

Rt
00566
c.1

J

**A Portable Message Passing Distributed Library
for Optimizing Combinatorial Problems
with Application to Circuits Testing**

by

Ahmad El Maamoun

M.S., Computer Science, Lebanese American University, Byblos

Thesis submitted in partial fulfillment of the requirements for the Degree of Masters of

Science in Computer Science

School of Arts and Sciences

LEBANESE AMERICAN UNIVERSITY

June 2007



LEBANESE AMERICAN UNIVERSITY

School of *Arts* and Sciences

Thesis Approval

Student Name **AHMAD EL MAAMOUN** I.D.#: **199604830**

Thesis Title: ***A PORTABLE MESSAGE PASSING DISTRIBUTED LIBRARY FOR OPTIMIZING
COMBINATORIAL PROBLEMS WITH APPLICATIONS TO CIRCUITS TESTING***

Program: **Computer Science**

Division /Dept: **Computer Science and Mathematics**

School: **School of Arts and Sciences, Byblos**

Approved by:

Thesis Advisor: ***Haidar M. Harmanani***

Member ***Danielle Azar***

Member ***Mounjed Moussallam***

Member

Date: **JUNE 29, 2007**



Plagiarism Policy Compliance Statement

I certify that I have read and understood LAU's Plagiarism Policy. I understand that failure to comply with this Policy can lead to academic and disciplinary actions against me.

This work is substantially my own, and to the extent that any part of this work is not my own I have indicated that by acknowledging its sources.

Name: *Ahmad el Maamoun*

Signature:



Date: *04-07-2007*

I grant to the LEBANESE AMERICAN UNIVERSITY the right to use this work, irrespective of any copyright, for the University's own purpose without cost to the University or its students and employees. I further agree that the University may reproduce and provide single copies of the work to the public for the cost of reproduction.

To my parents,

Acknowledgment

I would like to thank my advisor Dr. Haidar Harmanani Advisor for his guidance throughout my Thesis work. A thanks is also to Mr. Munjid Musallam and Dr. Danielle Azar for being on my thesis committee.

I would like to express my sincere gratitude to the Lebanese American University whose financial support during my graduate studies made it all possible.

Finally, I would like to thank my family for their long support.

Abstract

Various heuristic algorithms have been used to tackle combinatorial problems such as genetic algorithms and simulated annealing. Recently, MPI has recently emerged as a standard for parallel programming on cluster-based machines. However, there is a difficulty in proposing an MPI Java port.

This Thesis proposes a portable and distributed library for solving combinatorial optimization problems using Java. The library facilitates the use of genetic algorithms and simulated annealing and uses an MPI style message passing mechanism in order to create parallel processes that can communicate across the network using Java RMI. The library is optimized for communication and speed and improves the programmers efficiency through a visual interface. We verify our approach by formulating a new solution for the test generation problem in VLSI circuits based on parallel fault simulation. The problem is formulated and solved using GA and SA and solved using the proposed Library. The ISCAS benchmarks are attempted and favorable results are reported.

Table of Contents

<i>Chapter 1:</i>	1
<i>Introduction</i>	1
1.1 A Comparison of Heuristic Algorithms with Traditional Problem Solving Methods.	4
1.2 The mechanics of a Genetic Algorithm.....	6
1.3 The mechanics of a Simulated Annealing Algorithm.....	8
1.4 Combining Heuristics and Message Passing.....	9
<i>Chapter 2:</i>	12
<i>Review of Literature</i>	12
2.1 GALIB.....	12
2.2 GAUL.....	14
2.3 PGAPack.....	15
2.4 JGAP.....	16
2.5 JGAL.....	17
2.6 Galopps.....	17
2.7 GA.....	18
2.8 ParSA.....	19
<i>Chapter 3:</i>	20
<i>The Heuristic Framework System</i>	20
3.1 Genetic Algorithms.....	20
3.1.1 The Genome.....	20
<i>Mutation</i>	22
<i>Fitness</i>	22
<i>Schema</i>	23
3.1.1.1 JGenome.....	25
3.1.1.2 JByteArrayGenome.....	26
3.1.1.3 JCharArrayGenome.....	27
3.1.1.3 JDoubleArrayGenome.....	28
3.1.1.4 JSchema.....	28
3.1.2 The Population.....	29
<i>Quick Sort</i>	29
<i>Roulette Wheel Selection</i>	30
<i>Tournament Selection</i>	31
<i>Random Selection</i>	31
<i>Rank Selection</i>	32
<i>Linear Scaling</i>	32
<i>Sigma Truncation</i>	33
<i>Power Law</i>	33
<i>Asexual Crossover</i>	34
<i>Sexual Crossover</i>	36
<i>Smart sexual crossover</i>	36

<i>Two points sexual crossover</i>	37
<i>Replace the Worst Individuals</i>	38
<i>Replace Randomly</i>	38
<i>Replace the Lower Closest Individual</i>	38
<i>Replace the Parents</i>	38
<i>Replace the Best Between Parents and Children</i>	39
<i>Serial and Parallel Fitness Evaluation</i>	39
3.1.2.1 <i>JPopulation</i>	40
3.1.3 <i>The Algorithms</i>	43
<i>The Genetic Algorithm</i>	43
3.1.3.1 <i>JGeneticAlgorithm</i>	45
3.1.3.2 <i>JSimpleGeneticAlgorithm</i>	46
3.1.3.3 <i>JDemeGeneticAlgorithm</i>	46
3.1.3.4 <i>JIncrementalGeneticAlgorithm</i>	48
3.1.3.5 <i>JSteadyStateGeneticAlgorithm</i>	48
3.2 <i>The Simulated Annealing Algorithm</i>	50
<i>The Annealing Function</i>	50
<i>Transition Factor</i>	51
3.2.1.1 <i>JAnnealingAlgorithm</i>	53
3.3 <i>The Message Passing Interface</i>	54
3.3.1 <i>The MPI Server</i>	55
3.3.2 <i>The MPI Client</i>	56
3.3.3 <i>The MPI Buffer</i>	57
3.3.4 <i>MPI</i>	57
3.3.1.1 <i>MPI_Server</i>	58
3.3.1.2 <i>MPI_Client</i>	59
3.3.1.3 <i>MPI</i>	59
.....	60
<i>Chapter 4:</i>	62
<i>Case Study</i>	62
4.1 <i>The graphical user interface: The Wizard</i>	62
<i>The Simulated Annealing Algorithm</i>	64
4.2 <i>Testing the framework: The ULSI fault detector</i>	70
4.2.1 <i>Testing the Algorithms</i>	70
<i>The Genetic Algorithm</i>	70
<i>The Simulated Annealing Algorithm</i>	71
4.2.1 <i>The ULSI Fault Detection Program</i>	71
<i>Stuck-at Faults</i>	72
<i>Fault Equivalence</i>	72
4.2.2 <i>The Fault Simulator</i>	73
<i>Serial Fault Simulation</i>	73
<i>Parallel Fault Simulation</i>	74
4.2.3 <i>Simulating the Algorithms</i>	75
4.2.4 <i>Simulating the Simulated Annealing Algorithm</i>	75
<i>Chapter 5: Conclusion</i>	85
<i>Appendix</i>	87
<i>References</i>	88

List of Figures

Figure 1 Roulette Wheel Selection.....	31
Figure 2 One Point Asexual Crossover	35
Figure 3 Asexual switch Crossover.....	35
Figure 4 Two points Asexual Crossover	35
Figure 5 One Point Sexual Crossover	36
Figure 6 Two Points Sexual Crossover	37
Figure 7 The Genetic Algorithm Flowchart	43
Figure 8 The Simulated Annealing Algorithm Flowchart.....	52
Figure 9 The Single-Program, Multiple Data model [1]	54
Figure 10 The wizard's first screen.....	63
Figure 11 The wizard's second screen.....	63
Figure 12 The type of the solution	64
Figure 13 S.A. general properties	65
Figure 14 The Simulated Annealing last screen	65
Figure 15 Genetic Algorithm properties.....	66
Figure 16 The Simple Genetic Algorithm	67
Figure 17 The Incremental GA.....	67
Figure 18 The Steady-State Algorithm.....	68
Figure 19 The Deme GA	68
Figure 20 The Wizard's last screen	69
Figure 21 Parallel Fault Simulation [15].....	74

List of Tables

Table 1 Sample Genome with their corresponding fitness values	31
Table 2 SA parameters	75
Table 3 SA Test results	76
Table 4 Simple GA parameters.....	77
Table 5 Simple GA test Results.....	78
Table 6 Incremental GA	79
Table 7 Incremental GA test results.....	79
Table 8 Steady State GA parameters.....	81
Table 9 Steady State GA test results	81
Table 10 Deme GA parameters.....	83
Table 11 Deme GA Test Results.....	84

Chapter 1:

Introduction

Heuristic algorithms have been extensively used in solving combinatorial problems based on natural selection and natural genetics basics. These heuristic algorithms have proved to increase their efficiency in case the parallel execution, which would eventually introduce new information to every running algorithm through the use of data migration.

Genetic algorithms are based on the natural selection theory, which states the following:

”Natural selection is the process by which favorable traits that are heritable become more common in successive generations of a population of reproducing organisms, and unfavorable traits that are heritable become less common. Natural selection acts on the phenotype, or the observable characteristics of an organism, such that individuals with favorable phenotypes are more likely to survive and reproduce than those with less favorable phenotypes. If these phenotypes have a genetic basis, then the genotype associated with the favorable phenotype will increase in frequency in the next generation. Over time, this process can result in adaptations that specialize organisms for particular ecological niches and may eventually result in the emergence of new species... The term was introduced by Charles Darwin in his groundbreaking 1859 book The Origin of Species in which natural selection was described by analogy to artificial

selection, a process by which individuals with traits considered desirable by human breeders are systematically favored for reproduction.” [19]

The concept of Genetic algorithms was first introduced by John Holland in 1960, from the University of Michigan, and was proved to be very efficient in both explaining adaptable natural systems and creating one of the first artificial science systems.

Another type of heuristic algorithms, simulated annealing, has also proven its efficiency in solving global optimization problems. The algorithm’s name: “simulated annealing” is taken from a process used in production where the crystal is shaped by varying the temperature, in order to get the final shape at the ambient presumably low room temperature. The idea of using this technique in solving combinatorial problems was first presented by S. Kirkpatrick, C.D. Gelatt and M.P. Vecchi in 1983 [20]. The Simulated Annealing algorithm works by performing a random search around the current solution, and if this search resulted in finding a better solution, then it would replace the current solution with a certain probability that depends on a current global variable called the temperature. The global variable “temperature” is directly proportional to the probability of choosing an individual, in such a way that when being very large, the probability of choosing a newly generated solution would be high. Throughout the algorithm the temperature is gradually decreased to reach a final steady state value.

The power behind these algorithms lies in their simplicity compared to other algorithms that require heavy programming. One very important feature of these algorithms is the possibility of their application to different problems. In fact, the next important feature is the option of using these algorithms in more than one search

environment, which could include logic circuits, or random search algorithms which have a predefined initial population that should be ordered in a specific way (traveling salesman problem). Additionally, these algorithms inherit the ability to recover from a newly generated non-efficient solution, and can always find the path to reach an optimal solution by grouping the best parts of the previous generation and by applying random changes that will eventually introduce new efficient individuals. On the other hand, these algorithms have some limits which emerge from continuously choosing the “fittest” individuals of a population, which would cause the population to become homogeneous. This is when the migration would interfere to bring new genes to the population.

As a matter of fact, running more than one version of an algorithm on more than one machine, and performing a migration process of some selected individuals, wouldn't be an easy task to perform, unless a robust parallel programming standard was followed. If that was the case, the proper execution of the data transfer, or message passing processes, would be handled properly by the powerful standard, making sure that each algorithm would transfer his best findings with the others.

MPI, the Message Passing Interface, is a parallel programming standard that can be used to ensure the efficiency of data transfer between algorithms running on parallel machines. MPI does not necessarily operate on a cluster of computers, rather an interconnected network of computers can run MPI based programs since the basics of MPI are built on sending and receiving messages through a common bus, which can be in this case a network.

MPI was driven by the need of increased performance which was done by dividing the program into pieces and running every part on a machine, in parallel, hence the

tremendous time reduction and power. In fact, MPI offered an alternative way to create cheap supercomputers, by grouping heterogeneous workstations and using this parallel standard as a means of communication.

Briefly, the combination of Heuristic Algorithms and MPI, would result in a powerful system that would be able to solve any problem with ease and efficiency.

1.1 A Comparison of Heuristic Algorithms with Traditional Problem Solving Methods

Prior to the use of the heuristic algorithms, the traditional ways that were used to sweep through a search space were based on either 1- abusively checking all the items in the search space, 2- mapping the problem to an equation and trying to solve it, or 3- randomly checking the items in the search space in order to find an optimized solution.

1. The first approach, which consists of listing all the items in the search space and abusively checking them for an optimized solution, is based on a simple consecutive method. These exhaustive search algorithms are usually contained in a limited and narrow search space, however, their application on the problems that have an almost infinite search space would prove to be very inefficient. Going through every member of the search space might require dedicating a large amount of time, even with the use of the fastest existing parallel machines. One important algorithm that uses this search technique is the *dynamic programming approach*, that requires a very large space to run, since it lists all the possibilities for the provided problem.

