Czech Technical University in Prague
Faculty of Electrical Engineering

Master's Thesis

Implementation of the Parallel Selfish Gene genetic algorithm

Bc. Floris van der Meijs

Supervisor: Ing. Petr Fišer, Ph.D.

Study program: Electrical Engineering and Information Technology

Study field: Computer Science and Engineering

May 2008

Declaration

I hereby declare that I have completed this master's thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákona č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

Prague, 23 May 2008 .....

Abstract

On the crossroads between the fields of evolutionary computation, parallel systems and VLSI design, the Parallel Selfish Gene Algorithm (PSG) is situated. PSG is a new parallel evolutionary algorithm based on recent developments in modern evolutionary biology.

This thesis studies existing evolutionary and parallel evolutionary algorithms, outlining the state of the art in the field, and describes the design and implementation of PSG. The algorithm is subjected to a comprehensive set of experiments evaluating its performance in solving the difficult problem of evolving high quality test pattern generators for the built-in self-test of digital circuits.

The outcome is favorable; PSG is a processor-efficient algorithm. The extension onto multiple processors gives users the ability to reduce execution time or increase solution quality. Furthermore, the test pattern generators evolved by the algorithm are of a high quality, achieving fault coverage rates not normally obtained by heuristic methods.

Contents

List of Figures

List of Tables

1 Introduction

One of the most intriguing things about the field of computer science and engineering is the fact that its disciplines are intricately related and meet in various unexpected places. This work explores one such place, where evolutionary algorithms, parallel systems and VLSI design come together to form the Parallel Selfish Gene Algorithm, a parallel evolutionary algorithm based on the latest developments in modern evolutionary biology. This algorithm is used to solve the difficult problem of creating efficient test pattern generators for built-in self-test circuitry.

1.1 Aim of the work

The work stands at a crossroads between three major fields of science; unsurprisingly, the aim of the work is threefold. The first goal is to study and describe the state of the art in evolutionary algorithms and parallel evolutionary algorithms, in particular the Selfish Gene Algorithm, a lesser-known evolutionary algorithm that breaks away from its field by adopting a new and somewhat controversial interpretion of classical Darwinian evolution.

The second goal is to design and implement a parallel version of the Selfish Gene Algorithm, taking into account the special nature of the Selfish Gene Algorithm, as well as the constraints imposed by contemporary parallel systems. Of course, this implies the additional goal of thoroughly analyzing the algorithm's performance, scalability and efficiency.

The final aim is to utilize this algorithm to solve the problem of evolving cellular automata for use in the built-in self-test of digital circuits, again following the road of the lesser-known but potentially superior approach.

1.2 Background

The field of evolutionary computation is wide and diverse and has spawned, throughout its history, a host of research and algorithms. These algorithms, besides all being heuristic in nature, share one common characteristic: they attempt to exploit the principles of natural selection and population genetics to find quick and accurate solutions to difficult problems. One particular algorithm was created in 1998 and attempts to introduce some of the recent developments in modern evolutionary biology to the field of evolutionary computation. This algorithm is known as the Selfish Gene Algorithm.

The field of parallel systems and algorithms is a venerable discipline in the realm of computer science, with many academic as well as practical angles. Despite its age and status, the field is ever-changing, keeping abreast of modern developments in both hardware and software.

The synergy of these two fields results in parallel evolutionary algorithms. Unlike other parallel algorithms, parallel evolutionary algorithms are more than just evolutionary algorithms adapted to execute on multiple processors. In fact, their biological foundations may cause a hidden dimension to appear, by which parallel evolutionary algorithms exhibit unique behavior not found among other (parallel) algorithms.

In the field of VLSI design, the principle of built-in self-test (BIST) is based on the need to provide mechanisms by which a digital circuit may verify its functionality without the need for external testing. The success of BIST relies on a unit known as the test pattern generator, and the problem of creating and setting up good test pattern generators is a difficult one for which clever search algorithms are needed.

1.3 Structure of the thesis

The text is organized into eight chapters. After the brief introduction to the research areas given in this chapter, the next chapter delves into the field of evolutionary computation, introducing the overall concepts and also the Selfish Gene Algorithm. For the interested reader an overview of the algorithm's biological foundations is presented as well.

The third chapter addresses the field of parallel evolutionary algorithms in a similar way. This chapter also provides an overview of parallel systems and defines some of the metrics that are in use for analyzing parallel algorithms.

The Parallel Selfish Gene Algorithm (PSG) is introduced in chapter 4. A detailed overview of the algorithm's structure and architecture is given in the first part, while the second part introduces the concept of self-adaption and shows how it may be used in an algorithm like PSG. Chapter 5 describes the software development process used for implementing PSG.

Chapter 6 describes the problem domains that PSG was used with. The chapter opens with a well-known problem from graph theory and moves on to identify the issues concerning the built-in self-test of digital circuits and the generation of suitable test patterns. The so-called "Cellular Automata Problem" is then formally stated and defined.

Chapter 7 sets the stage for a comprehensive set of experiments designed to analyze the performance of PSG from all angles. In particular, the algorithm is analyzed from the point of view of parallel systems, evaluating its speedup and scalability, and from the point of view of VLSI design, evaluating the algorithm's success at evolving test pattern generators for built-in self-test.

The final chapter is a place for review and reflection, drawing conclusions from the results of the work, and stating some of the open questions and suggestions for further work.

2 Evolutionary algorithms

Since the early days of computer science, researchers have had the idea to use the principles of natural selection to solve abstract problems. As time went on, and the field widened to include such research areas as artificial life [1], the term *evolutionary algorithms* was increasingly used as an umbrella term covering all search and optimization algorithms inspired by the principles of natural selection.

In this chapter, we will briefly introduce the many facets of evolutionary algorithms, their common principles and specific varieties. In the second section of the chapter, one particular algorithm of interest to us, the *Selfish Gene Algorithm*, will be closely examined.

2.1 Evolutionary and genetic algorithms

A term like *evolutionary algorithms* does not have one single definition, and over time it has been used to describe a wide variety of algorithms. One kind of algorithm that is widely regarded as the quintessential evolutionary algorithm is the *genetic algorithm*. Genetic algorithms, made popular through the work of John Holland in the 1970s, combine many of the features now seen as cornerstones of evolutionary algorithms: a population of contending individuals that represent (partial) solutions to a problem, and evolutionary operations such as selection and recombination that are applied iteratively to these individuals.

These and other essential features of evolutionary algorithms will be described in the next section, while genetic algorithms will be revisited in section 2.1.2.

2.1.1 General principles

The basis of any evolutionary algorithm is the notion of a population of individuals. The population, as well as the individuals, can be either explicit or implicit. To this population the principle of natural selection is applied. Natural selection is a process by which some individuals produce more offspring than others. The likelihood of an individual passing on its genetic material to the next generation is known as its *fitness*. Quantifying fitness is far from straightforward, as it includes not only internal qualities of the individual itself, but also its environment, its actions, and a large amount of chance.

In nature, it is not always easy to see which individuals are fit and which are not.[1] In most evolutionary algorithms there is a deterministic fitness function, which is essentially a mapping from the set of all individuals to the set of real numbers. Other algorithms opt for a tournament mechanism where, rather than assigning an absolute fitness value to each individual, several individuals are compared head-to-head and a winner is picked. Regardless of the specifics, the selection principle provides the algorithm with the means to rank and order its population by fitness.

The second unifying notion of an evolutionary algorithm is the principle of *variation* and iterative evaluation. In one iteration, or generation, of the algorithm a number of operations are applied to the population. The purpose of these operations is to make some small variations in the composition of the population. The two most popular variation mechanisms are recombination and mutation, which will be described in section 2.1.2; however, many other mechanisms exist and we will meet some of them when we examine the Selfish Gene Algorithm in section 2.2.

[1] This is the origin of Charles Darwin's phrase "preservation of favoured races," which has sadly been taken out of context by many, to make some political point [2].

Summing up, an evolutionary algorithm is characterized by:

- A population of individuals which represent (partial) solutions to a problem.
- A mechanism to order individuals according to their fitness.
- A set of operations that are applied to the population every generation.
- Variation operations such as recombination, mutation, etc.

2.1.2 Types of evolutionary algorithms

In their overview of evolutionary computation [3], Bäck and Fogel recognize three important subsets of evolutionary algorithms: genetic algorithms, genetic programming and evolutionary strategies, although it should be noted that many more classifications and partitionings exist.

Genetic algorithms

The most popular evolutionary algorithm is the genetic algorithm. In a genetic algorithm, a population of (usually) constant size is filled with individuals that represent solutions to an optimization problem. These individuals are traditionally represented by binary strings, although real-valued implementations also exist.

Borrowing biological terminology, the representation of an individual is known as its *genotype* and consists of a string of *genes* called the *chromosome*, with each gene occupying a distinct *locus* on the chromosome.[2] The number of different genes that can occupy a locus is often finite: in the case of binary strings, a gene can have the value 0 or 1. The different values that can appear at a locus are sometimes called *alleles*.

Selection is usually done using an absolute fitness function, and produces pairs of individuals that are then recombined using a crossover operator. This operator swaps parts of the genotype of the two individuals. Another operator applied to the population is the mutation operator, which randomly alters some genes of some individuals.

At each generation, the operators are used to create a set of offspring from the set of individuals. A mechanism which is sometimes called *elitism* is then used to determine which of the parents and which of the offspring are allowed to propagate to the next generation.

The algorithm continues executing generation upon generation until some stopping condition is met. This may occur when the population has become homogeneous, when an individual with a certain fitness value is found, or simply when some time limit has been reached.

