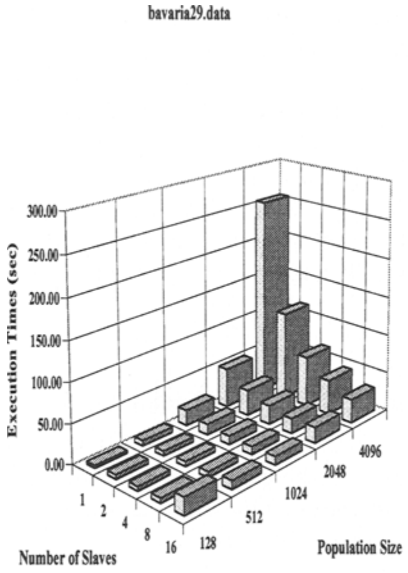
In light of the results presented in figures 2 and 3, one can observe that as the size of the population increased the performance of the parallel version improved (proportionally) as the number of slave tasks increased. One may also notice that for small populations (i.e. $p = 128$) the performance decreased as the number of slaves increased. In addition, when the number of slaves increased the communication and synchronization overhead increased and the program spent more time in communication than performing useful computations.

We also notice from fig. 2 that one cannot keep increasing arbitrarily the number of slave tasks used. There is a maximum number of slaves after which the execution time increases instead of decreasing (i.e. 8 slaves). This result can be justified by and is in line with the fact that, depending on the population size and the number of slaves, if the number of slaves keeps increasing the communication and synchronization overhead will overcome the advantage of using the PDGA. The worst case happened when we used very small populations (i.e. $p = 128$). In these cases with a single slave (sequential program) we obtained better results (most of the time) than with 4, 8, or 16 slaves. This shows again that the proposed PDGA is better suited for large populations. It is also worth and important to mention that with small populations a parallel and distributed version of a GA is most likely to converge to a local minima due to a small gene pool. Figure 3 shows that the number of generations is virtually invariant to the increase in the number of slave tasks used.

We should emphasize that all tests were performed on a cluster of workstations connected through an Ethernet LAN (baseband network), which becomes a bottleneck when the number of workstations is increased. In order to obtain better performance as the number of workstations is increased, it would be necessary to replace the interconnection network used (Ethernet) by a high-performance network (like ATM, FastEthernet, etc.).

4.2 Effect of the Mutation Interval (mut_int)

Figure 4 presents the results of varying the mutation interval, and how it affects the convergence and total execution time of the PDGA. These tests were per-



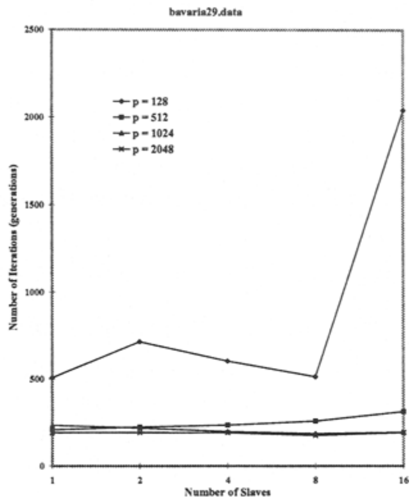


Fig. 2. (*Execution time*) as a function of (*number of slaves*) and (*population size*)

Fig. 3. (*Number of generations*) as a function of (*number of slaves*)

formed with 8 slaves, a population of $p = 1024$, and a mutation rate of $m = 10\%$. In the top of the graph we observe that the execution time decreases rapidly at first and then slowly reaches a minimum around $mut\_int = 30$. After that it begins to increase again. It is worthwhile to mention that for $mut\_int \leq 2$ the PDGA diverges. This behavior can be justified and explained by the fact that in this case mutations are performed almost on every generation, which does not allow the PDGA to get stabilized towards an optimal solution. The bottom of the graph shows a behavior similar to the behavior illustrated in the top of the graph, but with respect to the number of generations (iterations).

4.3 Effect of the Mutation Rate (m)

Figure 5 presents the results of varying the mutation rate, and how this affects the convergence and total execution time of the PDGA. These tests were performed with 8 slaves, a population of $p = 1024$, and a mutation interval, mut_int, equal to 20. In the top of the graph we observe that the execution time does not change drastically with variations in the mutation rate. However, the execution time slightly improves as we reduce the mutation rate below 30%, which seems logical because we are reducing the amount of work each slave must do to perform mutations. The bottom of the graph shows a behavior similar to the behavior illustrated in the top of the graph, but with respect to the number of generations.