2. The second approach works by trying to model the problem as an equation, and solve it. One method that is used in an attempt to solve this formula operates by generating the derivative of this equation, setting this derivative equal to zero and finally solving the resulting equality. This equality is solved by slowly approaching the peak through finding solutions that are closer to the peak. This method follows the notion of hill climbing, but it still faces the problem of being stuck at a local minimum, thus never approaching a better solution. Calculation based methods proved to be inefficient since not all the non deterministic problems can be described by a single equation, they are rather described by complicated equations whose derivative is very hard to generate, or sometimes impossible to calculate, in case the equation is non-continuous or noisy.
3. The third approach is based on a blind random walk in the search space. In case the search space was very large, the probability of this random search to locate an efficient solution would be almost equal to zero. This random search method was used as a last way out, to try to come up with a solution, when all the other deterministic search methods failed. The obtained results from the random search were not satisfactory and the search proved to be ineffective. As for the heuristic algorithms, the use of randomness is not in the searching method, but in the random choice that is guided by a search procedure, which converges towards an efficient solution through the natural selection rules.

Eventually, heuristic algorithms, have proved, that they are highly efficient reliable and robust, which are features that do not exist in the deterministic methods that were previously used to solve combinatorial problems.

As a final point, seeking efficiency was the drive behind finding new and better ways to perform the search methods, and the definition of efficiency is not simply to find a good solution, since the conventional methods might serve to find a good solution. Efficiency is rather finding the best solution with the limited provided resources such as time and space, and this is another strength of the heuristic algorithms: providing an optimum result with the limited allocated resources.

1.2 The mechanics of a Genetic Algorithm

To solve any type of problem using genetic algorithms, some procedures should be followed which usually consist of creating a population and determining how to evolve it. Following are the steps that should be followed prior to the use of a genetic algorithm.

The first step is to model the solution as a chromosome, which should vary over a limited range. In other words, the solution should be described as a limited sequence of items that belong to a homogeneous collection of items. For example, in solving the traveling salesman problem, the solution consists of the cities that can be visited once. Another example is in the VLSI circuits, where the solution is modeled as a string of Binary numbers.

The second step is to create a fitness function for the problem. A fitness function is the means for the genetic algorithm to find out if the optimization is going in the right direction. In other words, the fitness function is an important part of the algorithm which operates by obtaining a solution and deciding how good it is, or how close it is from the best solution. This decision can be achieved by performing a

simulation of the given problem, and checking how close the solution is to the optimal solution.

A population is a group of candidate solutions that are randomly determined. The population has many properties, that include the size, which can be greater than or equal to one individual, and it is usually a constant. In fact, there are some selection methods that are applied to the population, that remove the non-efficient individuals, which in turn resize the population. There is a possibility of many choices that can be taken when selecting individuals from a population for deletion, and the most widely used choices are either based on selecting the worst individuals, selecting the parents of the newly generated individuals, or simply based on a random selection among the population's individuals.

To create a new generation, a group of individuals should be selected from the current population. There are many selection methods, but the most commonly used techniques are the following: random, roulette wheel, tournament and rank of the fittest individuals. These selection techniques will be described in details in chapter 3. The algorithm will next apply the genetic operators: Mutation and crossover, on the selected individuals. Mutation works by randomly changing a part of a genome (gene), to a random value that falls within the predefined range. Crossover works by performing a mix to the genes of the genome(s) that were selected. These two genetic operators will be discussed later in chapter 3.

This process of changing the population, through selection, mating and replacement can be denoted as the process of evolution. This evolution process will continue to run until the algorithm terminates, which might happen in one of the following cases:

- The number of new generations has reached a predefined maximum.
- The maximum predefined fitness was reached.
- A combination of both methods, which states that if a solution having the maximum fitness was reached, it would stop. If that weren't the case, then the algorithm would continue the evolution process until reaching the maximum number of new generations.

Finally, there exists many combinations of the previously stated features, and each combination is considered as a variation of the genetic algorithms, knowing that each algorithm is optimized for a specific problem type.

1.3 The mechanics of a Simulated Annealing Algorithm

Another commonly used search technique in finding an optimal solution is simulated annealing which uses random processes to help guide the search until reaching a minimal state of Energy.

To solve any type of problems by using the Simulated Annealing, three main things have to be considered: assigning a global temperature and deciding on how and when to decrease its value, and creating an initial solution and a corresponding fitness function, which decides how "fit" is a certain solution.

Initially, a preliminary solution can be randomly generated, since any point of the search space can be selected as a starting point. The progression of the algorithm will make sure that this initial solution will converge to an optimum one.

An initial Energy state also has to be created which will decide on the probability of a newly generated solution to replace the current one. In fact, the power behind this algorithm lies on the possibility of a bad solution to be selected, which would help the algorithm to escape being stuck at local minima.

Applying some randomizing operators to the current solution, derives the newly generated solutions that would eventually replace the current solution. As for the temperature, which denotes the current energy state of the system, it is gradually decreased in order to reach the final steady energetic state. This way, the system will check a large searching area before converging to a low energy state, and narrow search space where, at this point, it would use hill climbing to reach the peak, or the optimal solution.

When the Energy of the system reaches the minimal assigned value, the algorithm will stop iterating and will display the fittest solution. The Simulated Annealing algorithm will be discussed more thoroughly in chapter 3.

1.4 Combining Heuristics and Message Passing

The main idea behind creating MPI, was to create standards for message passing systems, that can be implemented using any programming language. This idea was very successful and lead to the spread use of this standard, especially that some of the translations became available online free of charge. The early versions of MPI were

written using the C and Fortran programming languages. Later on, after the creation of the Java programming language, some versions of JAVA MPI were developed.

The declared standard did not state how the MPI process should be created and started, and left this task to be fulfilled in the implementation step. What was defined in the standard was the fact that every newly created process should have a rank id. This ID is generated by the process that has a rank equal to one. Another requirement, was to declare all the processes in a communicator called MPI_COMM_WORLD, where every process will have an associated rank value in an object called the communicator.

MPI's primary concern was to pass messages between the processes, and this is why the standard required all the 'send' and 'receive' methods to begin with the letters MPI_ to be differentiated when being imported to other programs.

There are over a hundred methods that were defined in the MPI standard, but to create an efficient program, only six or a bit more methods can be used.

The various methods that will be used to transmit messages between the running heuristic algorithms, will be based on the non blocking routines which allow the next statement to be executed whether or not this routine was locally complete [1].

The implementation of the MPI standards can be written in JAVA with the help of the native construct RMI. The benefits of the use of RMI are in the possibility of referring to a remote object's methods. Every RMI program is based on a client-server model, where every client has an interface that holds a signature of the remote methods of the server. When compiled and linked, a client stub and a server skeleton are created in the RMI registry whose role is to manage the remotely available objects and the Naming Services. When a client calls a remote method, the server object is first located

in the registry, then a client stub is returned to the client, holding information on how to use the parameters of the server [11].

The non blocking sending and receiving methods are called MPI_Isend and MPI_Ireceive, where the letter I stands for the word immediate. These two methods will fork and try to send or receive a message, and their completion can be detected through the use of two other methods MPI_Wait and MPI_Test, which return with a flag indicating whether the operation has completed at that time [1].

The algorithm that supports parallelization can be run on many machines in parallel or on a clustered machine. This heuristic algorithm can use any standard to establish the communication between the parallel running algorithms, including the message passing algorithm which can be easily fitted in this algorithm, due to its simple standards and basic requirements. Thus, when an algorithm discovers a new solution that it considers as very efficient, it can use the MPI standard to wrap the solution that was found in a message and send it over to the destined receiver. This migration process will introduce new unexplored solutions to all the participating members, which would eventually lead to an increase in efficiency.

Chapter 2:

Review of Literature

Previous implementations that proposed Heuristic Algorithm libraries, were written in various programming languages including C, C++, Fortran, Delphi and Java. The rest of this chapter describes all the available implementations, and states their features and capabilities.

2.1 GALIB

The GALIB library, was developed in 1996 by Matthew Wall at the Massachusetts Institute of Technology, under the Mechanical Engineering Department [1]. GALIB, which stands for the Genetic Algorithm Library, is a library of genetic algorithm objects that was described by its author as a collection of “tools for using genetic algorithms to do optimization in any C++ program using any representation and any genetic operator” [2]. This library was amongst the first trials to create a portable library of genetic algorithm components destined for general usage. In fact his library’s general features include the incorporation of many examples, which illustrate the use of its embedded components, in addition to its own random generators which are crucial to the search algorithms’ operation. Furthermore, GALIB uses some templates in the genome classes, which can be removed or disregarded in case the compiler does not support them, or in case the programmer doesn’t wish to use them.

GALIB also included some other important features distributed over the algorithm design, the system parameterization and the statistical methods. As for the algorithm design, this library included the support four different types of algorithms which differ in the replacement strategies, the evolution processes and in the possibility of parallel execution.

The first algorithm is the simple genetic algorithm which uses non-overlapping population and optional elitism. In other words, the newly generated population will replace the old one with the possibility of copying the best individual from a population to the other.

The second algorithm is the steady-state genetic algorithm which uses overlapping populations, with the ability for the user to define the percentage of the population that should be replaced. This algorithm's efficiency resides in the ability of it being highly parameterized where the user will shape the algorithm following his needs.

The third variation is the Incremental Genetic Algorithm whose population consists of only two individuals, and that contains custom replacement methods such as replace parents, replace worst, and replace random.

The final algorithm is the deme GA which uses a steady state algorithm with the possibility of parallel execution [1]. The parallelization of the algorithm depends on the PVM library that should be pre-installed on the multiprocessor system.

All these features are available, in addition to built in selection strategies, such as Roulette wheel selection, tournament selection, random selection, stochastic uniform sampling, and deterministic sampling [1]. As for the termination methods, they include convergence and number-of-generation termination. All these features, in addition to

some extra embedded ones, will be discussed thoroughly in the next chapter since they will all be used in the local implementation.

2.2 GAUL

The GAUL library which is an open source library that was released under the GNU General Public License, was developed in 2001 by Stewart Adcock under the C programming language. “It provides data structures and functions for handling and manipulation of the data required for serial and parallel evolutionary algorithms” [3]. GAUL was developed on Linux, but the code can be compiled on any “POSIX compliant system” [3].

This interpretation of the genetic algorithm library also contains some important features to make it a general purpose customizable library.

GAUL also supports a powerful possibility of parallel execution of the genetic algorithm. In fact the parallelization can be performed using any of the most famous parallel architectures ranging between MPI, Open MP, P Threads, and the forked-process model which should be previously installed on the system. Some of these parallel implementations leaves the task of multi population evolution to be handled by the operating system. So in case it was installed on a cluster machine, it would run as expected and offer optimized results. Some of GAUL’s other important features are the following:

- Different Heuristic algorithms were included such as tabu search, simulated annealing, simplex search, deterministic crowding, differential evolution, steepest ascents, and hill climbing.

- Support for many genome data types.
- Different steady-state and generation based GA can be selected [3].

Finally, the main worry of the author of this library was to create a library that can compile and run using either C or C++, or the advantage of C is that it is trivial to use from within C++ [3].

2.3 PGAPack

The PGAPack, the Parallel Genetic Algorithm Library, was developed by David Levine in 1996 in the ANSI C programming language [4]. This library was created due to the contribution of a large team of professionals that originate from various scientific research institutes or programs such as Argonne's Science and Engineering Research program and the U.S. department of Energy.

This library can be used as a package which can be called from either a FORTRAN or a C program. It is customizable in a way that users can create new evolution operators and data types. "PGAPack is a parallel genetic algorithm library that is intended to provide most capabilities desired in a genetic algorithm package in an integrated seamless and portable manner" [4].

PGAPack uses the MPI library to run its parallel version, but it requires the already existing installation of the MPI environment. The MPI support will be included upon installation, where the user has to specify the location where the files are located.

PGAPack includes some beneficial powerful features which facilitates the process of embedding it in other projects. Some of these features are the following:

- Object-Oriented data structure design

- Parameterized population replacement strategies varying between two major possibilities: the Generational replacement, and Steady-State replacement schemes.
- Multiple choices for evolution operators varying between mutation and crossover.
- Easy integration of hill-climbing Heuristics
- Multiple stopping criteria which are the following: iteration limit exceeded, population too similar, and no change in the best solution found in a given number of iterations [4].
- Multiple selection schemes which are: proportional selection, stochastic universal selection, binary tournament selection, and probabilistic binary tournament selection [4].

In addition to being available free of charge on the internet, the user of this library can download a fully documented user guide which was a part of a Ph.D. study.

2.4 JGAP

The JGAP, Java Genetic Algorithm Package, was developed by Klaus Meffert in 2002 under the Java programming language. It was designed in a “modular” way so that users would be able to customize it depending on their needs. JGAP’s motivating idea was to create a program that would help in designing a GP (Genetic Programming) application which is a similar algorithm to GA. Knowing that GP is the process of evolving Programs, this library was mainly designed for GP Programs developers, in which a program evolves to becoming an optimal one having all the required features.

In other words, “the main philosophy that was setup before extending JGAP towards GP, was to reach the goal by adding parts to GA to get GP” [6].

JGAP is a practical easy-to-use library where the user simply defines the traditional genetic options and executes the algorithm without the need for complex setup steps.

JGAP does not support multithreading or distributed evolution, but the author promises a future version that embeds parallelism.

2.5 JGAL

The JGAL (Java Genetic Algorithm Library) is also created under Java and it features the basic genetic principles in the evolution theory. It was created by Janusz Rybarski in 2006 and as stated by its author: “Java Genetic Algorithm Library is a set of classes and functions for design and use Genetic Algorithm” [5]. This library is a very simple and limited implementation since it only supports chromosomes which are coded as binary numbers. The selection procedures are also limited and it doesn’t support parallelism.