Genetic programming

The goal of genetic programming is not to evolve a specific solution to a problem, but to evolve a system or algorithm capable of solving a given problem. Traditionally, individuals in genetic programming represented finite-state machines; modern methods are capable of evolving complex computer programs.

[2] It should be noted that in nature, only prokaryotic lifeforms such as bacteria have a genotype consisting of one single chromosome. Other organisms have several; for instance, a human has 46 chromosomes and a goldfish has more than 100. Furthermore, it is important to remember that genetic material is found not only on our chromosomes, and that chromosomes contain more material than just genes.

While the principles of evolutionary computation (selection, variation) are identical to those in genetic algorithms, the actual operations in genetic programming depend heavily on the semantics of the genotype, whereas genetic algorithms are traditionally more oblivious of what the chromosome actually represents.

Evolutionary strategies

In contrast to genetic algorithms which are used to evolve solutions to a (discrete, combinatorial) problem, the purpose of evolutionary strategies is optimization of parameters. A much larger role is given to the mutation operator; since most implementations of evolutionary strategies operate in a real-valued search space, mutation consists of sampling data points from a normal distribution that somehow fits the population.

Other distinguishing features of evolutionary strategies are the clear distinction between a parent population and an offspring population, and the use of self-adaption mechanisms for the algorithm's runtime parameters. For another look at self-adaption, see section 4.2.

2.2 The Selfish Gene Algorithm

The *Selfish Gene Algorithm* (SG) is an evolutionary algorithm that was invented in 1998 by Corno et al. and is described in their paper "The Selfish Gene Algorithm: a New Evolutionary Optimization Strategy" [4]. The algorithm breaks away from canonical genetic algorithms by adopting a gene-centered view of evolution, a relatively young and somewhat controversial theory that has gained a following among biologists in recent years. The distinguishing feature of SG is the fact that the population is implicit, and that no operations such as recombination or elitism are necessary.

2.2.1 Biological rationale

The gene-centered view of evolution puts natural selection in a different light by taking the gene as the unit of evolution, rather than the individual. In his book "The Selfish Gene" [5], from which the algorithm derives its name, English biologist Richard Dawkins outlines the principles behind the gene-centered view: he argues that individual genes compete for survival and strive for appearance in future generations. Individuals exist only as "survival machines," providing the mechanisms by which successful genes can replicate and propagate. A successful gene is one that produces a good characteristic in its host, a characteristic that will increase the host's likelihood of reproducing and thus passing the successful gene on to the next generation.

Shifting the focus from individuals which are mortal and short-lived, to genes which are immortal and may survive indefinitely, the "slogan" of evolution—*survival of the fittest*—can be seen to depict a struggle between contending genes. Genes compete for appearance on individuals' chromosomes, while at the same time being forced to cooperate with one another: a gene might produce a good characteristic only in the presence of certain other genes. Dawkins calls this a form of "blind cooperation."

It should be noted that genes are in no way conscious entities acting out of some personal interests. Rather, their behavior emerges out of the actions of individuals.

2.2.2 The algorithm

In the Selfish Gene Algorithm, the idea of a population filled with individuals is replaced by a virtual population (VP) that is essentially an implicit, statistical model of a population. For all alleles on all loci, the VP holds information about the allele's frequency in the population.

Individuals are used as survival machines: they are constructed only for fitness evaluation and then destroyed. When an individual is fit, its genes are rewarded: the specific allele values making up the individual's genotype will see their frequency in the VP increase. Vice versa, genes belonging to unfit individuals will be penalized.

Since individuals serve no role other than fitness evaluation, a recombination operation is not needed; rather, recombination is implied by the way individuals are constructed from statistically probable genes in the gene pool. Likewise, elitism is absent. Mutation is kept, but is performed during selection (construction) of individuals.

2.2.3 Formal description

Letting n denote the number of genes (loci) in the genotype, a_i is the number of alleles available at locus i, and Σa is the total number of alleles across the genotype. The virtual population is a set of probability values $P_{i,j}$ where $1 \leq i \leq n$ and $1 \leq j \leq a_i$. Since these probabilities reflect the frequency of contending alleles, the probabilities for any locus must add up to 100%:

$$\sum_{k=1}^{a_i} P_{i,k} = 1 \tag{2.1}$$

In one generation, the algorithm constructs two "random" individuals and compares them in a tournament.[3] The winner of the tournament will see its genes rewarded: letting χ_i denote the allele present at locus i on the winner's chromosome, the frequency of that allele is increased:

$$P_{i,\chi_i} = P_{i,\chi_i} + \epsilon \tag{2.2}$$

All alleles on the winner's chromosome will see their frequencies increased by ϵ. At the same time, alleles on the loser's chromosome will see their frequencies decreased, again by ϵ.

Letting ρ denote the mutation probability, a random individual is constructed as follows: for each locus i, an allele will be chosen that is either random or proportional. The decision between random and proportional alleles is taken according to ρ: if a random number between 0 and 1 is greater than ρ, the allele will be proportional; otherwise, a random allele will be chosen.

In case of a proportional allele, the probability values $P_{i,k}, 1 \leq k \leq a_i$ are used to choose which allele to place on the chromosome. In case of a random allele, an allele is chosen from the range $1, 2, \ldots, a_i$ in a completely random way.

The Selfish Gene Algorithm also inctroduces the notion of *convergence*. A gene is said to have converged when one of the alleles at the gene's locus has a significantly higher frequency than all the others. In the algorithm, a gene is flagged as "converged" when the probability of one of its alleles exceeds a certain threshold. In the implementation described in chapter 4, this threshold is denoted by `convergenceThreshold`.

[3] Note that there is no strict need for an absolute fitness function.

2.2.4 Experimental analysis

Corno et al. implemented the Selfish Gene Algorithm and compared it head-to-head with a canonical genetic algorithm, solving the 0/1 Multiple Knapsack Problem. They found that SG was able to outperform the canonical genetic algorithm under certain conditions, and they conclude that "SG are easier to implement, have a smaller number of parameters to tune, do not rely on crossover operators, and are able to find more quickly optimal solutions. On the other hand, they tend to explore a smaller region of the search space, so they need to be inserted in some multi-start-like framework." [4]

Further work by Corno et al. is described in [6], where the researchers apply the algorithm to the much more difficult problem of evolving cellular automata. This is analyzed in greater detail in section 6.2.6.

3 Parallel evolutionary algorithms

When an evolutionary algorithm is adapted to utilize an arbitrary number of processors during execution, a *parallel evolutionary algorithm* (PEA) is created. PEA's are the topic of this chapter.

In the first section, a brief introduction to the field of parallel systems and algorithms is given, to familiarize the reader with its definitions, its metrics and its challenges. The second section introduces PEA's proper, while the third section examines the different varieties of PEA architectures more closely. In the last section of this chapter we will outline some of the work that has been done on the numerical analysis of PEA's, and what that means for the development of the Parallel Selfish Gene Algorithm.

3.1 Introduction to parallel systems

A parallel system is a collection of processing elements (PE's) that communicate and cooperate to solve challenging problems. Parallel systems come in all sizes, from the multicore processors in modern PC's, to dedicated supercomputers such as *BlueGene*, to distributed computing efforts organized over the internet. The term *parallel system* often refers to both the actual physical machine, and the algorithm implemented on it. In other words, a parallel system is a parallel algorithm together with hardware supporting multiple PE's.

Parallel systems can be classified according to various taxonomies. One of the oldest taxonomies is the taxonomy according to instruction and data flow known as *Flynn's taxonomy*, after Michael J. Flynn who proposed it in 1966. Flynn's taxonomy recognizes three types of parallel systems:

1. **Single instruction, multiple data** (SIMD), where all PE's execute the same series of machine instructions in lockstep, each operating on a different data stream. This type of system was popular in the past but is now used only in specialized systems such as 3-D graphics processors (GPU's).

2. **Multiple instruction, single data** (MISD), a rarely used form of parallel computing, where PE's receive the same data stream but execute different instructions on the data.

3. **Multiple instruction, multiple data** (MIMD), where PE's execute different sets of instructions on different data streams. This is the most common type of parallel system in use today. It is further divided into two subcategories:

    3a. **Single program, multiple data** (SPMD), where the same program is executed on all PE's, but the PE's are free to follow a different execution path and use different data. The Parallel Selfish Gene Algorithm described in chapter 4 is SPMD.

    3b. **Multiple program, multiple data** (MPMD), the most detached model, where the PE's are completely autonomous, running different programs working with different data sets. Distributed (web-based) systems based on a client-server architecture can be classified as MPMD.

Another taxonomy in use is the taxonomy according to memory organization. This taxonomy divides parallel systems into two groups:

1. **Shared memory** systems such as the multicore or multiprocessor systems which are in common use as PC's and servers. In these systems, the communication between PE's is done by reading and writing data blocks.

2. **Distributed memory** systems such as the computing cluster used for the experiments in chapter 7. In these systems, the communication between PE's consists of sending and receiving messages.

3.1.1 Parallel performance metrics

Parallel systems are used for two purposes:

1. To solve a challenging problem faster than on a sequential system.
2. To find certain "better" solutions to a problem that cannot be found by a sequential system in a reasonable amount of time.

The first purpose is the one that is traditionally seen as the primary purpose; although the second one is also very important, especially where heuristic methods such as evolutionary algorithms are concerned, the focus of researchers and implementers is mainly on the time savings that can be obtained. In this context, two metrics are used: *speedup* and *scalability*.

