An empirical study of parallel genetic algorithms for the traveling salesman and job-shop scheduling problems

by David Michael Helzer

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science
Montana State University

Abstract:
A genetic algorithm (GA) is a problem-solving technique that is based on the concepts of natural selection and genetics. GAs have become powerful methods for solving difficult optimization problems.

A GA begins with a varied population of individuals, each of which is a candidate solution to the problem being solved. The fitness of each individual is determined by how well it solves the particular problem. The GA evolves this population using crossover and mutation operators. Those individuals with high fitness levels are more likely to be selected for crossover than those with low fitness levels. The goal of the GA is to evolve a population of highly fit individuals, one of which can serve as the solution to the problem.

A GA can be parallelized to reduce its execution time by dividing the population of individuals among a number of different processors, each of which represents an island. The islands of subpopulations evolve independently from each other, promoting diversity among the indidividuals. Occasionally, individuals may be allowed to migrate between islands, depending on the physical topology of the islands.

This thesis describes how GAs have been implemented to solve the traveling salesman and job-shop scheduling problems. Four variations of PGAs have been designed, each of which uses a different island topology.

The four PGAs and the serial GA are compared for solution qualities obtained and execution times required. The PGA with high island connectivity and migration is shown to obtain higher quality solutions than other models examined in most cases. PGA models have shown to obtain solutions with error percentage improvements over serial GAs of 100.00%, 6.49%, and 21.21% for three selected traveling salesman problem data sets. These PGA models completed their executions in 12.1%, 7.2%, and 7.7% of the elapsed times required by the serial models. For the job-shop scheduling problem, solutions with error percentages that were 28.42%, 28.60%, and 64.50% better than the serial models' best solutions were achieved by the PGAs for three data sets. The elapsed times required by these PGAs were 15.1%, 13.4%, and 13.1% of the times required by the serial GAs.

AN EMPIRICAL STUDY OF PARALLEL GENETIC ALGORITHMS FOR THE TRAVELING SALESMAN AND JOB-SHOP SCHEDULING PROBLEMS

by

David Michael Helzer

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

April 2001

APPROVAL

of a thesis submitted by

David Michael Helzer

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

Dr. Year-Back Yoo — Y. B. Yoo (Signature) — 4/19/01 Date

Approved for the Department of Computer Science

Dr. J. Denbigh Starkey — J Denbigh Starkey (Signature) — 4/19/01 Date

Approved for the College of Graduate Studies

Dr. Bruce McLeod — Bruce R. McLeod (Signature) — 4-19-01 Date

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Signature David Heker

Date April 19, 2001

ACKNOWLEDGEMENTS

I would like to thank Dr. Year-Back Yoo for introducing me to the field of parallel computing and for being a source of much guidance and expertise throughout the research process of this thesis. Much gratitude is also given to the other members of my graduate committee, Drs. Gary Harkin, John Paxton, and Denbigh Starkey, for providing me with a computer science education that has prepared me for a future in the software field.

I would also like to thank my parents, Richard and Renae Helzer, for instilling in me a strong desire to succeed. Thanks are given to my sister, Melissa Helzer, for her support and friendship.

This work was sponsored by the National Science Foundation's Partnership for Advanced Computational Infrastructure with computing time from the Pittsburgh Supercomputing Center through grant number ASC010015P.

TABLE OF CONTENTS

TABLE OF CONTENTS – CONTINUED

TABLE OF CONTENTS – CONTINUED

LIST OF TABLES

LIST OF TABLES – CONTINUED

LIST OF FIGURES

LIST OF FIGURES – CONTINUED

LIST OF FIGURES – CONTINUED

ABSTRACT

A genetic algorithm (GA) is a problem-solving technique that is based on the concepts of natural selection and genetics. GAs have become powerful methods for solving difficult optimization problems.

A GA begins with a varied population of individuals, each of which is a candidate solution to the problem being solved. The fitness of each individual is determined by how well it solves the particular problem. The GA evolves this population using crossover and mutation operators. Those individuals with high fitness levels are more likely to be selected for crossover than those with low fitness levels. The goal of the GA is to evolve a population of highly fit individuals, one of which can serve as the solution to the problem.