2.6 Galopps

The “Galopps” library is created by Erik Goodman in 1997 in the C programming language. “It was based upon Goldberg's Simple Genetic Algorithm (SGA) architecture, in order to make it easier for users to learn to use and extend.” [10].

This library supports parallel programming through the use of the PVM standards, and has some other interesting features that are briefly explained below:

- Various traditional selection methods varying between roulette wheel, stochastic remainder sampling, tournament selection, stochastic universal sampling, and linear-ranking-then-SUS.
- Parameterized Genomes in the initialization and evolution stages.
- Various genetic operator including mutation (fast bitwise, multiple-field, swap and random sublist scramble) and crossover (1-pt, 2-pt, and uniform)
- Support of different fitness scaling techniques which include: linear scaling, Boltzmann scaling, sigma truncation, window scaling, ranking.
- Different selection for replacement algorithms including DeJong-style crowding replacement, parent replacement, and random replacement.
- Optional Elitism.
- Many Convergence techniques: "lost," "converged," "percent converged," & other measures.
- The definition of different representations in subpopulations, with the optional migration possibility.

All these feature in addition to some other capabilities make Galopps a rich library, since it was built on one of the best detailed GA books.

2.7 GA

The “GA” library is created by Jeff S Smith in the year 2000 under the Pascal (Delphi) class library. It is a limited library with no multithreaded support and a limited selection methods (only Random). It is best described by its author as follows:

“The genetic algorithm (GA) is basically a computer program which simulates evolution. Namely, a simulated population of chromosomes is randomly created and allowed to reproduce according to the laws of evolution with the hope that a very fit individual chromosome will eventually result” [8].

There are also other programs that were developed as a Genetic Algorithm library which will only be mentioned but will not be described in details. These libraries include the Jaga library (Java Genetic Algorithm Package) which was developed by Jan Koutnik in the year 2000 [9], and the GA Playground which was created by Ariel Dolan [7].

2.8 ParSA

The “ParSA” is created by G. Kliewer, and S. Tschoke, in the year 2000. It is an object-oriented simulated annealing library based on C++, which uses the MPI message passing interface [21]. The library is based on object-oriented programming and it can be easily extended with new features ranging from new cooling schedules to different acceptance methods. This library requires the existence of an MPI environment on the workstations, and it was tested on many MPI software such as MPICH, ScaMPI and WMPI. parSA’s performance was demonstrated by applying it on the Weekly Fleet Assignment Problem (FAP) which is an optimization problem that occurs in the process of operating an airline.

Chapter 3:

The Heuristic Framework System

This chapter proposes in details the work that creates a portable and distributed message passing library to solve combinatorial optimization problems. The work was done using the Java programming language, and makes use of the native construct Remote Method Invocation (RMI). This chapter is divided into three parts. The first two parts discuss the Heuristic algorithms that are included in this framework: the genetic algorithms and the simulated annealing algorithms. As for the third part, it discusses the Message Passing Interface.

Note that since Java is an Object Oriented language, this framework relied heavily on object oriented programming throughout the coding of all its elements. In addition, five packages were created, where each package holds all the related classes.

3.1 Genetic Algorithms

The discussion about the Genetic algorithms includes three major components which are the genome, the population, and the genetic algorithm itself.

3.1.1 The Genome

The Genome was previously defined in chapter one as being the result of encoding the solution in a chromosome. All the genome-related classes were grouped in one package called Genome, in order to ease the debugging and extension procedures.

Deciding on the type of the genome object, is one of the problem specific decisions that should be specified by the user. John Holland, in his initial study, relied on binary encoding and formulated all the theory based on that simple type. Evidently, this type of encoding has limited domains of application and most of the combinatorial problems need some rich encoding capabilities to be able to use this natural selection theory. The mostly used encoding scheme is the array of characters or strings, since the alphabet letters can denote a large number of choices ranging from city names (Ex: Traveling Salesman Problem), to teachers names (Ex: class scheduling problem). Another type of encoding is the tree encoding, but since a tree can uncontrollably grow, it is recommended that this type should not be used unless the user can have control over the size of the trees.

There are three possibilities of encoded genomes included in this library, but the user can add his own genome implementation. The possible genome types are Double, Character, and Byte.

The basic components that are necessary to create a genome object are the fitness function and the array of genes. In this study, we decided to add to the genome an essential evolution operator which is the mutation. The drive behind that decision came from the will to include relevant code inside every class, and to simplify the whole evolution process. And since the mutation factor performs a mathematical operation on the genes, it would be better to include this method in the genome so that it would run faster.

Mutation

Mutation is another word for change, alteration, or transformation. It is an essential genetic operator in the evolution process because it helps in discovering new genes that cannot be generated by other operators. The concept behind mutation is somewhat simple given that all what this operator has to do is to change a certain gene (which could be randomly selected) to a random value. Mutation cannot be used very frequently since it can create changes in the genome to a point where it would become a useless one. On the other hand, when a search process is stuck at a local peak and cannot find a way to escape it, the best choice at that time would be the use of mutation. The mutation probability is usually set to a low value, in a genetic algorithm, which should be increased as the evolution process progresses. However, if a mutation leads to the introduction of a deficient individual, the algorithm would have the means to decrease its bad effect on the evolution throughout the selection.

Fitness

Fitness is an objective function that measures the goodness of the solution that is embedded in the genome. Evaluating a genome's fitness doesn't necessarily rely on the contents of the genes, but should in some cases take the order of the genes into consideration. The fitness function plays a very essential part in the evolution process, since the algorithm's selection methods are founded on the fitness value of each individual. Deciding on an individual to contribute in the next generation according to his fitness value means that the selection process is directly proportional to the fitness.

Schema

Schemas (or Schemata) can be used to optimize some of the evolution methods such as crossover and mutation. Theoretically, a schema is an approach of resemblance between two genomes. A schema is simply an array that has the same size of a genome, and it is created by selecting the different genes, that have the same index position, of two chromosomes and replacing them by a '*' in the schema's array. As for the common genes, they will be copied as they are. For example if we have two genomes: 101001 and 100011, the corresponding schema would be the following: 10*0*1. As a result, this schema can represent four different genome formations: 100001, 100011, 101001, and 101011. A schema will have a lower contribution to the evolution process when the genes have a large range of values, but their role would be extremely efficient in the limited range genes (Ex: binary).

A schema has some functions that explain its properties, these properties are: the Order and the Delta of the schema. The Order is simply the number of fixed genes in a schema, and the Delta is the defining length of a schema which is the maximum distance between the first and the last fixed genes. The Order value varies between 0 and the size of the genome, and the Delta value varies between 0 and the genome size -1.

The schema's role in the evolution algorithm is mainly in the trial to preserve the good genes in a genome and keep them close together.

Since a mutation's percentage is relatively low, the need of taking an advantage of it is critical. Hence, a mutation shouldn't select a star (*) and mutate it, since a star indicates that there exists multiple values for this gene. So mutating a

“fixed” gene must have a greater influence over the fitness of the chromosome, which might be beneficial in many circumstances, especially to escape local peaks.

Schemas are also useful for crossover, in a trial to keep the common genes closer together. The analysis will include the interpretation of the Order and the Delta values at the same time. Therefore, when the Delta’s value is very close to the Order value, (Ex: `***110***` Order=3 Delta=2) it means that the schema has close significant genes and rarely separated by stars. This type of genome should transfer these grouped significant genes to the next generation because the fitness is directly affected by the presence of these fixed genes. If the Delta value is much greater than the order (Ex: `1*****0` Order=2 Delta=8) then we can conclude that the genome’s fixed genes are scattered. The more the Delta value is close to the Order value, the closer is the distance between the fixed genes, and the higher the probability of this genome to be selected for the next generation. This analysis can affect our decision on the choice of the crossover point of the genome, which should have a lower probability of intercepting the grouped genes. In case a fitness value of a schema was higher than the average fitness of the population then the probability of this schema to survive to the next generation should be high.

The Genome package contains all the classes that are related to the genome component, including the schema class and the initial genome class JGenome. The next sections discuss the contents of this Genome package.

3.1.1.1 JGenome

JGenome is an interface which includes all the mandatory methods and variables necessary for the creation of a Genome object. This inheritance feature is necessary for the program to support more than one type of genomes. In addition to the implementation of some methods in this interface that are important for all the genome classes to have.

To add a new Genome class, the user needs to create a class in the genome package and to have it implement the JGenome interface, and create all the predefined abstract methods. Some other minor additions will be mentioned later in the JPopulation class, which require the need to add a couple of lines that deal with object initialization.

The most important defined variables and methods in this interface are the following:

- `Vector genes`: this variable holds the array of genes which is the basic building block of the Genome.
- `double absFitness`: this variable holds the current value of the fitness of the genome object.
- `double scaFitness`: this variable holds the current value of the scaled fitness of the genome object. The method used to generate the value of the scaled fitness will be explained in section 3.1.2.
- `Object program`: this variable is of type Object because it is supposed to point to any object that is created in the algorithm, and it was created in a direct relation with the fitness function. To make use of this variable, the user has to have a fitness function created in his original program. The next step would be

to pass his object `program (this)` as a parameter to the Genetic Algorithm object which would route this parameter to the `JGenome` class. This way the user will be able to call any function written in his program, from the fitness function of the genome. This option was created to make it easier for the users to profit from this framework, in case they had already created their program and do not wish to do significant updates to their code.

- `double fitness()`: this method should be implemented by the user, since this is where the differentiation process of the genomes occurs depending on the subjective evaluation of the user. As we mentioned before, the user can create his fitness function in his original program and call it in the body of this fitness method through the use of the `program` variable.
- `void mutate(int idx)`: this function applies the mutation operator on the genes. The parameter `idx` denotes the location of the genome that needs to be mutated. In case the user wishes this location to be a random one, he should pass as a parameter the static global variable `RANDOM_INDEX`.

3.1.1.2 JByteArrayGenome

`JByteArrayGenome` is a class that implements the interface `JGenome`. Its genes are of type `byte` which is a small 8-bit integer type ranging from -128 to 127. this datatype is useful when dealing with data streams or raw binary data [11]. Since we talked about all its features when describing the `JGenome` interface, we will only mention the constructor.

```
public JByteArrayGenome(int geneSize, byte lowB, byte upB, boolean
randomize, int seed, Object prog)
```

The parameters that are passed to this constructor are the following:

`geneSize`: which defines the total fixed size of the gene.

`lowB`, `upB`: these two variables define the boundaries of the datatype variation. When defined, any mutation will only fall within the range that is defined, making sure that the genes would still be relevant to the user's original definition. For example, if using this datatype for a binary representation of a logic circuit, the boundaries should be set to zero and one.

`randomize`: this Boolean variable indicates to the genome, upon creation, if it should initialize the genes to random values or not.

`seed`: this is a seed value that is passed to the genome's random variable.

`prog`: this is where the program pointer is passed to the genome, to be used in the fitness function.

3.1.1.3 JCharArrayGenome

JCharArrayGenome has genes of type `char` which is a 16 bit integer type. This datatype uses Unicode to represent the characters, including a multitude of character sets ranging from Arabic to Greek to Latin and many more. The range of a `char` is between 0 and 65'536 [11] and it does not include negative values. The constructor for this class is very similar to the JByteArrayGenome class including the parameters. The only exception is for the datatype of `lowB` and `upB` which in this case are `char`.

3.1.1.3 JDoubleArrayGenome

JDoubleArrayGenome has genes of type **double** which uses 64 bits to store a number. The double datatype can be used for problems that require extreme accuracy over many iterative calculations [11]. It can also be used in mathematical applications to calculate the area of a circle for example, or to calculate a derivative value. This datatype has two constructors, one of which is similar to the previous class's constructor, and the other has an extra parameter added which is `stp`. This parameter defines the width of the fraction part of the double number.

3.1.1.4 JSchema

As explained earlier, JSchema is not a class that implements the interface JGenome, it is rather a tool used to find similarities between two genomes. The way JSchema objects are created will be discussed later in section 3.2. The important functions and variables in this class are the following:

- `String genes`: which is the array that holds the significant genes and the stars.
- `int order, int delta`: are the variables that hold the value of the Order and Delta.
- `int fixedPos`: this variable contains the index of the first significant gene.
- `Vector stars`: this array contains all the locations of the stars in the schema's genome.

This class's constructor is:

```
public JSchema (JGenome g1, JGenome g2, Object prog)
```

where `g1` and `g2` are the two genomes that form the schema.

3.1.2 The Population

The population class is where the evolution process takes place. The most important role of the population object is that it defines the current container for the genomes, and applies the evolution operators to generate a new population.

Upon its creation and initialization, the population object has all the required data it needs to perform a single evolution process, but it still lacks the engine that will call all these methods that will make the evolution proceed, this engine is the Genetic Algorithm.

After being created, the population object will get initialized and will verify that it has the proper number of genomes which can either be supplied by the Genetic Algorithm, or generated locally using a random algorithm. The next step would be to start the evolution by first calculating the fitness then sorting the genomes of the current generation by using the quick sort algorithm.

Quick Sort

Sorting the genomes in the current population is an essential step that should be performed before any evolution takes place. This step is important because all the later evolution operators will be applied with the assumption that the population is sorted, and will select the first members to be the genomes with the highest fitness value.

Quicksort, one of the fastest sorting Algorithms, was developed by C. Hoare in 1962. The way this algorithm works in sorting an array, is by choosing a random element from the array and distributing the smaller elements to its left and the bigger elements to its right. Then a recursive call on this same method will be initiated on the left sub part,

then on the right one. When a set has less than two elements, the recursion stops, thus ending the procedure [13]. The benefits from this sort is that it uses a function called swap which swaps the position of two elements in the array, minimizing the amount of space required.