The speedup S of a parallel algorithm solving a certain problem is defined as the ratio of the running time SU of the fastest known sequential algorithm and the running time T of the parallel algorithm [7].

$$S(n,p) = \frac{SU(n)}{T(n,p)} \tag{3.1}$$

where n is the dimensionality of the problem and p the number of PE's. The goal of parallel system designers is to achieve linear speedups; that is, if we solve the problem using k processors, we will find the result k times faster than using a sequential system.

$$S(n,k) = \frac{SU(n)}{T(n,k)} = \frac{SU(N)}{\left(\frac{SU(N)}{k}\right)} = k$$

Formally, linear speedup is defined as follows, using "big O" notation:

$$S(n,p) = \Theta(p) \tag{3.2}$$

In the above equations, special attention must be paid to the definition of "time." Some researchers measure speedup by measuring the CPU time consumed by the PE's. Others prefer measuring absolute "wall-clock" time. In recent years, commercial CPU's have reached astonishing throughput rates, whereas performance of interconnection equipment has progressed at a much slower pace. Together with clever mechanisms such as DMA transfers reducing the CPU's workload, this leads to a situation where the performance of a parallel system is all but defined by its communication system. For these and other reasons, wall-clock time will be used as the time measure in this document.

The holy grail of parallel systems is *superlinear speedup*, defined as the case when a parallel algorithm on p processors is more than p times faster than the sequential algorithm.

$$S(n,p) = \Omega(p) \tag{3.3}$$

Superlinear speedup is a rare occurance, and is usually caused by certain factors that severely penalize the sequential algorithm. An example of this is a search algorithm that would, in a sequential implementation, require more memory than is available in the system, causing extensive swapping of virtual memory pages; when the search space is segmented for use in a parallel system, the memory requirements are lower,

speeding up the execution. An example of an algorithmic penalty for the sequential solution relates to algorithms with pruning such as Branch & Bound: the sequential implementation might traverse a large non-optimal branch of the search space that, in a parallel system, may get prematurely pruned by another processor.

Closely related to speedup are the notions of *parallel cost*, denoted by C, and *efficiency*, denoted by E.

$$E(n,p) = \frac{SU(n)}{C(n,p)} = \frac{S(n,p) \cdot T(n,p)}{p \cdot T(n,p)} = \frac{S(n,p)}{p} \tag{3.4}$$

Scalability is loosely defined as the ability to utilize increasing numbers of processors effectively [7]. A perfectly scalable algorithm is one where E remains constant regardless of n and p. In reality, parallel systems can only make good use of a large number of PE's if n is sufficiently large. When more PE's are added to a small problem, the running time will stop decreasing at a certain point, and might even increase when the communications overhead starts consuming more and more of the system's resources. This phenomenon is called "parallel slowdown" and is similar in many ways to the phenomenon, seen in human resources management, that adding more manpower to a delayed project will often delay it further [8].

3.2 Overview of parallel evolutionary algorithms

Parallel implementations of evolutionary algorithms have been around for many years. One of the first implementations was described by Albert Bethke in 1976 [9], just one year after John Holland's pivotal work "Adaptation in Natural and Artificial Systems" [10], which introduced genetic algorithms to the world at large. One reason for the success and popularity of PEA's is the fact that the biological foundation behind evolutionary algorithms seems to suggest parallelization; after all, in nature each organism operates independently from the others, competing for shared resources in a manner similar to processing elements in a parallel system.

Another reason for the success and popularity of PEA's relates to the second and traditionally overlooked purpose of using a parallel system: the observation that a parallel implementation can often lead to better results that could not be obtained by sequential methods. In his review of PEA's [11], Marco Tomassini writes:

> Parallel evolutionary algorithms seem to be more in line with their natural counterparts and thus might yield algorithmic benefits besides the added computational power. [...] In general, it has been found experimentally that parallel genetic algorithms, apart from being significantly faster, may help in alleviating the premature convergence problem and are effective for multimodal optimization.

3.2.1 Speedup and solution quality

In the previous section, we introduced the definition of speedup (equation (3.1)). The problem with this definition is that it requires that both algorithms solve the same problem and reach the same solution. It is difficult to guarantee this second condition when heuristic methods such as evolutionary algorithms are used.

Many researchers measure the speedup of a PEA as the ratio of convergence times of the sequential and parallel implementations, i.e. the time needed for the population to become (mostly) homogeneous. While

this is a useful metric, it is worth noting that, strictly speaking, such a ratio should not be called "speedup," since the solutions reached cannot be guaranteed to be of the same quality.

Using this convergence-based approach for analyzing PEA's, superlinear speedup has frequently been claimed. Unfortunately many of these cases have the parallel algorithm reaching lower quality solutions than its sequential counterpart. Conversely, PEA's capable of impressive gains in solution quality display rather poor speedups.

Both Cantú-Paz [12] and Cao, et al. recognize that, as a general rule of thumb, "reduction in execution time comes at the cost of a degraded solution quality." [13] In other words, it appears that time savings and solution quality are inversely related to each other. If we want linear or even superlinear speedup, we must be prepared to sacrifice some solution quality; if we want to achieve high-quality solutions, we must accept that convergence-based speedup will be of the form $S(n, p) = O(p)$.

Another way of looking at speedup for a PEA involves measuring the time it takes the (sequential, parallel) algorithm to reach a solution of a certain quality. The feasibility of such an approach depends on the problem domain; if fitness values fall within a clearly defined range, let's say 0–100%, then it is easy to set a fitness threshold somewhere along this range and measure how long it takes the algorithm to evolve a population where the average fitness exceeds the threshold. Other problem domains might distinguish between valid and invalid solutions.[1] In this case, we may measure how long it takes until some proportion of the population represents valid solutions.

3.3 Classes of PEA architectures

PEA's can be classified according to their system architecture. There are five such classes:

1. Single-population master/slave
2. Single-population fine-grained
3. Multiple-deme coarse-grained, also known as the "island model"
4. Hierarchical hybrids
5. "Embarrassingly parallel" evolutionary algorithms

3.3.1 Single-population master/slave PEA's

This class of PEA's is the closest to sequential evolutionary algorithms. In the master/slave model, one processor (the master) maintains the population and performs the genetic operations such as mutation and recombination. The other processors (the slaves) are used solely for fitness evaluation. When the master generates a new individual, it sends this individual to one of the slaves; the slave executes the fitness function and sends the fitness value back to the master. Since the population is maintained by only one processor, this algorithm reaches exactly the same solution as a sequential algorithm, albeit in a reduced time.

[1] A good example of this is the Knapsack Problem.

3.3.2 Single-population fine-grained PEA's

Fine-grained PEA's are commonly implemented on dedicated supercomputers where processors are arranged in a regular topology such as a toroid or hypercube. Each processing element represents one individual in the population. The algorithm is synchronous, with processors exchanging genetic material in lockstep with their immediate neighbors.

3.3.3 Multiple-deme coarse-grained PEA's

Coarse-grained PEA's form the most popular class of PEA's [14, 12]. In this algorithm, each processor is assigned a portion of the population, which is called a *deme*. The demes evolve independently and from time to time some individuals migrate from one deme to another. This class is also known as the "island model" since it resembles the population genetics of an archipelago of islands. In fact, island model PEA's exhibit certain phenomena, occurring naturally in biological populations, that are not observed in single-population algorithms.

The biological rationale behind the island model is an evolutionary process known as *allopatric speciation* [2]. In plain terms, this means that a geographical separation between individuals in a population allows them to diverge from each other, creating new species.

A good example can be seen on the Galápagos Islands, where the narrow strips of ocean between the islands led to a wide diversity of small birds known collectively as "Darwin's finches." Figure 3.1 shows the fourteen species that make up Darwin's finches; although similar, these are fourteen distinct species that all evolved from a single immigrant species that reached the Galápagos Islands from the South American mainland, just a few million years ago.

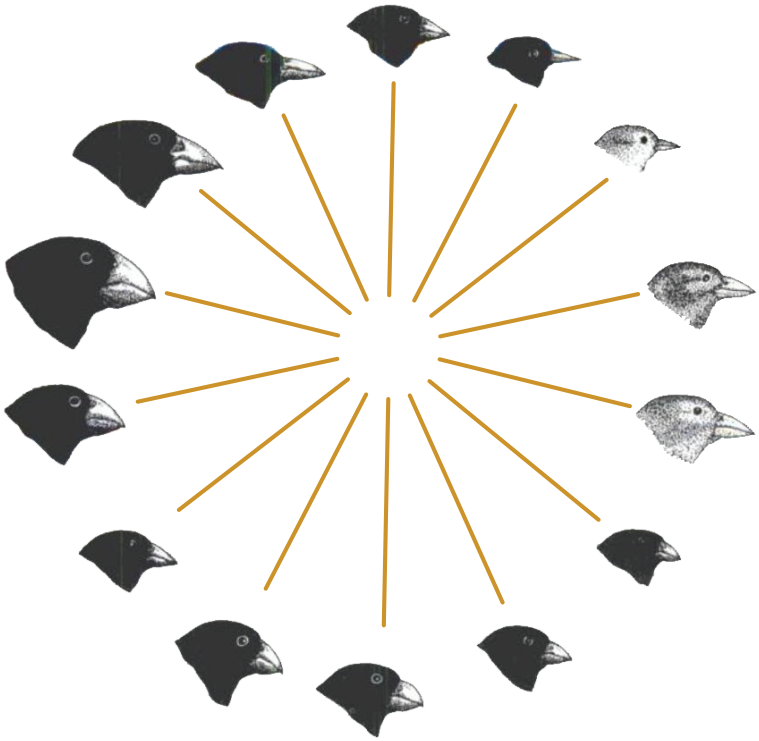


Figure 3.1: The fourteen species that make up Darwin's finches. Adapted from [15].