A GA can be parallelized to reduce its execution time by dividing the population of individuals among a number of different processors, each of which represents an island. The islands of subpopulations evolve independently from each other, promoting diversity among the indidividuals. Occasionally, individuals may be allowed to migrate between islands, depending on the physical topology of the islands.

This thesis describes how GAs have been implemented to solve the traveling salesman and job-shop scheduling problems. Four variations of PGAs have been designed, each of which uses a different island topology.

The four PGAs and the serial GA are compared for solution qualities obtained and execution times required. The PGA with high island connectivity and migration is shown to obtain higher quality solutions than other models examined in most cases. PGA models have shown to obtain solutions with error percentage improvements over serial GAs of 100.00%, 6.49%, and 21.21% for three selected traveling salesman problem data sets. These PGA models completed their executions in 12.1%, 7.2%, and 7.7% of the elapsed times required by the serial models. For the job-shop scheduling problem, solutions with error percentages that were 28.42%, 28.60%, and 64.50% better than the serial models' best solutions were achieved by the PGAs for three data sets. The elapsed times required by these PGAs were 15.1%, 13.4%, and 13.1% of the times required by the serial GAs.

CHAPTER 1

INTRODUCTION

Biological Foundations

The field of computer science is devoted to using computers to find solutions to various problems. A genetic algorithm (GA) is one such method used to solve problems. The ideas behind GAs are based on Charles Darwin's theory of natural selection and basic genetic principles. Darwin reasoned that some living organisms within a population are more adapted to their environment than other organisms. Those that are more adapted are said to have high fitness levels. Having high fitness levels provides individuals with a greater chance of surviving to reproduce.

Gregor Mendel was the first biologist to explain how traits pass from parents to offspring in the form of genes. The offspring of sexual reproduction typically do not identically resemble their parents. Instead, the genetic code of a child organism is obtained by combining the genetic codes of both parent organisms. Fit organisms, then, are likely to pass good fragments of genetic code to their young, thereby creating fit offspring. Over time, a population of organisms is likely to evolve into a more fit population and become more adapted to its environment.

Genetic Algorithm Overview

John Holland, at the University of Michigan, developed genetic algorithms in the 1960's. It was in 1975 that his ideas were presented to the world in his book *Adaptation in Natural and Artificial Systems* (Holland, 1975). GAs are relatively easy to implement with a computer program and have been used to successfully find solutions to a variety of problems including scheduling, optimization, and machine learning.

GAs begin with a population of individuals, each representing a candidate solution, or *chromosome*, to the problem being solved. The chromosomes themselves are made up of a set of values, or *genes*. The possible values that each gene can take on are known as the gene *alleles*. Each of these solutions is evaluated with a fitness function used to determine how well the solution solves the problem. The candidate solutions are selected based on their fitness levels to be *crossed* with other solutions to form new chromosomes (simulating reproduction in the natural world). A GA may also employ *mutation*, or random alteration of the genetic codes of individuals. During each evolutionary step in a GA, a set of children chromosomes is generated to replace the parent chromosomes. With each generation, better solutions are strived for until, after creating the final generation, the best candidate solution in the population is selected to serve as the solution to the problem being solved.

Parallel Genetic Algorithm Overview

Since the creation of the numerous generations in a GA can require a substantial amount of time to run on a computer, GAs are good candidates for parallel processing. Parallel computing seeks to use multiple central processing units (CPUs) within a single computer system to work together to solve problems. Typically, parallel computer systems can be expected to solve problems faster than serial systems (those with only one CPU).

Various methods for parallelization of GAs exist. One of the most popular of these methods is the island parallel genetic algorithm (PGA), during which the initial generation's chromosomes are divided among a set of processors. Each processor represents an island that is separated from the other islands. Each island's subpopulation of candidate solutions is evolved independently from the other subpopulations. This helps to promote diversity among the individuals and can help to generate higher quality solutions faster than would be possible with a serial GA. Occasionally, candidate solutions are allowed to migrate between islands, to ensure that the good solutions are allowed to spread to other subpopulations and so the genetic algorithm, as a whole, is working toward a common solution. The physical layout, or *topology*, of the islands determines which islands are allowed to exchange individuals via migration.