Quicksort's worst-case running time is $\Theta(n^2)$ on an input of n numbers [14], but it is often the best practical choice for sorting because it is remarkably efficient on the average [14] with an expected running time of $\Theta(n \lg n)$.

After sorting the population, the Algorithm will select the exact amount of individuals from the current population, clone them, and then apply the evolution operators on them. Most selection methods are based on the Darwinian's survival of the fittest theory. Of these methods we state the following: Roulette Wheel Selection, Tournament Selection, Rank Selection, and Random Selection.

Roulette Wheel Selection

The Roulette Wheel Selection method is based on the fitness value of each individual in relation to the total sum of the individuals' fitness values. Therefore the higher an individual's fitness value, the higher the probability of it being selected. This method starts by summing all the fitness values for all the population while keeping track of every individual's contribution to the sum. The next step is to generate a random number that varies between zero and the total calculated sum. The system then selects the individual, in whose domain the random number was classified. In other words, the total sum of the individuals' fitness can be modeled as a wheel that is divided

into sectors where each sector corresponds to a single individual. The random number falling in an individual's sector would get that individual to be selected.

Table 1 Sample Genome with their corresponding fitness values

1101001	50
0101011	20
1101110	90
1100010	10
0001001	130

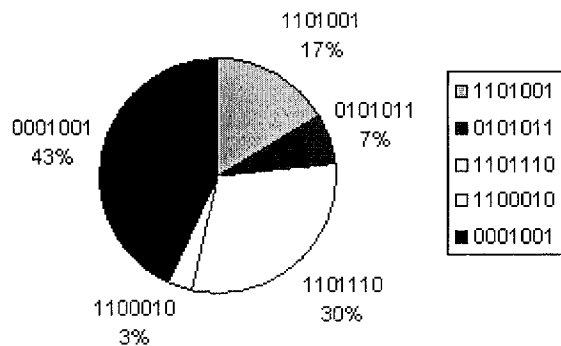


Figure 1 Roulette Wheel Selection

As an example, consider table 1 which contains various individuals that are associated to their fitness value. Fig. 1 shows each member's contribution to the wheel.

Tournament Selection

In the tournament selection, the system selects two individuals by using the Roulette Wheel Selection and then selects the one that has a higher fitness value. This type of selection makes sure that only the elite group are always selected for mating.

Random Selection

As its name indicates it, random selection treats all the individuals equally by making the probability of an individual to be selected, the same for every present member in the population. The selection method is performed by randomly generating a

number that ranges between one and the size of the population, this number would be the index of the selected individual.

Rank Selection

In this selection method, the individuals that have the highest fitness value are always selected.

But if we apply these methods on the normal fitness values, we might be faced with two different problems. The first one is fast convergence which happens towards the beginning of the evolution process. Fast convergence happens when only the individuals that have a high fitness value are selected, which would result in a complete reduction of the other members, resulting in a homogeneous population. The other problem is that in some populations, after a certain number of evolutions, the fitness value becomes almost equal for all the individuals, which will make it hard for the selection methods to choose different individuals for mating. The solution to these problems was to use a scaled value of the fitness in the selection process. The Scaling algorithms used in this study are the following: Linear Scaling, Sigma Truncation Scaling, and Power Law Scaling.

Linear Scaling

Linear Scaling is calculated from the normal fitness value by following this formula [2]:

$$f_{scaled} = a \cdot f_{normal} + b$$

where the multiplying factor a and the additive factor b are calculated by using these formulas:

$$a = (f_{Multiple} - 1.0) * f_{Avg} / (f_{Max} - f_{Avg})$$

$$b = f_{Avg} * (f_{Max} - f_{Multiple} * f_{Avg}) / (f_{Max} - f_{Avg});$$

where $f_{Multiple}$ is a multiplier selected by the user, f_{Avg} is the average value of the population's fitness, and f_{Max} is the value of the maximum fitness [16].

Sigma Truncation

The sigma truncation scaled fitness was proposed by Forrest in 1985 which suggested using the standard deviation for the calculation of the scaled value [1]. This scaled fitness is calculated using the following formula:

$$f' = f - (f_{Avg} - \sigma)$$

where f_{Avg} is the average value of the population's fitness, and σ is the population's standard deviation.

Power Law

The power law scaled fitness was proposed by Gillies in 1985 where the fitness is raised to the power of a constant k :

$$f' = f^k$$

The inventor of this scaling method used the value of $k = 1.005$, but any value can be used by the user flowing his needs.

In both linear and sigma truncation scaling, negative scaled fitness values are not allowed, so all negative values are replaced by zero.

After selecting the individuals that should mate to create the new generation, the genetic algorithm will call the evolution operators: crossover and mutation, and will initiate the evolution process. Crossover operators, differ depending on the number of genomes that should perform the crossover, and the way the crossover should be executed. From that we state the following possible crossover possibilities: Asexual Crossover, one point sexual crossover, two point sexual crossover and smart crossover which makes use of the schema class that was discussed earlier.

Crossover is the basic operator that performs most of the reproduction operation. In a crossover, the genome of the parents will be divided and mixed together, and the child will inherit different parts in a different order. The result would be new individuals that have some similarity to their parents, with a different fitness value.

Asexual Crossover

Asexual crossover is performed on one individual only by selecting a random location index on his genome and flipping the left and right parts around it. In some problems, asexual crossover is the only mating option that can be used. These problems can have their genomes previously defined and what is needed is a proper arrangement of the genes. A good example would be the Traveling Salesman problem which searches for the best arrangement of the individuals' genomes.

There are three types of asexual crossover: One point asexual crossover, asexual switch and Two points asexual crossover.

In the one point asexual crossover, a random index in the individual's genome is selected and the two parts around it are switched. In the two points asexual crossover, two random points are selected, and the upper part and the lower part around these two point are switched. In the asexual switch, two gene indexes are randomly selected and they get exchanged. The following figures explain the asexual crossover. in its three types.

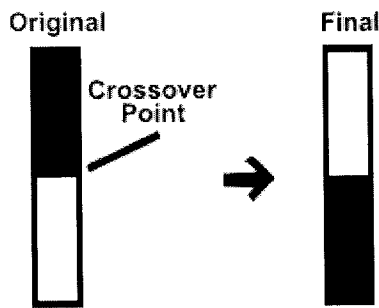


Figure 2 One Point Asexual Crossover

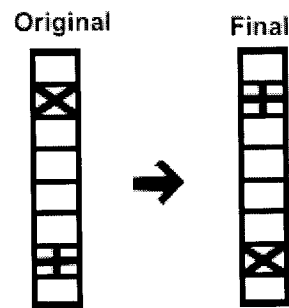


Figure 3 Asexual switch Crossover

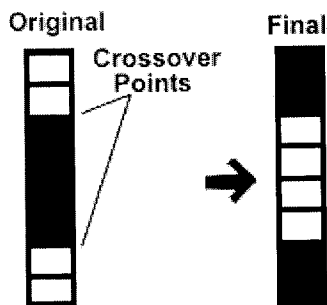


Figure 4 Two points Asexual Crossover

Sexual Crossover

In the sexual crossover, two individuals are selected from the population, then an index point is chosen, whether randomly or deterministically, to become the crossover index.

One point sexual Crossover

In the one point sexual crossover, the parents' genes that are before and after a previously specified crossover index, will be symmetrically exchanged, resulting in two new individuals as seen in the figure below.

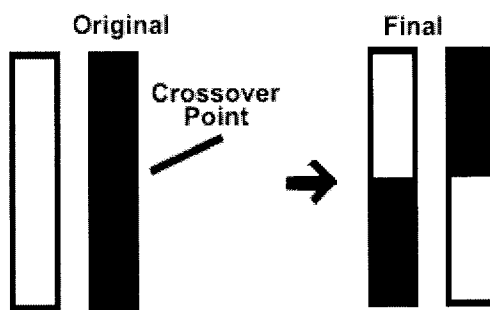


Figure 5 One Point Sexual Crossover

Smart sexual crossover

The smart sexual crossover creates a schema from the two selected parents and decides in a smart way on the exact location of the crossover point. After deciding on this point, a normal one point sexual crossover is executed. The schema was discussed in details in section 3.1.1 and 3.1.1.4.

Two points sexual crossover

In the two points crossover, two random points are selected in the parents' genome and the crossover is done in the following manner: the leftmost and the rightmost parts of the first parent are copied to the first genome child. The middle part is copied to the second child genome, and the same way is done to the second parent's genome.

The two points crossover is explained in the following figure.

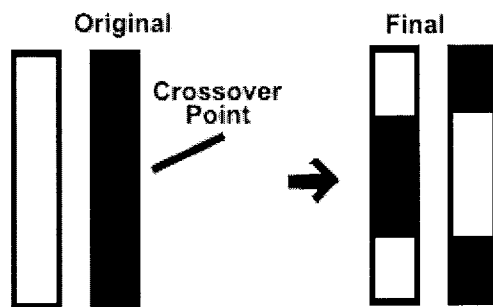


Figure 6 Two Points Sexual Crossover

After generating the new individuals, a random test is performed to check if the mutation should take place or not, depending on the percentage of mutation that was specified earlier by the user. This test is performed in the following way: a random number ranging between 0 and 100 is generated, if this number ranges within the supplied percentage value, and then the mutation operation should take place. The mutation function is very simple since it calls the mutation function that is embedded in the JGenome class, which was discussed earlier.

When the evolution operators finish executing, the new individuals are placed in a new array waiting to be inserted in the current population. There are many ways to create a space for these new individuals in the current population, and these ways are the following: Replace the worst individuals, Replace randomly, Replace the lower closest individual, Replace the parents, and Replace the best between parents and children

Replace the Worst Individuals

In this method, the worst individuals are selected from the population and marked for deletion, to be replaced by the new individuals.

Replace Randomly

In this method, a completely random number varying between zero and the total size of the population, is generated, and the corresponding individual is deleted.

Replace the Lower Closest Individual

In this method, the fitness value of the new individual is calculated and an individual with a close but lower fitness value is selected for deletion.

Replace the Parents

In this method, the parents that were selected for mating will be deleted.

Replace the Best Between Parents and Children

In this method, the fitness value of all the children is calculated, and the best two individuals amongst parents and children will be selected.

After inserting all the newly generated individuals into the current population, the Genetic Algorithm will need to calculate the fitness of these individuals, as a part of the evolution process

Serial and Parallel Fitness Evaluation

There are two methods in this system that can initiate the fitness calculation of the individuals in the current population. The user is the one who should decide which method to use depending on the fitness function that he supplied to the system.

The first method will go over every individual in the current population and will call for the fitness function to evaluate it. This method is a simple serial method that is not very efficient, especially in complicated fitness calculation functions.

The second method has the ability to supply to the fitness function, an array that contains the whole population that is destined for evaluation. In this case, the user has to supply the system with a fitness function that can apply a sort of parallel computation, which will evaluate the whole population. *(see chapter 4 for more details)*

Having explained all the theory behind the population class, it is time now to mention some of the important variables and methods that are included in this class.

3.1.2.1 JPopulation

There are three constructors for the JPopulation class. The first one creates a JPopulation object that will be used under the parallel Genetic Algorithm. The second one creates a JPopulation object destined for normal evolution based algorithms. The third constructor is used for cloning the JPopulation object, where all the variables are included in the parameters list.

```
public JPopulation(int gSize,int gStep,int crScal,int crAlgo,  
    int crType,int pSize,int crPt1,int crPt2,int gType,  
    double gUBd,double gLBd,int selType,Vector inPop,  
    float mutPerc,int fillPop,int nPSize,  
    int popNum, /*used for parallel populations only*/ Object prog)
```

`gSize` is the size of the genome.

`gStep` is related to `JDoubleArrayGenome`, it defines the Genome's precision.

`crScal` defines the method that should be used for scaling.

`popNum` is the number of the parallel populations that will be evolving concurrently.

`crAlgo` denotes the type of the Crossover Algorithm that should be followed.

`crType` denotes the type of the Crossover operation that should be performed.

`pSize` defines the size of the population.

`crPt1` defines the first crossover point, in case it was previously defined by the user.

`crPt2` defines the second crossover point, in case it was previously defined by the user.

`gType` defines the type of the genome that is followed throughout the evolution.

`gUBd` defines the upper bound value of the genome.

`gLBd` defines the lower bound value of the genome.

`selType` defines the way the individuals should be selected for mating.

`fillPop` defines the method that should be used in filling the initial Population

`nPSize` defines the size of the new population

`prog` is the pointer to the program object that was discussed in section 3.1.1.

The most important defined variables and methods in this class are the following:

- `Vector newGeneration`: this is the array that holds the current generation.
- `Vector currGeneration`: this is the array that holds the new generation.
- `int crossoverType`: this is the variable that holds the value of the crossover type that should be followed throughout the evolution process (one point sexual, two points sexual, asexual).
- `int crossoverAlgo`: this is the variable that holds the value of the crossover algorithm that should be followed throughout the evolution process (random point, smart point, specific point).
- `int crossoverScaling`: this variable specifies if the scaling should be used and if yes, then which type (linear, sigma truncation, power law)
- `int selectionType`: this variable specifies the type of the selection that should be followed throughout the evolution process (Rank, roulette wheel, tournament, random).
- `boolean anneal(...)`: this is the annealing function.
- `void evolve`: this is the evolution function.
- `void crossover(...)`: this function performs the Crossover operation with the supplied genome(s) indexes depending on the type of the crossover previously defined.