Implementations of the island model vary in their migration strategy and can be described as synchronous or asynchronous. In synchronous implementations migration occurs at specific time intervals, whereas in asynchronous implementations migration occurs when some internally measured level of convergence has been reached.

Cantú-Paz [12] states that an island model PEA performs optimally when migration occurs after demes have converged, implying that asynchronous implementations would be a better choice. Liu, et al. [16] concur that the end-of-epoch barriers in a synchronous PEA result in considerable overhead. Tomassini [11] notes that asynchronous migration is the preferred strategy for algorithms that follow the "steady-state reproduction" model, where a small number of offspring replace other individuals in the population as soon as they are produced instead of replacing the whole population after all the individuals have gone through a full generational cycle.

3.3.4 Hierarchical hybrid PEA's

This class of PEA's combines multiple demes with single-population PEA's. As the name suggests, they consist of more than one layer. At the upper level they are multiple-deme coarse-grained PEA's, while at the lower level they are single-population PEA's (either master/slave or fine-grained). This gives the algorithmic benefits of the island model along with the computational efficiency of single-population PEA's.

3.3.5 "Embarrassingly parallel" evolutionary algorithms

There is also a class of "embarrassingly parallel" algorithms that are essentially multiple-deme PEA's without any migration. In other words, each processor executes a sequential evolutionary algorithm and the final solution is taken as the best solution among all processors. Not only are these algorithms trivial to implement, studies have shown that they lead to a poor solution quality [17].

3.4 Numerical analysis

A comprehensive theoretical analysis of PEA's is given by Erick Cantú-Paz in his book "Efficient and Accurate Parallel Genetic Algorithms" [12], which was based on his doctoral work under the supervision of evolutionary algorithms pioneer David Goldberg. The book introduces the classes of PEA architectures and, starting with the master/slave model, derives a set of equations for the optimal population size and the optimal interconnection network, among other things. The theoretical hypotheses are accompanied by experimental verification.

We will not reproduce the analysis here, since the equations are of limited use in a non-standard PEA like the Parallel Selfish Gene Algorithm. However, a few conclusions are of interest to us; first of all, the author argues that unless a dedicated supercomputer is available, an island model implementation is likely to give the best results.

Limiting ourselves to island model PEA's, we see that the topology of the interconnection network has very little effect on the behavior of the PEA. Rather, it is the degree of the network's graph that has a significant influence on the PEA's performance; the best (and fastest) results are obtained when the degree is approximately $p/2$, i.e. each deme has a direct connection with approximately half of the others.

4 The Parallel Selfish Gene Algorithm

In the previous chapters, we have become acquinted with the field of evolutionary algorithms and its extension, parallel evolutionary algorithms (PEA's). We have met the Selfish Gene Algorithm (SG), an unusual evolutionary algorithm based on recent developments in modern evolutionary biology.

In this chapter, we will describe the Parallel Selfish Gene Algorithm (PSG), a new evolutionary algorithm intended as a parallel version of SG. In the first section, the architecture of PSG will be discussed. The second section introduces the concept of self-adaption and shows how it can benefit a genetic algorithm such as PSG.

4.1 System architecture

PSG is based on the "island model" of PEA's. There are two reasons for this choice of architecture; first of all, there is the observation that island model PEA's have the potential to reach solutions of higher quality than may be reached by sequential methods or single-population PEA's. Another reason is the fact that the island model is a more appropriate choice for implementation on a Beowulf-style cluster of PC systems [11].

Following the discussion in section 3.3.3, an asynchronous migration strategy was chosen. It is worth noting that the Selfish Gene Algorithm adjusts only a handful of allele probabilities at each generation, meaning that SG resembles the steady-state reproduction model and should therefore benefit greatly from an asynchronous migration strategy.

Most asynchronous PEA's operate by migrating a certain number of individuals when the deme has reached some level of convergence. In PSG, the communication packet consists of the allele values of converged genes. This corresponds with the philosophy behind the gene-centered view of evolution, namely that it is the genes themselves that are important, not the individuals.

Communication packets are sent once a deme has reached a certain level of convergence. In general, the communication packet can be described as follows:

> A communication packet consisting of the allele values of X converged genes is sent when Y genes have converged, where $X \leq Y \leq n$.

X and Y are called the *epoch thresholds*. For each consecutive communication event these thresholds are slowly increased. The period between two communication events is called an *epoch*. After any epoch e, the thresholds are increased according to the following formulas.

$$X_{e+1} = X_e + \left\lceil \frac{X_e}{\gamma} \right\rceil \tag{4.1a}$$

$$Y_{e+1} = Y_e + \left\lceil \frac{Y_e}{\gamma} \right\rceil \tag{4.1b}$$

where $\gamma \geq 1$ is called the *epoch length modifier*. For the first epoch, the thresholds X_0 and Y_0 are given as parameters of the algorithm. Note that special care must be taken to ensure that X and Y never exceed n.

Communicating demes send their communication packets to δ nearest neighbors, where δ is the degree of the (simulated) topology. The actual topology does not matter: Cantú-Paz [12] found that topologies of the same degree generally reach the same solutions in the same amount of time.

On the side of the recipient, incoming allele values are used to adjust the recipient's virtual population. For each incoming allele value, the probability of that allele is increased, and the probabilities of all other alleles on the same gene are decreased. Denoting the locus of the incoming allele by i and the specific allele value by j, probabilities are adjusted as follows:[1]

$$P_{i,j} = \frac{P_{i,j} + \mu}{1 + \mu} \tag{4.2a}$$

$$\forall k,\ 1 \leq k \leq a_i, k \neq j,\ \ P_{i,k} = \frac{P_{i,k}}{1 + \mu} \tag{4.2b}$$

where $\mu > 0$ is called the *migrant mixing coefficient*.

4.1.1 Pros and cons

Advantages of this strategy are:

- The communication packet is kept small. The size of the communication packet $s_e = O(X_e) = O(n)$.
- No random selection: sending only the allele values for converged genes maximizes the information content of the message. By contrast, nonconverged genes are highly stochastic and therefore contain less information.
- The "communication after convergence" property is maintained.

Disadvantages are:

- Dependencies between genes may be broken. It is not unthinkable that alleles might thrive only in the presence of other, not yet converged alleles on other genes. If these thriving alleles were to be sent to another deme without the alleles on which they depend, they might cause a degradation of solution quality in the recipient deme. This issue may be addressed by tuning the epoch thresholds X_0 and Y_0.
- If μ is too small, migrant alleles might not be given a chance to survive in the recipient demes, making the communication futile.
- If μ is too low, migrant alleles might "kill" promising alleles in the recipient deme, and information may be lost.
- If no further convergence (or worse, divergence) occurs in a deme after it has communicated, the deme will never reach its next epoch threshold and will continue to operate in isolation, only accepting migrants and never sending any. This problem can be alleviated by imposing a limit on the number of generations a deme may execute while waiting for the next epoch; if this limit is reached, the deme will send out migrants regardless of the number of converged genes. In this case, the number of alleles in the communication packet may be (even significantly) lower than X_e, since there is no guarantee for the number of converged alleles.

[1] Note that the "=" symbol in these equations denotes assignment and not equality.

4.1.2 Parameters of PSG

The algorithm inherits the following parameters from its sequential version:

1. Number of genes n.
2. Number of alleles per gene a_i, total number of alleles Σa.
3. Tournament reward ϵ
4. `convergenceThreshold`, the probability level at which convergence is flagged.
5. Mutation rate ρ.

The parallel implementation introduces the following new parameters:

6. Number of processors p.
7. Degree δ.
8. Epoch thresholds X_0 and Y_0.
9. Migrant mixing coefficient μ.
10. Epoch length modifier γ.
11. `epochGenerationLimit`, the maximal number of generations in one epoch.
12. Various implementation-specific parameters, such as how often to check for incoming messages, when to terminate execution, etc.

4.2 Self-adaption

One of the drawbacks of heuristic methods such as evolutionary algorithms is the high number of parameters to tune. The principle of self-adaption attempts to reduce this number by providing mechanisms by which the algorithm can set its own parameters without the need for human involvement.

Self-adaption has its roots in the field of evolutionary strategies. Just as an evolutionary strategies algorithm intelligently finds optimal parameters for its problem domain, so too are the parameters of the algorithm itself adapted to maximize the algorithm's utility. In [18, 19], Thomas Bäck examines the ways self-adaption mechanisms can be used for the mutation rate of genetic algorithms.

4.2.1 Methods of self-adaption

All self-adaption mechanism are based on the notion that there is an indirect link between favorable control parameters and the fitness values of individuals. Three distinct methods exist:

1. Deterministic self-adaption
2. Dynamic self-adaption
3. Dynamic self-adaption by genome encoding

Deterministic self-adaption

The simplest way of implementing self-adaption is by introducing a deterministic function $\rho(n, t, T)$ which generates a time-dependent schedule for the mutation rate ρ according the chromosome length n, the number of generations t, and some predefined generation limit T. In [19], an example of such a function is given as:

$$\rho(n, t, T) = \left(2 + \frac{n-2}{T-1}t\right)^{-1} \tag{4.3}$$

Despite its simplicity, the deterministic schedule was found to give better results than a dynamic self-adaption method.

Dynamic self-adaption

Dynamic self-adaption relies on feedback information from the algorithm to adjust its mutation rate. Generally, the average fitness in the population is tracked, but other runtime metrics are also used. A common practice is to let each individual maintain its own mutation rate. Due to the difficulties of constructing a mapping between individuals and populations on one side and mutatation rates on the other, new parameters have to be introduced that govern this mapping. This of course undermines one of the main reasons for using self-adaption: the reduction in input parameters.