Experimental Goals

This thesis seeks to explain how genetic algorithms can be implemented to solve problems and how they can be parallelized. Little documentation exists comparing the quality of PGAs with that of serial GAs. Two optimization problems, the traveling salesman and job-shop scheduling, are presented, along with methods used to find solutions to those problems with a GA. Additionally, new experiments have been performed to determine how the serial genetic algorithm compares to four variations of the island model PGA. Answers to the following three questions were reached.

1) What type of island topology can be expected to obtain the best solution qualities when using a PGA?
2) How much improvement in solution quality can be expected when using an island model PGA compared to a serial GA?
3) How much speedup can be expected when using an island model PGA compared to a serial GA?

It is hoped that the results obtained from the experiments with the traveling salesman and job-shop scheduling problems can be generalized to other problems that can be solved with a GA as well. The goal of this thesis has not been to develop the best possible genetic algorithms to solve these problems. Instead, the focus has been on determining how the serial and parallel variations of the programs compare with each other.

Hardware Resources

The National Science Foundation's Parternship for Advanced Computational Infrastructure supported this research by providing computing time on a Cray T3E system at the Pittsburgh Supercomputing Center. The Cray T3E contains 512 Digital Alpha 450 MHz processors arranged in a three-dimensional torus topology. The memory of the system is physically distributed with 128 MB for each processor.

Message Passing Interface

To implement the parallelism and perform the communication between the processors, the Message Passing Interface (MPI) was used. MPI is a popular standard that includes a set of portable library functions that allow inter-process communication to take place. When using MPI, the number of processors desired is specified when running a program. At the start of the program, a process is launched on each of the processors desired. Each process is assigned a unique identification number. It is with this identification number that processes can indicate the other processes with which they are to communicate. All of the programs were implemented in C.

CHAPTER 2

GENETIC ALGORITHMS

Combinatorial Optimization Problems

A genetic algorithm is composed of a set of operations modeled after those found in the natural world. GAs are most often used to find good solutions to combinatorial optimization problems that have large search spaces. A combinatorial optimization problem is one for which a discrete set of possible solutions exists. Examples of such problems include ordering, scheduling, and packing problems. For all of these problem types, there exists an optimal solution in a search space of all candidate solutions. Many of these types of problems belong to time complexity classes for which polynomial solutions remain unknown. To perform an exhaustive search through a large search space to find the optimal solution to one of these problems could require a great deal of time.

Heuristics and the Search Space

A GA is one of many heuristics classified under the field of artificial intelligence. A heuristic is a set of rules used to guide a search through a search space. Rather than performing an exhaustive search through all candidate solutions in a search space, heuristics seek to examine only some of the candidate solutions and still obtain a good one. Other heuristic techniques include simulated annealing and tabu search.

An example of a search space is shown in Figure 1. This space is that of the function $f(x, y) = x + y$, where $(0 \leq x, y \leq 10)$. One problem to be solved may be to find the (x, y) values that maximize this function. The solution, then, would be the pair $(x = 10, y = 10)$. One can see that even when only integer values are considered, there are eleven possible values that the parameters x and y can take. This gives a total of 121 different combinations of (x, y) pairs that would have to be examined to perform an exhaustive search and guarantee that the function be maximized. One could design a genetic algorithm to maximize the value for $f(x,y)$ and avoid having to perform such a search.