- `void calculateFitness(...)`: this is the method that initiates the *Serial* calculation of the fitness for all the individuals.
- `void calculateParallelFitness(...)`: this is the method that initiates the *Parallel* calculation of the fitness for all the individuals.
- `Vector fitness(...)`: this function calls the fitness function in the user's program and is expected to return a `Vector` that contains the fitness values of the supplied Genomes.
- `void init(...)`: this is the method where the different JGenome constructors are called. So in case a user wishes to add a new JGenome class, he should add the call for the new class's constructor in this method.

3.1.3 The Algorithms

The Algorithms are the main engine that run the heuristic search, in which all the critical decisions are taken. These decisions are the factor that differentiates an algorithm from the other, and this is why we have included in this study four different Genetic Algorithms in addition to the Simulated Annealing Algorithm.

The Genetic Algorithm

The Genetic Algorithm has many roles to perform, in order to make the evolution process run in an adequate way. Some of the decisions that need to be taken by the Algorithm are passed to the population object in the form of parameters.

In fact, the first step that is taken by the Genetic Algorithm is to create the population object and initialize it. All the parameters that are passed to the population's constructor were initially passed to, or were automatically deduced by the Genetic Algorithm.

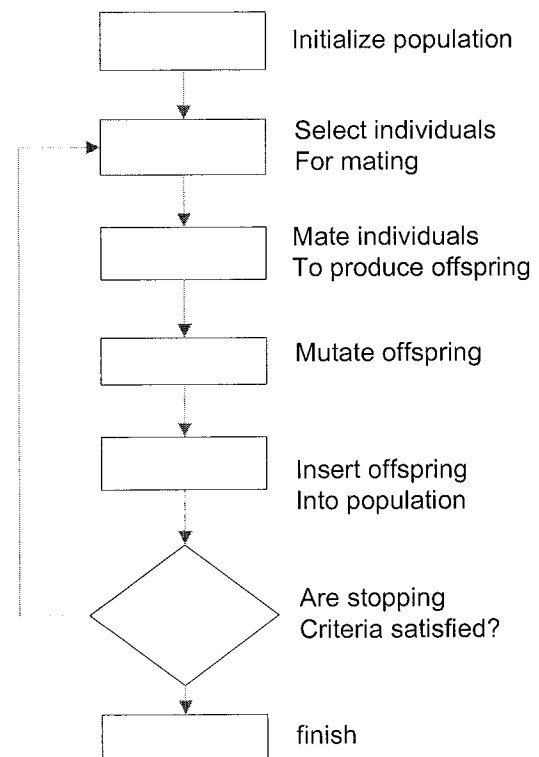


Figure 7 The Genetic Algorithm Flowchart

The next step is to create a loop that will be exited if the stopping criteria of the Genetic Algorithm were satisfied. These stopping criteria depend on the whole evolution process, and are divided into three parts:

- **Number of Generations:** This part depends on the total number of generations that was reached. This total number is passed to the algorithm upon creation, and after every evolution process, it is compared to the evolution counter that exists in the Algorithm itself. If that counter reached the total number, the Algorithm will stop the loop and will report the best solution found so far.
- **Convergence of population:** This part depends on the individuals that exist in the population. Upon its creation, the Genetic Algorithm holds a value of the fitness of the best individual that should be reached. After every evolution, the Genetic Algorithm will call the fitness function in the population, which evaluates the genomes' fitness value. What the algorithm does after every generation, is to compare this stored fitness to the fitness value of the best individual. If these two values were equal, then the Algorithm will stop the execution and report the solution found.
- **Convergence or number of populations:** The third part is a combination of the previous two parts. This method will guarantee that the program will exit anyway, instead of it being stuck in an infinite loop (stuck at a local minimum). The way this method works is by checking first if the fitness of the best individual has reached the maximum fitness value that the Algorithm holds. If it wasn't reached, the Algorithm will check for the number of populations that was

created so far, if it reached its limit, then the Algorithm will stop, otherwise it would continue its evolution process.

After deciding on whether to stop or not, the Algorithm will call the evolve function that exists in the population object. The evolve function, as discussed before, performs the evolution process and creates one new generation only. This new generation will be stored in a separate place, waiting to be inserted in the new population. The next step is to delete some of the genomes from the current generation and fill in their place the set of newly generated Genomes. The decision on which individuals to delete from the current generation depends on the Algorithm type. However, the population class supports all the deletion methods that are used by the Algorithms.

These steps that were mentioned before are drawn in the flowchart in figure 7.

The four types of Genetic Algorithms are discussed in the next sections including the main Genetic Algorithm interface that holds the common functionalities for all the Algorithms.

3.1.3.1 JGeneticAlgorithm

The JGeneticAlgorithm interface contains the essential components for creating a Genetic Algorithm. Every Genetic algorithm that needs to be created should implement this interface to inherit the definition of the main components of the Algorithm.

Every Genetic algorithm in this study holds only two methods: a constructor, and a method to run the algorithm. The first role the Algorithm has to perform, is to

collect the parameters from the user, and this is done through the constructor. After setting all the variables' values, the algorithm will be tailored to perform in the way that the user wishes, and this is when the "runAlgorithm" method should be called.

The runAlgorithm method does not contain a lot of detailed low level programming, instead it only contains the main functions of the algorithm, and the rest was left for the population object to accomplish.

3.1.3.2 JSimpleGeneticAlgorithm

The JSimpleGeneticAlgorithm class uses non-overlapping populations with the optional Elitism that will enable the best genome from the previous generation to be transferred to the new population. In every evolution step, a new population is created having the same size of the current population. Then the Algorithm will replace the current population with the new one, while taking into consideration the Elitism factor.

The most important defined variable that characterizes this Algorithm is the following:

`boolean elitism`: this variable is received by the Algorithm through the constructor defining if the elitism option should be used in the evolution process.

3.1.3.3 JDemeGeneticAlgorithm

The JDemeGeneticAlgorithm class is the class that supports parallelism through the use of the Message Passing Interface.

This algorithm is very easy to use since the needed MPI methods are all embedded in this framework, so the user doesn't need to worry about installing any external library or program.

This algorithm will start by creating and initializing an MPI object which takes care of the server and client initialization, and acquires a rank ID. The client that has an ID = 1 will be responsible to create a population and clone it several times depending on the population size that was supplied by the user. Then the algorithm will wait until the number of clients that are supposed to log in is complete. After the last client logs in to the server, this algorithm will send the population object to all the clients and will send another flag message that starts the evolution process on all the clients.

Each independent population will start by running a Steady-State Algorithm for a repeated number of times. The next step involves using MPI to perform a migration of some individuals to a neighboring population. Every population will send the migrating genomes to the population that has the next rank in the list. As for the population with the highest rank, it will send its migrating genomes to the initial one.

The type of messages that are sent and received by the clients are of non-blocking buffered type, since some populations might finish their evolution and wouldn't be able to receive any other messages from the other clients.

When an Algorithm finishes its evolution process, it sends the best individual of the current population to its neighbor who in turn, will send it to his neighbor, in case it was an optimal solution.

The most important defined variables that characterize this Algorithm are the following:

- `String serverName`: this variable contains the name of the server to which it should connect.
- `int port`: this variable defines the port to which the clients should connect.

- `int migratorsNumber`: this variable defines the number of Genomes that should migrate from a population to the next one.
- `int generationsBeforeMigration`: this variable defines the number of evolutions that should be performed before a migration takes place.
- `int neighborToReceiveFrom`: this variable defines the neighbor that the current Algorithm should receive the migrating genomes from.
- `int neighborToSendTo`: this variable defines the neighbor that the current Algorithm should send the migrating genomes to.
- `int populationsNumber`: this variable defines the number of populations that should be running in parallel.

3.1.3.4 JIncrementalGeneticAlgorithm

The `JIncrementalGeneticAlgorithm` class uses overlapping populations with the new generation's size being equal to two.

The Incremental genetic algorithm allows custom replacement methods to define how the new generation should be integrated into the population. So, for example, a newly generated child could replace its parent, replace a random individual in the population, or replace an individual that has the closest lowest fitness value.

3.1.3.5 JSteadyStateGeneticAlgorithm

The `JSteadyStateGeneticAlgorithm` is very similar to the incremental genetic algorithm with the only difference that the user specifies the percentage of the

population that should be replaced. This Algorithm also supports custom replacement methods and custom population initialization.

The most important defined variable that characterizes this Algorithm is the following:

- `int popReplacementPercentage`: this variable contains the percentage of the population that should be replaced at each iteration.

3.2 The Simulated Annealing Algorithm

The simulated annealing algorithm doesn't involve many complicated internal methods, which makes it an easy algorithm to implement. The total steps that are performed by this algorithm are basic and don't require much computation and analysis. The first step that is taken by this algorithm is to define the initial solution which could be passed by the user. If the initial solution did not exist, then the algorithm will generate a random solution and will use it as a starting point. The next step is to initiate the randomization step according to the current temperature, in order to find a neighbor to the current solution.

This step is embedded in the population object, and it is referred to as the annealing function.

The Annealing Function

The annealing method, that is contained in the population class, is basically a simple one. The basic functionality of this method is to randomize the current solution and to perform the mutation. At this point the Algorithm will check the probability of mutation by calling a function that performs some random calculation and returns if a mutation should take place or not.

The annealing function is usually followed by the transition procedure, which selects the best of the new and current solutions, depending on the current temperature value.

Transition Factor

A transition factor is the basis of the decision function that is used by the Simulated Annealing Algorithm to decide whether the current individual should be replaced by the newly generated one or not.

Consequently, the Annealing method checks the current temperature and calculates the transition factor which is done using the following formula:

$$\mathbf{transitionFactor} = e^{((f_{Current}-f_{New})/temp)}$$

where $f_{Current}$ is the fitness value of the current individual, f_{New} is the fitness of the new individual, and $temp$ is the current temperature. The next step is to generate a random number ranging between 0 and 1 and check if this number is smaller or equal to the transition factor. If it was then the transition to the new individual should take place.

Note that when the temperature is high enough, the new individual will most likely replace the current individual.

The Algorithm next checks if the number of trials for the current temperature has reached its maximum, and if it has, then the algorithm will decrease the value of the current temperature.

In fact the user can choose whether the reduction value is a constant the he will define, or a calculated value that is automatically generated.

The next flowchart explains in details the different steps that are taken by the Annealing Algorithm.

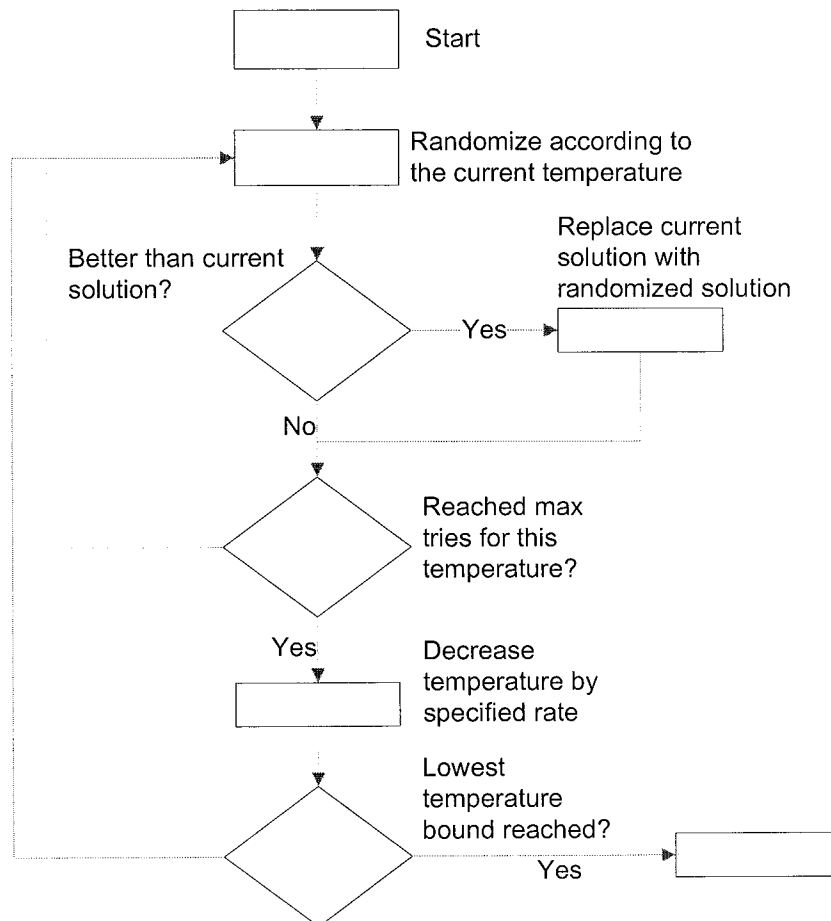


Figure 8 The Simulated Annealing Algorithm Flowchart

Upon the creation of the Simulated Annealing Algorithm, the user decides on the temperature reduction method that will be used throughout the annealing process. There are two methods for reducing the temperature in this project. The first one is to reduce the temperature by a constant. The second one is to use an equation that will handle reducing the temperature depending on the number of cycles requested by the user.

The ratio that multiplies the current temperature in order to decrease it is the following:

$$\text{Ratio} = e^{(\ln(\text{lowest temperature}/\text{highest temperature})/(\text{number of cycles} - 1))}$$

Next the Algorithm checks if the lowest temperature that was defined by the user was reached, and if it has, then the system will stop the iteration process, and will exit and display the solution reached that has the best fitness value. In the next section, the Simulated Annealing class will be mentioned in details including the most important variables and methods.