Dynamic self-adaption by genome encoding

A novel approach that has not seen much analysis involves letting the algorithm set its mutation rate using the algorithm's own principles of natural selection. In this method, mutation rates are appended to the problem domain's chromosome and evolved along with the solutions to the problem. The aim of the mechanism is to exploit the correlation between optimal control parameters and individual fitness.

4.2.2 Self-adaption in PEA's

In [20], Skinner et al. describe an island model PEA featuring self-adaption. Their algorithm uses a synchronous migration strategy, migrating individuals between demes when all demes have executed a preset number of generations E. Following the observation that deterministic self-adaption gives the best results, they adapted Bäck's equation (4.3), with E substituted for the generation limit.

$$\rho(n, t, E, l) = \left(\frac{n \cdot l}{20} + \frac{2 \cdot n \cdot l}{E-1}t\right)^{-1} \tag{4.4}$$

where l denotes the number of bits used to encode each gene.

Results of their experiment were favorable, although it should be noted that no comparison was made between the algorithm's performance using self-adaption and its performance using other means of setting ρ.

4.2.3 Self-adaption in PSG

In PSG, a deterministic self-adaption mechanism is used, based on equations (4.3) and (4.4). Since the algorithm resembles the steady-state reproduction model and uses an asynchronous migration strategy,

the generation limit was eliminated from the function. Instead, SG's notion of convergence is exploited as a promising source of feedback, using the assumption that there is a correlation between the level of convergence and solution quality. Letting $c \leq n$ denote the number of converged alleles in the deme, the mutation probability may be calculated as follows:

$$\rho(n, c, t) = \left(2 + \frac{n-2}{n}c + \frac{f(t)}{n}\right)^{-1} \tag{4.5}$$

where $f(t)$ is an (as yet unknown) function of t. In order to find $f(t)$, experimental analysis was used. It was found that $f(t) = \Theta(t)$ caused the mutation rate to drop too quickly, whereas $f(t) = \Theta(\log t)$ had the opposite effect of the mutation rate showing very little time-dependence. As a compromise, a function $f(t) = \omega(\log t) = o(t)$ was needed, which led to the following final formula for the mutation rate schedule:

$$\rho(n, c, t) = \left(2 + \frac{n-2}{n}c + \frac{\sqrt{t}}{n}\right)^{-1} \tag{4.6}$$

5 Software development

In the previous chapter we defined the Parallel Selfish Gene Algorithm (PSG). Stepping away momentarily from the field of problems and algorithms, we cross into the field of software engineering to take a closer look at the development process by which PSG was implemented.

Development was split up into three *phases*; at the end of each phase, a fully functional application was delivered. The phases themselves were based on traditional "waterfall" cycles consisting of requirements, design, implementation, and testing. In some phases the implementation and testing cycles were further divided into *iterations*, each iteration comprising both implementation and thorough testing. Care was taken to deliver functional, tested code at the end of each iteration: doing so reduces the risk of bugs being carried over from one iteration into the next. Figure 5.1 shows a Gantt chart for the whole development process. The chart is highly simplified for reasons of space.

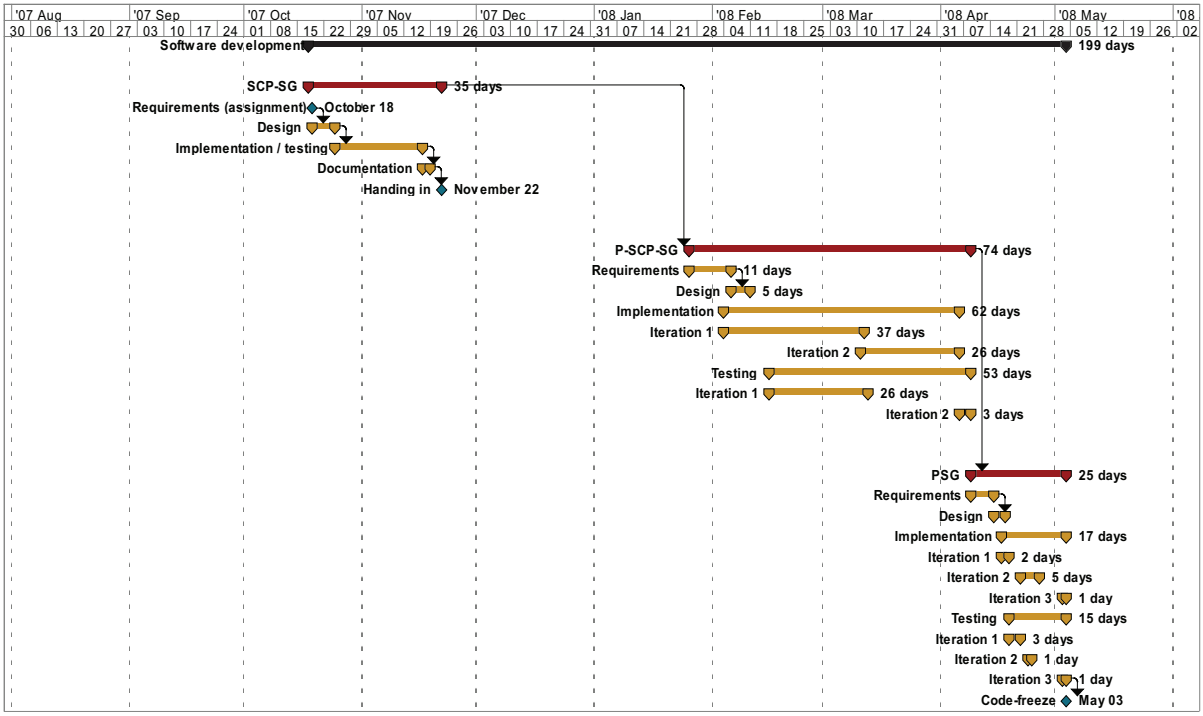


Figure 5.1: Simplified Gantt chart of the PSG software development process.

5.1 Phase 1: SCP-SG

Phase 1 started as a homework assignment for a course on *Softcomputing* (SCP). The assignment was to develop an evolutionary algorithm for solving the Vertex Cover Problem (see section 6.1). Thus the following (self-imposed) requirements were set for phase 1:

- Implement the Selfish Gene Algorithm for solving the Vertex Cover Problem.
- The design shall be modular, making it easy to add functionality.
- The implementation shall feature an abstract interface for problem domains, making it easy to adapt the algorithm to new problems.

- It will be possible to compile and execute the application on a Linux computing cluster.

SCP-SG was implemented in ISO C++ [21]. At the end of phase 1, the project contained 922 lines of source code in 24 files. SCP-SG has the following structure:

- The class `SgEngine` is the heart of the program. It can load a problem instance, execute a number of generations, and keep track of the best solution found.
- The class `ProblemDomain` is an abstract representation of a problem domain. Because it is an abstract representation, the fitness function is undefined.
- The class `VertexCoverProblem` is derived from ProblemDomain and contains the fitness function described in section 6.1.
- The classes `AdjacencyList` and `AdjacencyMatrix` are used by `VertexCoverProblem` to access the problem instance's graph.
- The class `Population` contains the virtual population. This class also handles the selection of individuals for the purpose of fitness evaluation.
- The class `StoppingConditions` implements a mechanism for determining when to stop computation. Three kinds of stopping criteria are defined: stopping after a specific number of generations, stopping after a specific number of seconds of wall-clock time have elapsed, and stopping when a specific number of alleles has converged.
- The class `Parameters` is a singleton class storing runtime parameters such as the mutation rate and the convergence threshold.
- The class `Random` is an abstract interface to a pseudo-random number generator.
- The classes `StdRandom` and `BoostRandom` are derived from `Random` and respectively use the standard C library and the Boost C++ libraries [22] for generating random numbers.

5.2 Phase 2: P-SCP-SG

Phase 2 constituted the biggest effort. The goal of phase 2 was to extend SCP-SG and create a parallel implementation called P-SCP-SG. During the requirements and design phases the system architecture of PSG, described in section 4.1, was determined along with the communication protocol and the self-adaption mechanism.

Functional requirements for phase 2 are largely described in the previous chapter. Additionally one non-functional requirement was added, specifying the Message Passing Interface (MPI) [23] as the communications library to use.

Development was divided into two iterations. In addition to a number of changes, iteration 1 added the following new modules:

- The class `Communication`, which contains all the code for sending and receiving messages from neighboring PE's. The different message types used in the system are described in-depth in section 7.2.1.
- The class `Logger`, providing singleton access to the application's log files from anywhere in the source code.
- The classes `Statistics` and `GlobalStatistics`, providing singletons holding various runtime statistics such as the number of generations executed, etc.

Iteration 2 saw the implementation of the deterministic self-adaption mechanism, and integrated the new code from iteration 1 to create a stable build that could compile and execute without failure. At the end of phase 2, the project contained 2671 lines of source code in 40 files.

5.3 Phase 3: PSG

The final phase of software development had as its primary task the integration of a new problem domain: the Cellular Automata Problem, described in section 6.2. Implementation of the new problem domain was contained in the first iteration, with the remaining two iterations concentrating primarily on acceptance testing and small maintenance patches to ensure a stable codebase for the experiments described in chapter 7.