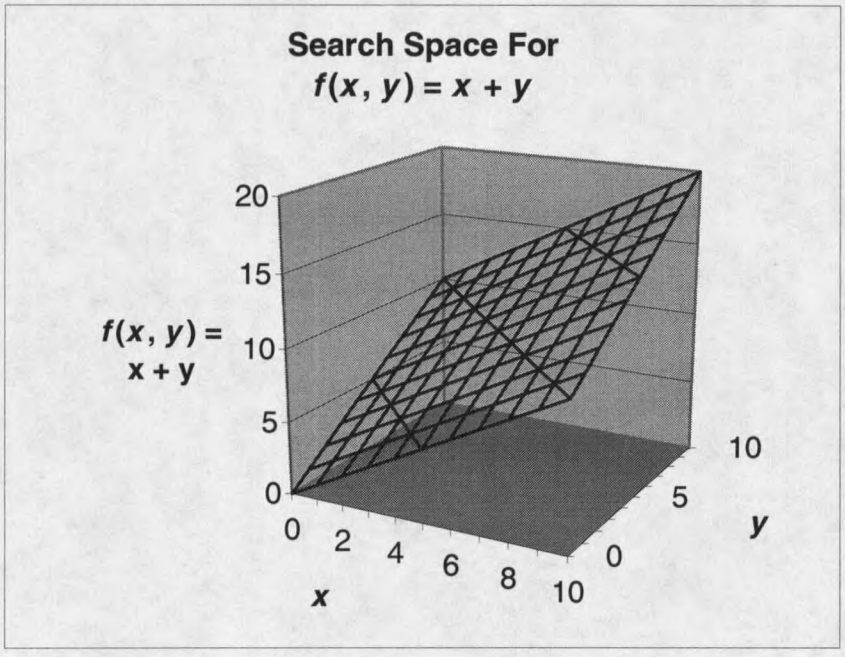


Figure 1: Search Space for the Function $F(x, y) = x + y$

One benefit to using a GA is that it can overcome the challenges posed by search spaces that are uneven. Such search spaces may have groups of good solutions separated by bad ones. Rather than converging upon a local optimum as many other heuristics tend to do with uneven search spaces, a genetic algorithm is able to search through many locations in a search space simultaneously, thereby reducing the chances of a poor, premature convergence (Mitchell, 1996).

Genetic Algorithm Definition

A typical GA follows the steps shown in Figure 2, each of which will be described in further detail. Much variation can exist in GAs. However, the steps shown here are common in most.

```
 1) Randomly generate n individuals.
 2) Loop once for each generation desired:
 3)    Calculate the fitness f(x) of each individual x.
 4)    Place the best number_elite individuals in new population.
 5)    Loop to generate (n - number_elite) new individuals:
 6)       Select two parent chromosomes.
 7)       Cross the parents to generate two new individuals.
 8)       With some probability, mutate the children.
 9)       Place the two children into a new population.
10)    End loop.
10)    Replace the current population with the new population.
11) End loop.
11) The most fit individual is selected as the solution.
```

Figure 2: Genetic Algorithm

Initial Population

A GA must begin with some initial population of individuals, each of which represents a candidate solution to the problem being solved. Typically, this population of chromosomes is generated using some method involving randomness. It has been shown that starting with an initial population of chromosomes with high fitness levels can be beneficial. They should be generated using some type of randomized heuristic that is known to build high quality solutions to the problem (Ahuja & Orlin, 1997). As in the field of genetics, each chromosome is made up of a set of genes. In nature, these genes encode some type of individual trait; in a genetic algorithm, the genes encode a candidate solution.

Selection of Individuals

To make use of the idea of natural selection, a GA must provide a method by which fit individuals are more likely to be selected for crossover than weak individuals. This is performed by developing a problem-specific fitness function that determines how well a candidate solution solves the problem. Once the fitness levels of the chromosomes in a population are known, a probability of selection can be assigned to each chromosome. Selection of only the best individuals can cause a GA's population to quickly become very similar, thereby converging upon an answer without examining enough of the search space. Selection of too many weak individuals may prevent a GA

from locating the best solutions due to concentrating too heavily on the poor ones (Mitchell, 1996).