3.2.1.1 JAnnealingAlgorithm

The JAnnealingAlgorithm class contains the essential components for creating a Simulated Annealing Algorithm. This Algorithm creates an initial solution by creating a JPopulation object with a size equal to one. The JPopulation constructor can be called, with some disabled parameters.

The function that contains the algorithm is called runAlgorithm and it performs the jobs that were shown in figure 8.

The most important defined variables that characterize this Algorithm are the following:

float initialTemperature: this variable contains the value of the initial temperature.

float currTemperature: this variable holds the value of the current temperature.

float lowestTemperature: this variable holds the value of the lowest temperature that should be attained by the system.

int cyclesPerTemperature: this variable holds the number of cycles that should be performed at every temperature value.

3.3 The Message Passing Interface

The Message Passing Interface or MPI, is the followed standard in this system which deals with the process of exchanging messages between the different Algorithms that are running in parallel. MPI's benefit is only used in the Deme Genetic Algorithm which, as described before, initializes and runs many populations on more than one machine. In fact, MPI uses the SPMD (Single-Program, Multiple-Data) model of computation [1], which states that the user has to write a single program and run it on multiple processors, where every process will know what part of the code it should execute.

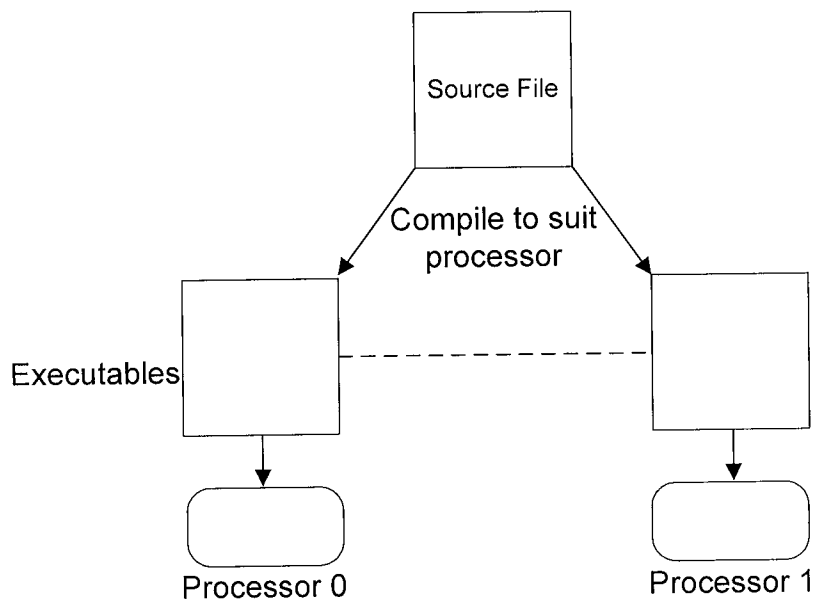


Figure 9 The Single-Program, Multiple Data model [1]

In this framework system, MPI relied heavily on the native JAVA RMI which had the major role in creating a server and connecting it with the different clients. The MPI

classes are all grouped in one package called MPI, and most of these classes serve the requirements of RMI (client, server creation and initialization...).

This MPI implementation was created based on a client server architecture, since the SPMD model does not prevent the use of this architecture [1].

The design of the MPI standard did not have too many requirements, but one of them was the presence of a communicator which holds a list of all the clients that are connected to the server. The definition of the communicator is the following: “A communicator is a communication domain that defines a set of processes that are allowed to communicate with one another” [1].

3.3.1 The MPI Server

To create a server object under RMI, a server interface has to be created, which would hold a definition of all the methods that will be called by the remote client, and all these method definitions should throw a Remote Exception. As for the interface itself, it has to extend the Remote object which will enable this server object to be called through the network. The next step would be to create a server class and make it implement the server interface, which means that it should translate the method definitions into concrete methods. After creating the server object, a specified registry port has to be reserved, to which the server object will be associated, along with the server name.

The MPI server has a rank counter associated to it, which generates a new rank ID and assigns it to a connecting client. The rank value is very important in the life of a client process, especially when using the SPMD model, because the rank is the only way that every process can distinguish the part of the code that it should execute.

When a client connects to the server, it has to initiate the method login which will register this client in the server's processes list. This list is an essential part of the communicator.

After creating the server, the compiler has to be supplied with the variable that will initiate the creation of a server Skeleton and a server Stub, which will make the communication between the clients and the server possible.

3.3.2 The MPI Client

To create a client object under RMI, a client interface has to be created, which will define the methods that will be called remotely, and these methods should throw a Remote Exception. When an MPI client connects to the server, it uses the RMI method Naming.lookup which will search for the server object in the registry, and when found, will return a remote link that will be saved in a remote server interface on the client side. The next step for the client is to call the method connect which will add the client to the communicator and assign his rank on the server.

Upon creation, the client is assigned a rank ID number that is supplied from the server. Then the client will search in the supplied code for the part that he is supposed to run according to his rank. This code division should be handled by the programmer and the client shouldn't be worrying about code differentiation.

The core of MPI is embedded in the client, since all the message passing methods are defined in it. These methods begin with the letters MPI_ to make sure that they are not mixed up with other methods in the code.

The server makes sure that at all time, all the registered clients have a list of the other clients, so that the communication will be possible between them. So whenever a new client gets connected, the server will broadcast a message to all the clients forcing them to update their client list.

The most basic methods of communication are: `MPI_Send` and `MPI_Receive`.

`MPI_Send` is initiated from a client. This method will be directed to the destination client and will block the code on the source client until the destination client reaches the part of his code that states `MPI_Receive` from the source client. The other name for these two methods is blocking send and blocking receive. There are many types of methods that send and receive data which can be non-blocking, or even buffered, which brings us to the next subject: the MPI Buffer.

3.3.3 The MPI Buffer

Upon creation, every client initializes a buffer: the MPI Buffer. `MPI_Buffer` is a simple class whose functionality is to hold all the messages that reach the client and store them in a buffer. Thus when the client reaches the part that initiates a receive, this client will check the buffer for any messages that are queued. MPI Buffer is a very efficient way of sending and receiving data especially when being used in applications where not all the processes are homogeneous. For example if the parallel program was running on a network of heterogeneous machine, where each machine is running a process, some machines will finish their tasks before the others. This would result in a wasted processing time, if the source client is waiting for the destination client to reach the part that initiates the message reception.

3.3.4 MPI

To create an MPI based application, the user has to create an MPI object. This object will automatically deal with the server and clients objects, and all what the user has to do is to distribute his code on the machines and specify the name of the server and the port to which the clients should connect.

The first task that is done by the MPI object is to check the name of the local machine and compare it to the server name. If that name was the same, then it will try to create an MPI Server. A successful server creation will imply that this machine is supposed to run the server and receive client requests from other machines. In case of a failure or a success, the MPI object will move to creating a client and connecting it to the server that presumably should already be existing.

In case the user wishes to force all the clients to start running the code at the same time, he should use blocking send and receive methods to block the clients from executing any code. And when the server checks that all the clients are connected, it will send a message to the list of clients that he has, signaling them to start the execution.

The next part will skim through the different classes that form the MPI package.

3.3.1.1 MPI_Server

As discussed previously, the `MPI_Server` class includes all the initialization methods that will enable the MPI standard to work properly. These methods include server creation, client login, message broadcasting and many more...

The most important defined methods and variables that characterize this Class are the following:

`Hashtable broadcastList`: this variable holds a link to all the clients that are currently connected to the server.

`int rankCounter`: this is the counter that assigns the ranks to the connecting clients.

`synchronized int connect(...)`: this method is created in a way (synchronized) that only one client can run it (or one client can connect to the server) at a certain time.

`synchronized boolean login(...)`: this method is also a synchronized method and it will ensure that all the clients have a distinct rank value.

`void broadcastMessage(...)`: this method sends a message to all the clients. It is mainly used when a new client connects to the server, to update the clients' lists.

3.3.1.2 MPI_Client

`MPI_Client` is the class that contains all the sending and receiving methods which are the building blocks of the communication process. In addition to the client initialization and connection methods, there is a `finalize` method that logs a client off the server and stops execution of its code. Non blocking buffered send and receive methods are also implemented in this class, which are very useful for our parallel algorithm.

The most important defined methods and variables that characterize this class are the following:

`Hashtable processList`: this variable keeps a record of all the clients that are currently connected to the server.

`MPIBuffer buffer`: this is the buffer that is used to store messages.

`MPIServerInterface MPI_Server`: this is the variable that holds a link to the server.

`MPI_Send()`: this method sends a blocking message to a destination client.

`MPI_Receive()`: this method blocks until it receives a message from a source client.

`int MPI_IbSend(...)`: this method sends a non blocking buffered message to a destination client

`MPI_IbReceive()`: this method doesn't block and it receives a message from a source client through a buffer.

3.3.1.3 MPI

The MPI class is what the user has to create in case he wishes to use the MPI library. As discussed before, the MPI object will create a server object, followed by client object. After creating the MPI object, the user has to call the `MPI_Init(...)` method which performs all these initialization methods.

The most important defined methods and variables that characterize this Class are the following:

`int myRank`: this is the variable that holds the rank of the client object.

`MPIClient mpiClient`: this is the variable that holds the MPI client object.

`MPIServer mpiServer`: this is the variable that holds a link to the MPI Server.

`void MPI_Init (...)` : this is the method that initializes the server creation and then creates the client object.

`void createServer (...)` : this is the method that creates the server object.

Chapter 4:

Case Study

4.1 The graphical user interface: The Wizard

The Graphical User Interface or GUI, was added to this framework to facilitate its usage. The wizard's main job is to collect the data from the user in a smooth way, and then to display the constructor that should be used to create the proper Heuristic Algorithm.

The wizard is a succession of screens that are displayed to the user asking him about the options that he wishes to include or exclude in his Algorithm.

The wizard is of a dynamic type, meaning that the next screen is always related to the previous one, since some algorithms are not related to the others, thus the impossibility of using a common wizard for all the algorithms.

The wizard's first screen is a welcome screen that explains its features. It also states the necessity of including this framework in the user's project to take full advantage of all the built in functionalities.

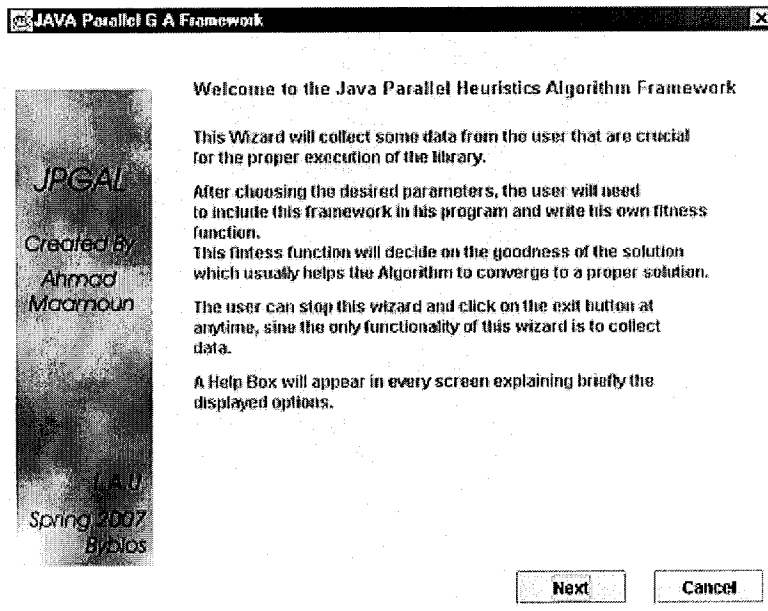


Figure 10 The wizard's first screen.

The second screen gives a choice for the user to select the algorithm that he wishes to use. The list includes the supported algorithm: Simple GA, Incremental GA, Steady State GA, Deme GA, and Simulated Annealing.

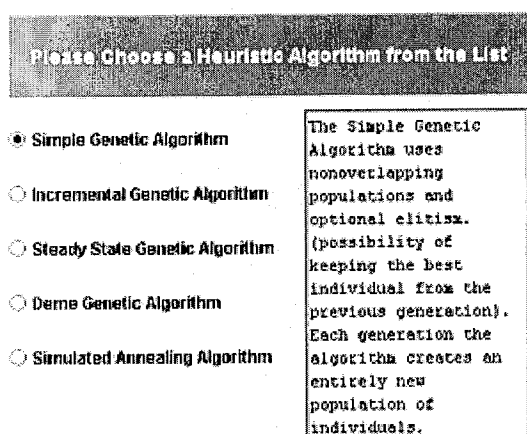


Figure 11 The wizard's second screen

A help box is placed in every screen that is shown to the user to help him decide on the options that he can choose. Upon the selection of any algorithm, the help box will change its contents, and display some details about the user's choice.

The third screen is related to the properties of the Solution that the user wishes to use in his algorithm. The user has to decide on the Solution's type: (BYTE, CHARACTER, DOUBLE), its size, the lower and upper bounds of the type that he chose, and the maximum expected fitness. In this and all the following screens, the help box will be situated in the bottom of the panel.

Properties of the Solution

Select the type of your solution: CHARACTER

Enter the precision value: []

Enter the size of your solution: []

Enter the upper bound value: []

Enter the lower bound value: []

Enter the Expected Fitness of the Solution: []

The type of the solution depends on the problem in hand. For example if the problem is a VLSI circuit which solution is an input vector consisting of 'zeroes' and 'ones' then the choice should be BYTE. More details can be embedded by []

Figure 12 The type of the solution

The Simulated Annealing Algorithm

In case the user chose the Simulated Annealing algorithm, the next screen will ask the user to input the Simulated Annealing related variables. These variables are the following: the initial, the minimal and maximal temperatures, and the temperature reduction type which is either by a constant or a calculation.