The following new modules were added:

- The class `CellularAutomaton`, providing the means to build cellular automata and extract test vectors from them.
- The class `TestVectors`, which facilitates checking coverage of test vectors and fault vectors.
- The class `BistProblem`, which implements the abstract problem domain interface developed in phase 1.
- The class `MessageStatistics`, one more addition to the group of modules collecting runtime statistics. In order to gain a meaningful insight into the performance of any algorithm, it is vital to collect comprehensive statistics while the program is running. `MessageStatistics`, along with its close cousins from phase 2, provides this.

With phase 3 complete, the application reached its final state. Following suspension of development, the project contained 3695 lines of source code in 61 files. The gradual growth of PSG through its three phases is indicated graphically in Figure 5.2.

5.4 Dependencies

Readers interested in compiling and executing PSG will be interested to learn what kind of dependencies the project has. First of all, it must be noted that one of the earliest requirements was that the application be designed for a Linux computing cluster, so no extra effort was taken to provide compatibility for other systems. However, since the source code strictly adheres to the ISO standard for the C++ programming language [21], compatibility should not be an issue.

In particular, two libraries are needed to compile and execute PSG.

1. The Boost libraries [22] were used wherever the standard C++ libraries did not provide the necessary functionality. Boost libraries are available on all popular platforms and are thoroughly tested by the community.

2. An implementation of the Message Passing Interface (MPI) [23] is needed, even for sequential operation. In an effort to guarantee compatibility, two leading MPI implementations were tested during development: MPICH and OpenMPI.

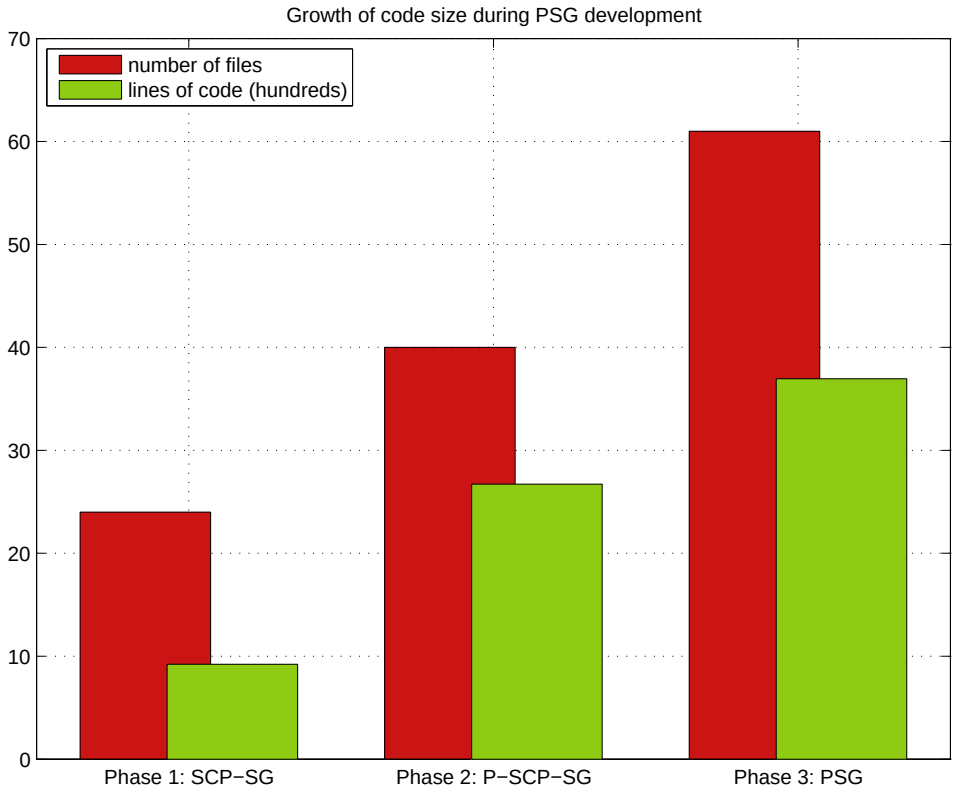


Figure 5.2: Growth of the number of files and the number of lines of code during software development.

The following development tools were used. Although they are not strictly necessary for compilation and execution, they are recommended.

3. The **CMake** build system [24] was used to maintain the project's makefiles and to automate compilation. Building the application using other build tools requires the user to manually create makefiles.

4. The **Darcs** revision control system [25] was used to maintain the project's repositories. Even in the case of a single developer, Darcs is an invaluable tool when source code is being developed and executed on several systems, each needing its own patches to the code. A web interface to the darcs repositories for all development phases is available at `http://dev.harpavieja.cz/darcsweb/`.

6 Problem domains

One of the strengths of evolutionary algorithms is the fact that they can be used to solve a wide range of optimization problems. In general, adapting an evolutionary algorithm to solve a different problem involves only three steps:

1. Defining the genome for the given problem. This includes setting the number of genes in the cromosome, the bit lengths, etc.

2. Writing the fitness function. The fitness function is the algorithm's mapping from genotype to phenotype and links the abstract genetic domain with the concrete problem domain.

3. Tuning the algorithm's parameters. Different problems have different characteristics, and settings that may work well for one problem might not work for another.

In this chapter, we will describe the two problem domains that were implemented for the Parallel Selfish Gene Algorithm. The first is the *Vertex Cover Problem*, a well-known problem from the realm of graph theory. The second problem is known as the *Cellular Automata Problem*, and it is a real-world problem from the field of VLSI design.

6.1 The Vertex Cover Problem

The Vertex Cover Problem is one of Karp's 21 NP-complete problems [26] and is defined as follows:

> A vertex cover for an undirected graph $G = (V, E)$ is a subset C of its vertices such that every edge in E has at least one endpoint in C. The Vertex Cover Problem is an optimization problem with the goal of finding a vertex cover of minimal size.

6.1.1 Data structures for graphs

Undirected graphs with n vertices are supplied to the program as an adjacency matrix: a symmetric binary matrix $G[n \times n]$, where $G(i, j) = 1$, iff there is an edge between i and j. Internally, the program uses an adjacency list; after loading the adjacency matrix, the program converts it to an adjacency list, and this is the structure that is used in fitness evaluations.

6.1.2 Definition of the genome

In the Selfish Gene Algorithm, a problem domain is described by the number of genes, and the number of possible alleles for each locus. Intuitively, the number of genes equals the number of vertices in the graph, and the number of possible alleles is equal to 2 for each locus. This means that each locus can have two values: 0 or 1. A value of 0 at locus i means that the i'th vertex in the graph is excluded from the vertex cover, a value of 1 means that it is included.

6.1.3 Fitness evaluation

Fitness evaluation is rather straightforward. An individual is described as an n-bit binary vector; from this we can construct subsets C and U of the graph. Subset $C \subseteq V$ is the set of vertices included in the vertex cover, and subset $U = V \setminus C$ is the set of vertices not included in the vertex cover.

For each vertex $v_c \in C$ we know that all edges with an endpoint in v_c are "covered," so no checking is needed. For each vertex $v_u \in U$ we have to check that all neighboring vertices are included in the vertex cover; in other words, that for each neighbor v_n of v_u, $v_n \in C$. If this holds, the cover is valid and the fitness value is equal to $n - |C|$. If the cover is not valid, the fitness is set to negative infinity.

6.1.4 Problem instances

Two sets of problem instances were created for use with the algorithm. The first set of problems is a set of random graphs created according to the Erdős-Rényi model. A MATLAB script was created based on the work of David Gleich [27]. The script creates graphs according to some parameters such as the number of vertices and the degree of connectivity.

The second set of problems was taken from [28]; it is a set of twenty interesting graphs specifically chosen for solving the Vertex Cover Problem.

All problem instances were simple text files representing an adjacency matrix.

6.2 The Cellular Automata Problem

The goal of the *Cellular Automata Problem* is to evolve the optimal configuration of a cellular automaton for use in the *built-in self-test* (BIST) of a combinational circuit. This problem is a lot harder than the Vertex Cover Problem, which is why it was chosen as the problem to be solved by the Parallel Selfish Gene Algorithm.

6.2.1 Built-in self-test (BIST)

In integrated circuits, a built-in self-test mechanism is a function by which a circuit is able to verify its functionality and detect any logic faults that may occur. The basic principle of BIST involves a controller inside the circuit which generates a sequence of input vectors which are placed on the circuit's (external) inputs. The controller then checks if the circuit exhibits the expected behavior and raises a signal if any error is detected.

A crucial part of the design of a BIST mechanism is the choice of the circuitry responsible for generating the test vectors. This circuitry is knows as the *test pattern generator* (TPG). Many different types of TPG have been developed [29], the most popular being linear feedback shift registers (LFSR) followed closely by cellular automata (CA). The latter are described in-depth in section 6.2.2.

There are various ways of testing a specific TPG in software. This generally involves applying the sequence of test vectors obtained from the TPG to a virtual representation of a benchmark circuit. Two approaches are popular: full fault-simulation and fault vector covering.

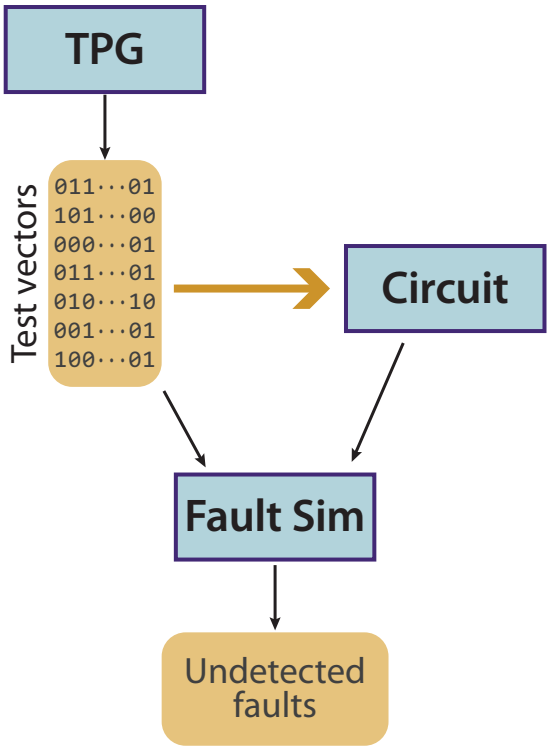


Figure 6.1: Full fault-simulation method of TPG evaluation.