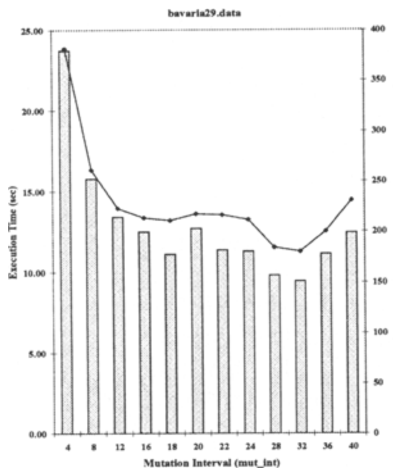


Fig. 4. Effect of the Mutation Interval (*mut_int*) on the PDGA

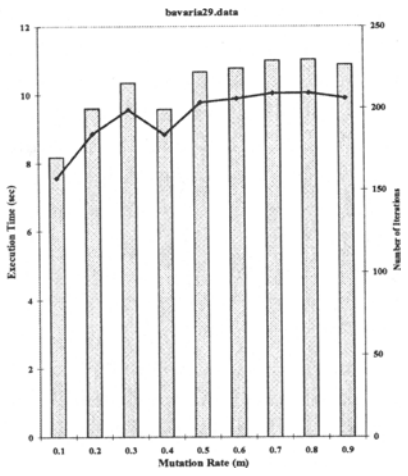


Fig. 5. Effect of the Mutation Rate (m) on the PDGA

5 Conclusions and Future Work

In this paper we introduced a parallel/distributed paradigm of a GA (PDGA) and we successfully applied it to the TSP. We used PVM to implement the proposed PDGA and we tested it on real data. We compared the proposed PDGA with a sequential GA and showed that said PDGA yielded very good results for large populations. Moreover, we also showed that, for a given population size, it is not recommended to increase the number of slave tasks arbitrarily. In particular, it has been shown in this paper that there is an optimum combination, in terms of population size and number of slave tasks, which leads to a better computational speed.

There are still many unanswered questions remaining. It is necessary to experiment with different encoding, crossover and mutation operators. We should also study in more detail how the mutation and migration rates affect the convergence speed of our PDGA. It would be interesting to test the PDGA with very large size problems (i.e. graphs with thousands of nodes), as well as other variations of the classical TSP. We are experimenting with several new heuristics as well as decisional rules to support computationally efficient migration to improve the gene pool. In particular, we are studying several migration schemes (i.e. random migration, alternating slaves, etc.). We are also analyzing other implementations of the basic PDGA using a different programming paradigm (i.e. MPMD) in order to reduce the intrinsic communication overhead and the need of synchronization exposed by the master-slave paradigm. We also must produce a deeper performance analysis considering granularity (communication vs. computation time and frequency) and scalability factors, in order to tune up our parallel/distributed implementation.

We will also be studying how to apply GAs and our experience with the TSP to other scheduling and networking problems (i.e. general routing, ATM routing and bandwidth allocation, load balancing, etc.). We are also developing a new implementation of our PDGA that will run over a network of heterogenous workstations directly attached to an ATM switch and with the latest version of PVM. Despite the lack of a deep performance analysis of the PDGA proposed, the results presented in this paper show the utility, versatility, efficiency and potential value of the proposed PDGA to solve and tackle NP-Complete problems. The PDGA presented is not only a good example of parallel/distributed programming, but it also shows how an inherent parallel algorithm can be efficiently implemented using well understood programming paradigms, flow control techniques, and load balancing approaches.

References

[1] T.E. Anderson, D.E. Culler, D.A. Patterson, and the NOW team. A Case of NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[2] L. Davis, editor. *Handbook of GA.* Van Nostrand Reinhold, NY, 1991.

[3] K. DeJong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* Phd's thesis, University of Michigan, Ann Arbor, MI, 1975.

[4] B.R. Fox and M.B. McMahon. Genetic Operators for Sequencing Problems. In Rawlins [12], pages 284–300.

[5] A. Geist, A. Beguellin, J. Dongarra, et al. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Net. Parallel Comp.* The MIT Press, 1994.

[6] M. Gorges-Schleuter. Explicit Parallelism of Genetic Algorithms through Population Structures. *Parallel Problem Solving from Nature*, pages 150–159, 1991.

[7] J. Holland. *Adaptation in Natural and Artificial Systems.* U. Mich. Press, 1975.

[8] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolutionary Programs.* Springer-Verlag, Berlin, Germany, 1993.

[9] T.M. Mitchell. *Machine Learning.* Series in CS. McGraw-Hill, 1997.

[10] H. Mühlenbein. Evolution in Time and Space - The Parallel Genetic Algorithm. In Rawlins [12], pages 317–337.

[11] S. Rana, A.E. Howe, D. Whitley, and K. Mathias. Comparing Heuristic, Evolutionary and Local Search Approaches to Scheduling. In *Proc. of the* 3^{rd} *Artificial Intelligence Planning Systems Conference*, 1996.

[12] G.J.E. Rawlins, editor. *Foundations of Genetic Algorithms.* Morgan-Kaufmann Publishers, Inc., San Mateo, California, 1991.

[13] T. Starkweather, D. Whitley, and K. Mathias. Optimization Using Distributed Genetic Algorithms. *Parallel Problem Solving from Nature*, 1991.

[14] R. Tanese. Distributed Genetic Algorithms. In *Proc. of the* 3^{rd} *Int. Conf. on Genetic Algorithms*, pages 434–439. Morgan-Kaufmann Publishers, Inc., 1989.

[15] D. Whitley. A Genetic Algorithm Tutorial. *Stat. and Computing*, 4:65–85, 1994.

[16] D. Whitley and T. Starkweather. Genitor II: a Distributed Genetic Algorithm. *Journal Expt. Theory Artificial Intelligence*, 2:189–214, 1990.

[17] D. Whitley, T. Starkweather, and D. Shaner. The Traveling Salesman and Sequence Scheduling: Quality Solutions Using Genetic Edge Recombination. In Davis [2], chapter 22, pages 350–372.