Please Choose the properties of the Algorithm

Enter the initial temperature

Enter the lowest temperature

Enter the number of cycles per temperature

Select the temperature reduction type

Enter the temperature reduction constant

The initial temperature is the starting value of the temperature which will be decreased gradually with the evolution process.

Figure 13 S.A. general properties

The last screen will show the constructor of the JAnnealingAlgorithm and some instructions to the user on how to include the initial solution, in case he had one.

The Simulated Annealing Constructor

This is the constructor that should be used

```
JAnnealingAlgorithm myAlgo = new
JAnnealingAlgorithm(12,0,12,JPopulation.GENOME_TYPE_C
HARACTER,0,1,null,100,0,12,JPopulation.FILL_USING_RAN
DOM,CALCULATION,,5,myProgram)
```

Figure 14 The Simulated Annealing last screen

The Genetic Algorithm

In case the user chose any other Genetic Algorithm, then the wizard will ask the user some genetic Algorithm related questions spread on the screens.

The next screen will collect information on the mutation percentage, the size of the population, and on the selection types which vary between: RANDOM, ROULETTE WHEEL, TOURNAMENT, and RANK

As for the evolution operators, the wizard will collect the following data:

- Fitness scaling. The choices are: NO SCALING , LINEAR SCALING, SIGMA TRUNCATION SCALING, POWER LAW SCALING
- Crossover Algorithm. The choices are: RANDOM POINT, SPECIFIC POINT, SMART POINT.
- Crossover type: The choices are: ASEXUAL, ONE POINT SEXUAL, TWO POINT SEXUAL
- Replacement percentage, will only be displayed if the user's choice was Deme GA or Steady State GA only.
- Replacement Strategy, which will only be displayed in the Incremental GA. The choices are: REPLACE THE WORST, REPLACE PARENT, REPLACE RANDOM, REPLACE THE CLOSEST
- Elitism, which will only be displayed in the simple GA.

Collecting information about the Algorithm

Enter the Mutation percentage

Enter the population size

Select the genome selection method

Select the Fitness Scaling method

Select the Crossover Algorithm

Select the Crossover Type

The population size is nothing but the total number of solutions that will be created to participate in the searching. The population size will remain constant throughout the evolution process.

Figure 15 Genetic Algorithm properties

Figure 15 shows the information that is related to the population, which is common to all the Genetic Algorithms.

The next screen asks the user to input the termination method that should be used for the algorithm, in addition to the name of the class that contains the fitness function. The other information depends on the Genetic Algorithm that was previously selected.

If the case was Simple GA, then the screen will collect information about elitism.

Collecting information about the Algorithm

Select the algorithm termination method: GENERATION NL

Enter the maximum number of generations: []

Enter the name of the class that contains the fitness function: myProgram

Select if Elitism should be applied: FALSE

This choice sets how the Algorithm should stop the evolution process.

Figure 16 The Simple Genetic Algorithm

If the case was Incremental GA, then the screen will collect information about the replacement strategy that should be followed throughout the evolution process.

Collecting information about the Algorithm

Select the algorithm termination method: GENERATION NL

Enter the maximum number of generations: []

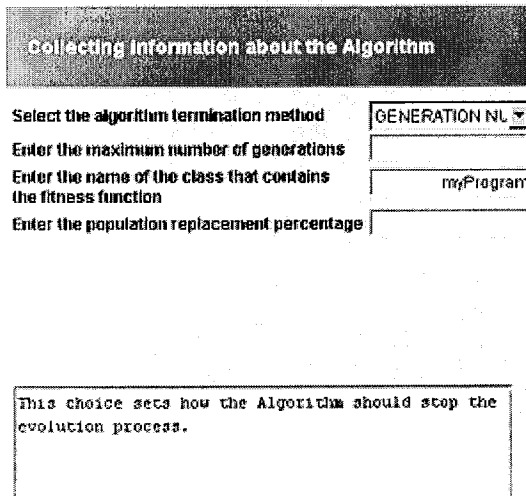
Enter the name of the class that contains the fitness function: myProgram

Select the replacement strategy: RANDOM

This choice sets how the Algorithm should stop the evolution process.

Figure 17 The Incremental GA

If the case was Steady-State GA, then the screen will collect information about the population replacement percentage that should be followed throughout the evolution process.



Collecting Information about the Algorithm

Select the algorithm termination method: GENERATION NL

Enter the maximum number of generations:

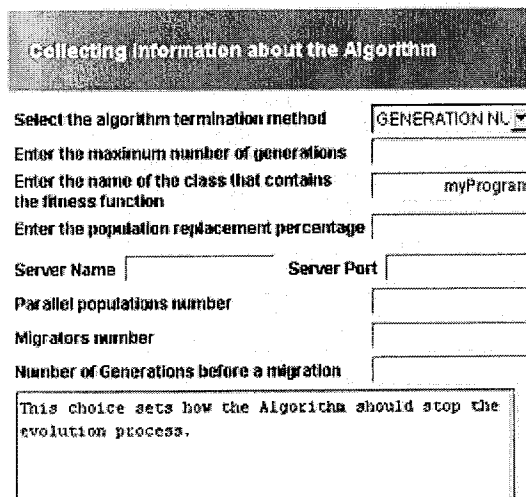
Enter the name of the class that contains the fitness function: myProgram

Enter the population replacement percentage:

This choice sets how the Algorithm should stop the evolution process.

Figure 18 The Steady-State Algorithm

If the case was Deme GA, then the screen will collect information about the population replacement percentage, the server name, the server port, the number of the parallel populations, the migrators number and the number of generations that should be performed before a migration occurs.



Collecting Information about the Algorithm

Select the algorithm termination method: GENERATION NL

Enter the maximum number of generations:

Enter the name of the class that contains the fitness function: myProgram

Enter the population replacement percentage:

Server Name: Server Port:

Parallel populations number:

Migrators number:

Number of Generations before a migration:

This choice sets how the Algorithm should stop the evolution process.

Figure 19 The Deme GA

The last screen is common to all the Algorithms, where it will show the constructor of the Genetic Algorithm that the user chose and some instructions to the user on how to include the initial solution, in case he had one.

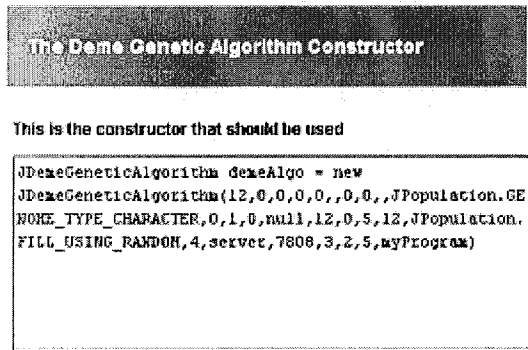


Figure 20 The Wizard's last screen

4.2 Testing the framework: The ULSI fault detector

4.2.1 Testing the Algorithms

The Genetic Algorithm

Having explained how the Genetic Algorithms functioned in the previous chapters, this chapter will begin by introducing the way that this framework will be tested by first writing some definitions which are followed by explaining the fault detection program that was incorporated in this work, especially for testing purposes.

The test will be applied on all the Genetic Algorithms that were developed in this work, with different options set to each one. In order to test all the components that were mentioned earlier including different crossover algorithms, diverse genome selection methods and different scaling techniques will be applied.

All the testing phase will be performed using the ULSI fault detection program which takes as an input an ISCAS Benchmark, locates all the stuck-at faults, and tests the supplied input to check how many faults it can detect. The Algorithm in this case will be the engine that will supply the ULSI circuit tester with the proper input, and will receive, as a returned value, the number of detected faults detected by the testing vectors.

The Simulated Annealing Algorithm

The test that will be applied on the Simulated Annealing Algorithm will focus on changing the parameters that are related to this specific Algorithm. In other words, the test will focus on checking the effect of the initial temperature by varying it between one test and the other. This test will also check the effect of changing the number of cycles per temperature, and changing the temperature reduction type. Parallel fault simulation will be used in this test, thus the algorithm will accept the array of individuals and return their fitness values.

4.2.1 The ULSI Fault Detection Program

“The complexity for minimum component costs has increased at a rate of roughly a factor of two per year ... Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer.” – Moore’s Law [19].

In 1965 Gordon Moore declared that every 24 months, the number of transistors on a chip will double, thus increasing the probability of a fault to occur on an integrated circuit. This is why testing this circuit for production faults is an important issue that should be inspected.

Stuck-at Faults

A “stuck-at” fault is a defect that happens in the stages of production of integrated circuits, where a short circuit can force a connector to be stuck at a certain voltage value, regardless of the voltage that was applied at its boundary. There are two types of “stuck at” faults: stuck at zero, where the connector is short circuited with the ground, and a stuck at one where the connector’s voltage is short circuited to a high voltage source.

Stuck at faults can be detected at the gates that are joined to these connectors by checking the displayed logic value at their output. If the gate is expected to display a logic value that is different from what it is currently displaying, then the fault is detected. For example driving a logic zero to the inputs of an AND gate should result in a zero logic value at its output. If the displayed value was a logic one, this is when we can detect that there is a stuck at fault in this Gate.

Fault Equivalence

Every connector in an integrated circuit can be faulty, but some faults can be detected in more than one location. In other words a fault’s detection can be equivalent to some other fault’s detection, which makes the two faults Equivalent. For example, a stuck at zero fault that exists on an input of an AND gate will be detected in the same way that a stuck at zero fault will be detected at the output of this gate. Applying fault equivalence on a circuit can reduce the number of faults that should be detected, thus reducing the time spent to check the circuit for defects.

4.2.2 The Fault Simulator

The fault simulator accepts as an input, a file that is written following the ISCAS Benchmark format which positions the inputs in the following way: INPUT(G0) then the outputs: OUTPUT(G7), then the gates: G13 = AND(G20, G25).

After accepting this file, it starts creating the different components that form this circuit and links them using various data structures. The next step would be to find all the stuck at faults that exist in the circuit and apply fault equivalence on them to reduce the number of faults that need to be detected.

This fault simulator supports two types of simulations: Serial and parallel fault simulation

Serial Fault Simulation

Serial Fault simulation is the basic method that is used to simulate faults in a circuit. It begins by first receiving an input vector of possible input values. Then the circuit gets simulated, and the logic values of all the gates are stored in a data structure. These values show how the normal fault-free simulation of the circuit should result. Next the faults get simulated each one at a time, by injecting a fault in the circuit and then checking the outputs for any change than the expected value. If any of the outputs shows a difference, and this is when the fault is detected.

Parallel Fault Simulation

Serial Fault Simulation is a very slow procedure which is destined more for humans than for machines. This is when circuit testers started coming up with alternative testing methods that are faster and more efficient.

The basic idea that was behind parallel fault testing was to traverse the circuit once, and for every gate check its value with more than one input possibility [15]. To perform a Parallel Fault Simulation, the common bits of the input vectors are stored in sequence in a variable that can store enough data such as the long datatype. The next step would be to simulate the circuit with these inputs and to use bitwise logical operators to calculate the results.

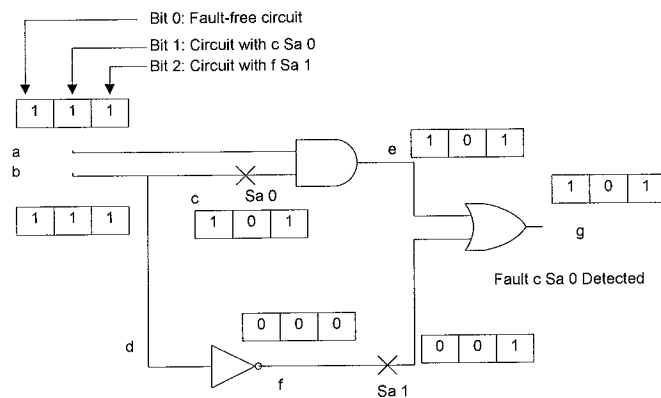


Figure 21 Parallel Fault Simulation [15]

4.2.3 Simulating the Algorithms

In the second part of this chapter, after presenting the VLSI fault simulator circuit, the algorithms will be simulated for error-free code verification on the c432, c1355, c1908, and the c3540 ISCAS Benchmarks.

In every simulation option, the parameters that were submitted to the algorithm will be stated, but there are some default variables that are common to all the algorithms. The first default parameter is the solution type which in this case is chosen to be the JGenomeByteArray type with the upper bound = 1 and the lower bound = 0. As for the genetic algorithms, they will be assigned the Limited Convergence termination method.