Full fault-simulation

Figure 6.1 shows the principle of full fault-simulation. The heart of this method is a piece of software called the *fault simulator*. The fault simulator can load a benchmark circuit and introduce faults into it, mimicking the behavior of real hardware circuits that develop logic faults.

A series of test vectors is extracted from the TPG. These test vectors are given to the fault simulator; the fault simulator loads the circuit, introduces faults, and applies the test vectors to the circuit's inputs. It is then able to determine which of the introduced faults can not be detected by a BIST controller using this set of test vectors.

The result of the full fault-simulation method is that for any set of test vectors and any circuit, we are able to find the set of undetected faults. It stands to reason that when more and more test vectors are extracted from the TPG, the set of undetected faults becomes smaller and smaller. A common practice is to keep extracting test vectors until the set of undetected faults becomes empty. The number of test vectors necessary for achieving 100% fault coverage is then used as a quality measure for the TPG.

Fault vector covering

Unfortunately, full fault-simulation is a very time-consuming affair, and so another approach called *fault vector covering* is sometimes employed; the principles of this method are shown in Figure 6.2.

The idea behind fault vector covering is that the costly fault simulation is performed only once for each circuit, producing some intermediate result with which we can evaluate many different TPG's quickly. This intermediate result is a set of fault vectors V_F and is constructed as follows.

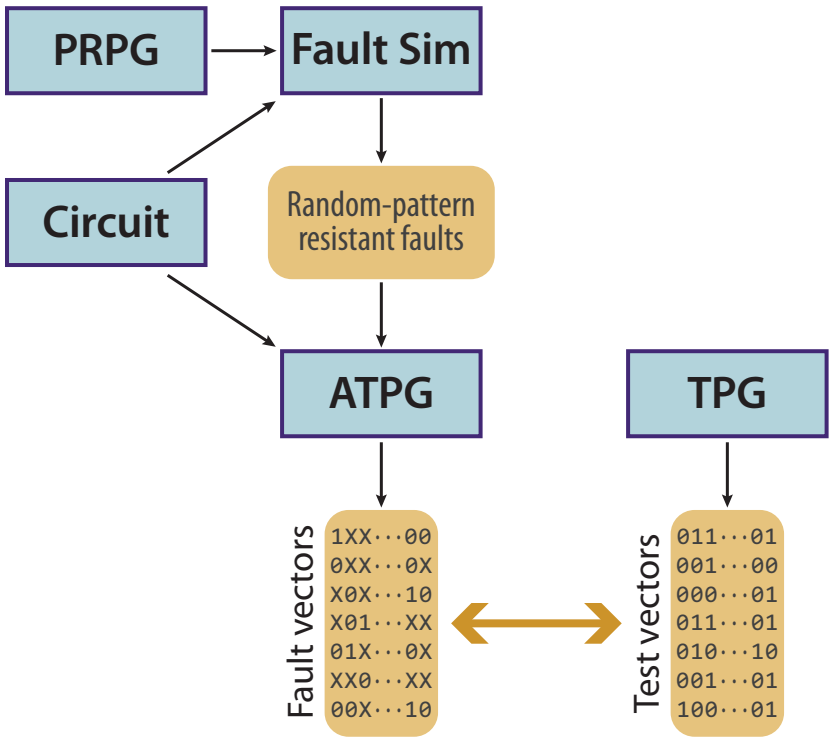


Figure 6.2: Fault vector covering method of TPG evaluation.

First, the fault simulator is employed in the usual way. A pseudo-random pattern generator (PRPG) is used as the TPG, and a fixed number of test vectors is extracted. Commonly, an LFSR is used as the PRPG in this step. The fault simulator loads the circuit and applies the vectors from the PRPG. The undetected faults returned by the fault simulator are known as *random-pattern resistant faults* (RPRF) since they are the faults that remain undetected when an arbitrary sequence of (pseudo-)random test vectors is applied to the circuit.

In the second step, a piece of software known as the *automatic test pattern generator* (ATPG) is used. For each fault in the set of RPRF's, the ATPG is able to calculate a so-called *fault vector*. A fault vector $F \in V_F$ is a binary vector with *don't-care* states: a $\{0, 1, X\}^i$ tuple, where i is the number of circuit inputs. When the non-X symbols in the fault vector are applied to their respective circuit inputs, the fault will be detected.

The ATPG creates one fault vector for each of the RPRF's and collects them in V_F, the set of fault vectors. This set can be seen to represent the "difficult" faults of the circuit; if a TPG is able to produce test vectors corresponding to these fault vectors, all the RPRF's from step 1 will be detected. The problem of TPG evaluation is now reduced to the problem of matching test vectors with fault vectors.

A test vector T from the TPG is said to "cover" a fault vector $F \in V_F$ if for every k, where $1 \leq k \leq i$, either $T_k = F_k$ or $F_k = X$. A given TPG may then be tested by taking a fixed number of test vectors from the TPG and determining how many of the fault vectors are covered by these test vectors. The number of uncovered fault vectors is used as a measure of the TPG's quality in a manner similar to the number of undetected faults in the case of full fault-simulation.

6.2.2 Cellular automata

A cellular automaton [30] is a system of cells composed in a grid. Each cell has a finite number of states. Time advances in discrete steps, and the state of each cell in the grid at time t is determined by the states of some of the cells at time $t-1$.

Cellular automata come in all shapes and sizes. The kind of CA that is most commonly used as a TPG is known as the *elementary cellular automaton*, which is a 1-bit CA (each cell has only two states) where the cells are arranged in a ring. The state of a cell at time t is calculated according to the states of three cells at time $t-1$: the state of the cell itself, the state of its left neighbor and the state of its right neighbor. In other words, the state transition rule of the cell is a binary function of three variables. There are $2^{2^3} = 2^8 = 256$ distinct rules, and each is identified by a number from 0 to 255 according to a scheme called the *Wolfram code*.

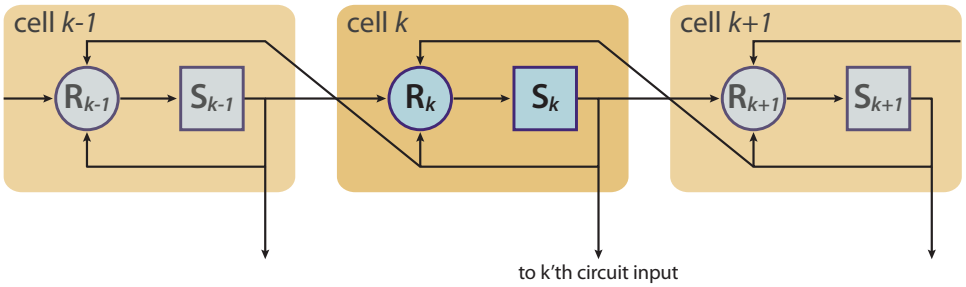


Figure 6.3: Cell structure of an elementary cellular automaton. S_k is the state of cell k, R_k is the state transition rule, a function of S_{k-1}, S_k and S_{k+1}.

Figure 6.3 shows the cell structure of an elementary cellular automaton. In the diagram, the state of the k'th cell is denoted as S_k; the cell's state transition rule R_k is a function of the cell's state and the neighboring cell's states: $R_k = f(S_{k-1}, S_k, S_{k+1})$. When the CA is used as a TPG, a test vector is constructed by concatenating the states S_k, where $1 \leq k \leq i$ at a given time t; in other words, each cell of the CA is connected to one circuit input. At time $t+1$, each state S_k is recalculated by its rule R_k according to the states S_{k-1}, S_k, S_{k+1} at time t.

Many of the 256 distinct CA rules are equivalent to one another, and only a few of them are useful in constructing TPG's. The most important ones are **Rule 60**, **Rule 90**, **Rule 150** and **Rule 240**. In the CA implementation for the Parallel Selfish Gene Algorithm, only these four rules are available.

6.2.3 Definition of the genome

The genome for the Cellular Automata Problem is based on the definition from Corno et al. [6]:

- For a cellular automaton of i cells, the number of genes $n = 2 \cdot i$.
- For each cell, one locus is used for the CA rule and one locus for the cell's initial state.
- "Rule" cells can have four different allele values corresponding to the four important rules introduced in the previous section.
- "Initial state" cells can have two values corresponding to the binary nature of the 1-bit CA.
- The total number of alleles $\Sigma a = 2 \cdot i + 4 \cdot i = 3 \cdot n$.

6.2.4 Fitness evaluation

Fitness values are calculated according to the fault vector covering method described in section 6.2.1. A set of 1000[1] test vectors V_T is generated by the CA and compared to the set of fault vectors V_F. This is done by iterating through the fault vectors; for each vector $F \in V_F$, an inner loop checks if any vector $T \in V_T$ covers F. When any vector is found to cover F, the fitness value is incremented by $|V_F|^{-1}$ and the next fault vector is checked. This produces fitness values in the range 0–1.

The worst-case time complexity for one fitness evaluation is $O(|V_F| \cdot |V_T| \cdot i) = O(|V_F| \cdot n)$ (since V_T is of constant size).