Fitness-Proportionate Selection

Under fitness-proportionate selection, an individual's probability of selection is directly proportionate to its fitness level (Mitchell, 1996). The probability of an individual being selected using fitness-proportionate selection is given by the formula

$$P(x) = \frac{Fitness(x)}{\sum_{i=1}^{n} Fitness(i)}.$$

An example population of individuals that has been assigned selection probabilities in this manner is listed in Table 1.

x	*Fitness (high is good)*	*Selection Probability*
1	2	2 / 16 = 0.13
2	4	4 / 16 = 0.25
3	3	3 / 16 = 0.19
4	7	7 / 16 = 0.44
	Sum = 16	*Sum = 1.0*

Table 1: Example Population with Fitness-Proportionate Selection Probabilities

Rank Selection

Rank selection can serve as an alternative to fitness-proportionate selection. Under rank selection, very weak individuals have a greater chance of being selected than

under fitness-proportionate selection, possibly preventing premature convergence (Mitchell, 1996). When using this method, a population's individuals are first ranked according to their fitness levels from n to 1 (n being the chromosome having the highest fitness level). The probability of selecting an individual x is given by the formula

$$P(x) = \frac{Rank(x)}{\sum_{i=1}^{n} Rank(i)}.$$

An example of a population with selection probabilities assigned using rank selection assignment is shown in Table 2.

x	*Fitness Level (high is good)*	*Rank*	*Selection Probability*
1	2	1	1/10 = 0.10
2	4	3	3/10 = 0.30
3	3	2	2/10 = 0.20
4	7	4	4/10 = 0.40
		Sum = 10	*Sum = 1.0*

Table 2: Example Population with Ranked Selection Probabilities

Solution Encoding

Whenever designing any type of computer program, potential solutions to the problem being solved must be represented in some way. The representation chosen for a GA is especially important since it can determine how well it performs. It must be flexible enough to provide crossover and mutation operators. It is important that all possible solutions be able to be encoded and that only valid solutions be represented. It is

also best if a small change in a chromosome produces only a small change in the solution it represents (Kershenbaum, 1997).

Binary Encodings

The simplest method for encoding solutions is using only strings containing the binary numbers zero and one. With only binary numbers to represent solutions, chromosomes are typically very long strings. Holland argued that solutions encoded with long strings containing a small number of alleles perform better than those encoded with short strings containing a large number of alleles. He reasoned that long strings promote more implicit parallelism, which will be described shortly (Mitchell, 1996).

An example of a binary encoding system is shown in Figure 3. The problem is to find the square root of the number 16. While this is not a combinatorial optimization problem, it could be solved using a genetic algorithm. The binary equivalents of the numbers being represented could be used for the actual chromosomes.

Problem: Find the square root of 16.

Chromosome	01101	00101	00100	10001
Solution	13	5	4	17

Figure 3: Binary Encoding Example

Many-Character Encodings

Rather than using only binary numbers to represent candidate solutions, a GA can use a wider range of characters. Such an encoding is referred to as a many-character encoding. When using a wider variety of characters to represent solutions, chromosomes tend to be much shorter than when represented with only binary numbers. They are natural to use but can be difficult to provide the flexibility required for the crossover and mutation operators. The theory that they do not perform as well as binary encodings has been questioned. The performance seems to depend on the problem being solved and the biological operators used (Mitchell, 1996).

An example of a many-character encoding system is shown in Figure 4. The problem being solved is the traveling salesman. Each of the cities is assigned an index number. The chromosomes, then, contain an ordered list of indices representing the order in which the cities are visited.

Problem: Traveling Salesman – Given a set of city indices and distances between those cities, find the shortest path that visits all cities exactly once and ends on the same city it starts with.

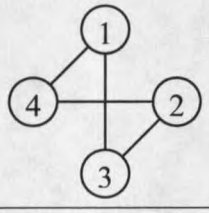
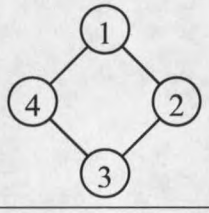
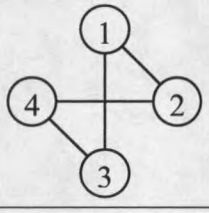
Chromosome	1-3-2-4-1	1-2-3-4-1	1-2-4-3-1
Solution			

Figure 4: Many-Character Encoding Example