4.2.4 Simulating the Simulated Annealing Algorithm

In this simulation test, the following parameters were set:

Table 2 SA parameters

Test #	Circuit Name	Genome Size	Mut. %	Initial Temp	Lowest Temp	Exp. Fitness	Reduction Type	Cycles/Temp
1	C432	36	1	150	0	524	CALCULATION	20
2	C432	36	5	200	0	524	CONSTANT	25
3	C1355	41	1	150	0	1574	CALCULATION	20
4	C1355	41	5	200	0	1574	CONSTANT	25
5	C1908	33	1	150	0	1880	CALCULATION	20
6	C1908	33	5	200	0	1880	CONSTANT	25
7	C3540	50	1	150	0	3431	CALCULATION	20
8	C3540	50	5	200	0	3431	CONSTANT	25

Table 3 SA Test results

test #	Total detected faults
1	502
2	513
3	1501
4	1524
5	1798
6	1829
7	3204
8	3225

From the previous results we can conclude that by increasing the initial temperature, we will get better results, and that is completely logical since the algorithm is generating more individuals and taking more time to find a better solution

4.2.5 Simulating the Genetic Algorithms

4.2.5.1 Simulating the Simple Genetic Algorithm

In this simulation test, the following parameters were set:

Table 4 Simple GA parameters

Test ID	ISCAS Circuit	Gen. Size	Exp. Fitn.	Mut. %	Cross. Scal.	Cross. Type	Cross. Algo.	Pop. size	Select. Type	Term. Algo.	Gen. Num.	Elitism
1	C432	36	524	1	No	Sexual 1 point	Ran. Point	15	Roulette wheel	Limited conv	150	No
2	C432	36	524	3	Linear	Sexual 1 point	Smart point	25	Tourn.	Limited conv	150	Yes
3	C1355	41	1574	1	No	Sexual 1 point	Ran. Point	15	Roulette wheel	Limited conv	150	No
4	C1355	41	1574	3	Sigm.	Sexual 2 point	Smart point	25	Tourn.	Limited conv	200	Yes
5	C1908	33	1880	1	No	Sexual 1 point	Ran. Point	15	Roulette wheel	Limited conv	200	No
6	C1908	33	1880	3	Pow.	Sexual 2 point	Smart point	15	Tourn.	Limited conv	200	Yes
7	C3540	50	3431	1	No	Sexual 1 point	Ran. Point	15	Roulette wheel	Limited conv	150	No
8	C3540	50	3431	3	Linear	Sexual 2 point	Smart point	25	Tourn.	Limited conv	200	Yes

Table 5 Simple GA test Results

test #	Total detected faults
1	500
2	508
3	1497
4	1512
5	1805
6	1819
7	3215
8	3217

From these results we can conclude that the results can be improved by changing some factors. These factors include

- increasing the number of generations
- Using some scaling function (power law, sigma truncation or linear scaling)
- Increasing the size of the population

After enabling the Elitism option, which is the special feature of this algorithm, we also noticed some improvement.

4.2.5.2 Simulating the Incremental Genetic Algorithm

In this simulation test, the following parameters were set:

Table 6 Incremental GA

Test ID	ISCAS Circuit	Gen. Size	Exp. Fitn.	Mut. %	Cross. Scal.	Cross. Type	Cross. Algo.	Pop. size	Select. Type	Term. Algo.	Gen. Num.	Replac.
1	C432	36	524	1	No	Sexual 1 point	Ran. Point	20	Roulette wheel	Limited conv	200	Rand.
2	C432	36	524	3	Linear	Sexual 2 point	Smart point	20	Tourn.	Limited conv	200	Worst
3	C1355	41	1574	1	No	Sexual 1 point	Ran. Point	25	Best.	Limited conv	150	Rand.
4	C1355	41	1574	3	Sigm.	Sexual 2 point	Smart point	25	Roulette wheel	Limited conv	150	Rand.
5	C1908	33	1880	1	No	Sexual 1 point	Ran. Point	15	Roulette wheel	Limited conv	100	Worst
6	C1908	33	1880	3	Pow.	Sexual 2 point	Smart point	20	Tourn.	Limited conv	100	Rand.
7	C3540	50	3431	1	No	Sexual 1 point	Smart point	15	Best	Limited conv	150	Closest
8	C3540	50	3431	3	No	Sexual 1 point	Ran. Point	15	Best.	Limited conv	150	Closest.

Table 7 Incremental GA test results

test #	Total detected faults
1	489
2	510
3	1511
4	1506
5	1804
6	1812
7	3216
8	3189

From these results we can conclude the following:

- The replacement option has affected our results in a significant way since choosing to replace the worst individual has shown a noticeable improvement.
- Using the roulette wheel selection has revealed an improvement since the bad individuals still have a chance to be selected and to introduce new genes to the population.
- The last tests have shown that the use of schemas improved our solution.

4.2.5.3 Simulating the Steady State Genetic Algorithm

In this simulation test, the following parameters were taken:

Table 8 Steady State GA parameters

Test ID	ISCAS Circuit	Gen. Size	Exp. Fitn.	Mut. %	Cross. Scal.	Cross. Type	Cross. Algo.	Pop. size	Select. Type	Term. Algo.	Gen. Num.	Repl. %
1	C432	36	524	1	No	Sexual 1 point	Ran. Point	25	Tourn.	Limited conv	150	10
2	C432	36	524	3	No	Sexual 1 point	Smart point	20	Roulette wheel	Limited conv	150	25
3	C1355	41	1574	1	Sigm.	Sexual 2 point	Ran. Point	15	Roulette wheel	Limited conv	150	10
4	C1355	41	1574	3	Sigm.	Sexual 2 point	Ran. point	15	Roulette wheel	Limited conv	200	25
5	C1908	33	1880	1	Pow.	Sexual 1 point	Smart. Point	15	Best	Limited conv	200	10
6	C1908	33	1880	3	Pow.	Sexual 1 point	Smart point	20	Tourn.	Limited conv	200	25
7	C3540	50	3431	1	No	Sexual 1 point	Ran. Point	15	Roulette wheel	Limited conv	150	10
8	C3540	50	3431	3	Linear	Sexual 2 point	Ran point	15	Tourn.	Limited conv	200	10

Table 9 Steady State GA test results

test #	Total detected faults
1	503
2	505
3	1515
4	1525
5	1812
6	1820
7	3223
8	3201

From these results we can conclude the following:

- Increasing the replacement percentage is beneficial most of the time.
- The use of roulette wheel selection is showing better results than tournament selection.

4.2.5.4 Simulating the Deme Genetic Algorithm

In this simulation test, some general common parameters were specified for all the circuits. These parameters are:

Parallel populations number = 4

Migrants number = 4

Number of generations before a migration = 5

Table 10 Deme GA parameters

Test ID	ISCAS Circuit	Gen. Size	Exp. Fitn.	Mut. %	Cross. Scal.	Cross. Type	Cross. Algo.	Pop. size	Select. Type	Term. Algo.	Gen. Num.	Repl. %
1	C432	36	524	1	Linear	Sexual 2 point	Ran. Point	15	Roulette wheel	Limited conv	100	10
2	C432	36	524	3	Linear	Sexual 2 point	Smart point	20	Tourn.	Limited conv	250	20
3	C1355	41	1574	1	No	Sexual 1 point	Ran. Point	20	Roulette wheel	Limited conv	250	30
4	C1355	41	1574	3	Sigm.	Sexual 2 point	Smart point	25	Tourn.	Limited conv	250	15
5	C1908	33	1880	1	Sigm	Sexual 1 point	Ran. Point	20	Roulette wheel	Limited conv	150	20
6	C1908	33	1880	3	Pow.	Sexual 2 point	Smart point	20	Tourn.	Limited conv	150	15
7	C3540	50	3431	1	No	Sexual 1 point	Ran. Point	15	Roulette wheel	Limited conv	150	10
8	C3540	50	3431	3	Linear	Sexual 2 point	Smart point	25	Tourn.	Limited conv	200	25

Table 11 Deme GA Test Results

test #	Total detected faults
1	Best: 511
2	Best: 519
3	Best: 1521
4	Best: 1526
5	Best: 1811
6	Best: 1828
7	Best: 3230
8	Best: 3228

From these results we can conclude the following:

- The migration feature has added good genes in every simulation.
- The migration is making up for deficiencies that exist amongst the parallel populations since they are all sharing their findings with each others.
- The final detected faults by the different parallel populations have close numbers, although at the beginning some populations were improving faster than the others. But when a migrant was being introduced to the population, it was boosting the number of detected faults.

Chapter 5: Conclusion

The work that was done in this thesis spanned over an important period of time, where the MPI and the VLSI packages were first developed and tested independently, and then joined along with the GA components and algorithms to form this completely “independent” framework. The word “independent” was chosen simply because all the previous works did not include a built-in library that would manage a standard such as the Message Passing Interface. Instead, the couple of works that were conducted with the support of MPI, needed a previous installation of this package on the workstation. External packages such as MPICH and MPICH 2 were proposed in those cases, for the machines running the windows O.S..

The code encapsulation feature of this framework was demonstrated in the VLSI circuit tests. In fact, the VLSI fault detector was developed independently to test the integration with the framework which was a success. This makes our framework a reliable resource for the students of the introductory courses to use .

After comparing the results that were performed on the same ISCAS files, using all the algorithms, an obvious speed up was noticed through the use of parallelism. In fact, since every copy of the population will be directed in a different direction, the search will make sure to broaden the search area. Another benefit is that whenever a version of the algorithm gets stuck at a local peak, the migrating individuals will add new data and new possibilities for the algorithm to use in his next evolution. The testing

results were satisfactory but the problem was with the time that the VLSI circuit was taking to simulate the circuit. Some extra work needs to be done on this part to optimize the VLSI fault simulator and reduce the calculations to a minimum value in order to make the simulation process run faster.

This project can be extended with many extra features, especially in the VLSI part, where other types of fault simulation methods can be added like the concurrent fault simulation or satisfiability simulation. More solution types can also be added to globalize even further the use of the built in algorithms. As for the genetic algorithms, they can be extended with other heuristic searching algorithms such as the tabu search, deterministic crowding, differential evolution... which can be also parallelized through the use of the MPI package that made the parallelization job an easy one.

As for the parallel Algorithm, dynamic user login can be easily developed and added so that whenever a user logs in, he will get a copy of the database and start the evolution process.

Appendix

The ISCAS Benchmarks that were used in this study are listed in the following table:

Circuit Name	No. of gates	No. of Inputs	No. of Outputs	Total Faults	Collapsed Faults
C03	18	7	4	104	69
C432	160	36	7	864	524
C1355	546	41	32	2710	1606
C1908	880	33	25	3816	2041
C2670	1193	233	140	5340	2943
C3540	1669	50	22	7080	3651
C5315	2307	178	123	10630	5663

References

- [1] Wilkinson, B., & Allen, M. (2005). *Parallel programming, techniques and applications using networked workstations and parallel computers* (2nd ed.). [n.p.]:Prentice Hall.
- [2] Wall, M. (1996), *GALIB: A C++ library of genetic algorithm components*. [n.p.]: Massachusetts Institute of Technology.
- [3] Adcock, S. (2005). *GAUL: Genetic algorithm utility library*. Retrieved June 25, 2007 from: <http://gaul.sourceforge.net/>
- [4] Levine, D. (1996). *PGAPack: Parallel genetic algorithm library*. Retrieved June 25, 2007 from: http://www-fp.mcs.anl.gov/CCST/research/reports_pre1998/comp_bio/stalk/pgapack.html
- [5] Rybarski, J. (2006). *JGAL: Java genetic algorithms library*. Retrieved June 25, 2007 from: <http://jgal.sourceforge.net/>
- [6] Meffert, K. (2002). *JGAP: Java genetic algorithm package*. Retrieved June 25, 2007 from: <http://jgap.sourceforge.net/>
- [7] Dolan, A. (1998). *GA playground*. Retrieved June 25, 2007 from: <http://www.aridolan.com/ga/gaa/gaa.html>
- [8] Smith, J. (2000). *GA: Genetic algorithms*. Retrieved June 25, 2007 from: http://www.softtechdesign.com/products/GA_Delphi/GeneticAlgorithm.htm
- [9] Koutnik, J. (2000). *JAGA: Java genetic algorithm package*. Retrieved June 25, 2007 from: <http://cs.felk.cvut.cz/~koutnij/studium/jaga/jaga.html>
- [10] Goodman, E. (1997). *GALOPPS: Genetic algorithm optimized for portability and parallelism system*. Retrieved June 25, 2007 from: <http://garage.cse.msu.edu/software/galopps/index.html>
- [11] Naughton P., & Schildt, H. (1999). *Java2: The complete reference*, (3rd ed.). [n.p.]:Osborne/McGraw-Hill.
- [12] Ivask, E. (1998). *Genetic algorithms in test pattern generation*. Unpublished master's thesis. Tallinn Technical University, Estonia.
- [13] Kernighan, B., & Ritchie, D. (1988). *The C programming language*, (2nd ed.). [n.p.]:Prentice Hall.
- [14] Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2001). *Introduction to algorithms*, (2nd ed.). [n.p.]: The MIT Press.

- [15] Bushnell, M., & Agrawal, V. (2000). *Essentials of electronic testing for digital, memory & mixed-signal VLSI circuits*. [n.p.]: Springer.
- [16] Goldberg, D. *Genetic algorithms in search, optimization, and machine learning*. [n.p.]: Addison-Wesley Publishing Company.
- [17] Holland, J.H. (1975). *Adaptation in natural and artificial systems*. [n.p.]: University of Michigan Press
- [18] Otten, R.H.J.M., & Van Ginneken, L.P.P.P. (1989). *The annealing algorithm*. [n.p.]: Kluwer Academic Publishers
- [19] Wikimedia Foundation. *Wikipedia, The free encyclopedia*. Retrieved June 25, 2007 from: <http://www.wikipedia.com>
- [20] Kirkpatrick, S. , Gelatt, C. D. & Vecchi, M. P. (1983). *Optimization by simulated annealing*. Science, Vol 220 number 4598
- [21] Kliwer G. & Tschoke, S. (2000). *Parallel simulated annealing library*. Retrieved June 25, 2007 from: <http://wwwcs.uni-paderborn.de/~parsa/>