The final fitness value is a value in the range 0–1 indicating the percentage of fault vectors covered by this CA.

6.2.5 Problem instances

Circuit name	No. of circuit inputs i	No. of fault vectors $\lvert V_F \rvert$	Percentage of don't-cares
`c17`	5	13	40%
`c432`	36	112	51%
`c499`	41	111	1.6%
`c880`	60	132	75%
`c1355`	41	104	1.5%
`c1908`	33	121	49%
`c2670`	233	146	83%
`c3540`	50	121	79%
`c5315`	178	123	89%
`c6288`	32	94	27%
`c7552`	207	146	72%

Table 6.1: Problem instances constructed from combinational ISCAS'85 benchmark circuits.

Since the built-in self-test is a much-studied topic in VLSI design, a standard set of benchmarks known as the *ISCAS'85* benchmark set is available [31] and is used by most researchers. In order not to overload the computing cluster, which is shared by many users, only the combinational circuits from this set were studied. For each benchmark circuit, the set of fault vectors V_F was constructed as follows:

1. Using the program `atalanta-m` [32], originally developed at Virginia Tech [33, 34], a list of logic faults was created by fault-simulating the circuit using an LFSR as the TPG. The LFSR's generating polynomial, as well as its seed, were random integers of i bits generated using a Mersenne Twister pseudo-random number generator.

2. A set of fault vectors covering all faults was created, again using `atalanta-m`.

3. The resulting set of fault vectors often had duplicate items, so in the last step, the set of fault vectors was sorted lexicographically, and duplicates were stripped. The final set of fault vectors V_F was then written to a file and used as a problem instance for the Parallel Selfish Gene Algorithm. Note that

[1] This particular value was selected as it has been shown to lead to good coverage when $100 \leq |V_F| \leq 150$, while keeping the fitness evaluations short.

only the file containing the fault vectors is needed: the algorithm has no knowledge of the actual benchmark circuit.

Care was taken to ensure that the number of fault vectors was in the range 100–150 for all circuits in the benchmark set. However, for some "easy" circuits such as `c17` this was not possible, and for those circuits the set is smaller. For reference, the full set of combinational circuits in the ISCAS'85 benchmark set is given in Table 6.1, along with the number of circuit inputs i, the number of fault vectors $|V_F|$ and the percentage of don't-care symbols in the fault vectors. Figure 6.4 conveys this information in a graphical form.

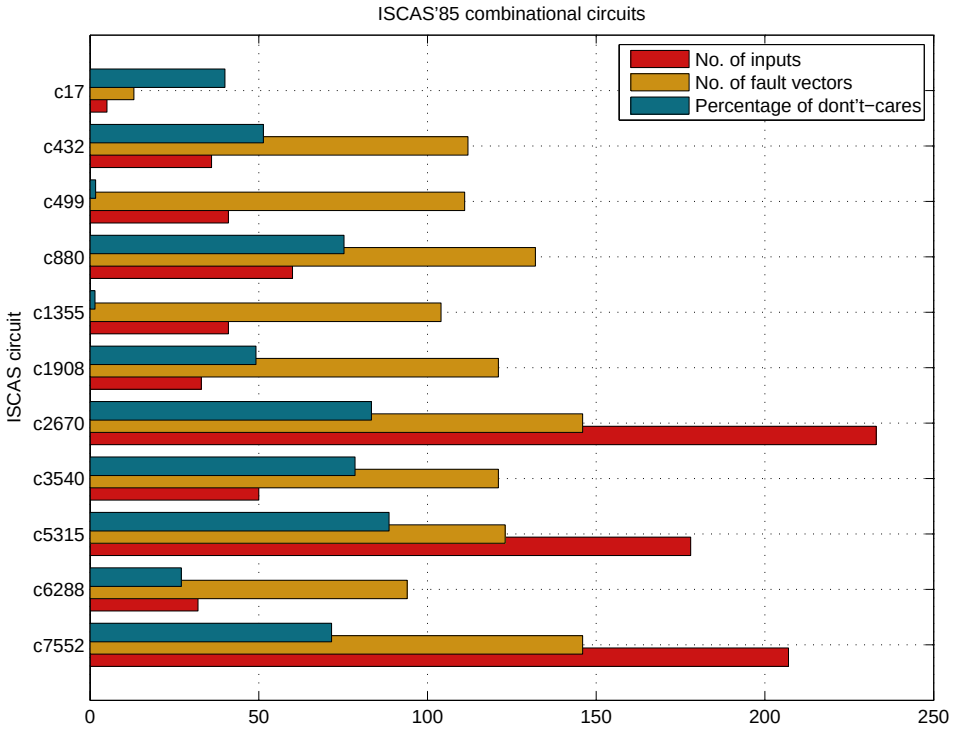


Figure 6.4: Problem instances for the Cellular Automata Problem. Included are the combinational circuits from the ISCAS'85 benchmark set. In the graph, the number of circuit inputs is depicted, along with the number of fault vectors generated and the number of don't-care symbols in those fault vectors.

6.2.6 Previous work

The Parallel Selfish Gene Algorithm is not the first evolutionary algorithm solving the problem of evolving cellular automata for BIST design. In [6], Corno et al. used the Selfish Gene Algorithm to solve this problem, although their approach had some subtle differences from what was adopted for PSG. In their approach, they recognized the fact that certain combinations of CA rules **Rule 90** and **Rule 150** produce cellular automata with a maximal period, that is, one that traverses all possible $(2^i - 1)$ non-zero states before returning to its initial non-zero state [35]. This type of CA is known as the *90/150 LHCA*. The researchers "pre-loaded" the algorithm with the 90/150 LHCA and started the search in the neighborhood of the 90/150 LHCA. In contrast, the implementation of the Cellular Automata Problem in the PSG starts the search "from scratch," without any prior knowledge.

Another difference is the fact that Corno et al. allowed all 256 CA rules to be used; the final difference is their use of fault-simulation for the fitness function, compared to the fault vector covering method used in PSG.

In their paper, the researchers showed that they were able to find cellular automata that detected more faults than the 90/150 LHCA, reaching full 100% fault coverage for many circuits in the ISCAS'85 set.

7 Experiments

In this chapter, the performance and behavior of PSG will be thoroughly analyzed and described. In the first section, the process of tuning the algorithm's parameters is discussed. With all but one of the parameters fixed at their optimal values, we show some general performance characteristics in the second section before moving on to the third section, where scalability and speedup are evaluated, the two most important characteristics of any parallel system. In the fourth section we compare the performance of PSG to that of a random search algorithm. Finally, the last section rounds off the chapter by evaluating the *real* fault coverage obtained when cellular automata evolved by PSG are used as test pattern generators for BIST circuitry.

All experiments in this chapter were performed on the "star" computing cluster installed at the Department of Computer Science and Engineering of the Faculty of Electrical Engineering, Czech Technical University. The cluster gets its name from the topology of its interconnection network: the network uses a central switch and therefore resembles a star. The system falls into the "cluster of workstations" category of parallel systems: a collection of PC-based machines connected by a local network. Detailed parameters of the cluster are given in Table 7.1.

Number of machines	8
Number of processors per machine	4 (two dual-core CPU's)
Total number of CPU's	32
CPU type	Opteron 2218 (AMD K8 architecture)
Memory per machine	4 GiB
Networking technology	InfiniBand, 10 Gb s^{-1}
Operating system	Linux kernel 2.6.16, x86-64 ISA
Communication library	OpenMPI version 1.2.5

Table 7.1: Parameters of the "star" computing cluster.

7.1 Parameter tuning

Any evolutionary algorithm stands or falls with the choice of its parameters and selecting or *tuning* the parameters is often the most time-consuming part of an algorithm design. In section 4.2 we saw that there are some methods by which one can reduce the number of parameters to be tuned, but the fact remains that there will always be some parameters that have to be tuned; there is no such thing as a "tuneless" algorithm.

In this section we will describe the process of parameter tuning for PSG. The following parameters were addressed:

1. ϵ, the probability reward and penalty given during tournaments.
2. `convergenceThreshold`, the probability level at which convergence is determined for any allele.
3. X_0, the number of alleles sent in the first migration event.
4. Y_0, the number of alleles that must be converged in order to trigger the first migration event.
5. μ, the migrant mixing coefficient.
6. γ, the epoch length modifier.
7. `epochGenerationLimit`, the number of generations after which a migration event is triggered regardless of the number of converged alleles.
8. `intervalCheckEpoch`, `intervalStoppingLocal` and `intervalStoppingGlobal`: the number of generations between consecutive checks of, respectively, the migration event conditions, the local stopping conditions and the global stopping conditions.

Notably absent from this list is δ, the degree of the communication network's graph. Due to its importance, this parameter will be examined in-depth, along with p, in section 7.3. During the parameter tuning experiments, the degree was set to the value $\delta = \frac{p}{2}$, which is often found to lead to optimal results [12, 13].

In the interest of keeping the number of experiments within reasonable limits, only three problem instances were used for the parameter tuning phase, all of them from the set of ISCAS'85 benchmark circuits given in Table 6.1 in the previous chapter.

1. `c6288`
2. `c3540`
3. `c5315`

Parameter tuning was divided into two iterations; during the first iteration, related parameters were chosen two at a time. For each parameter, three to five different values were chosen. The set of experiment configurations was taken as the Cartesian product between the two sets of parameter values. The experiments were then executed for 20 minutes, once using $p = 4$, once using $p = 8$.

For each problem instance, the set of best fitness values found during the 20-minute run was rearranged by subtraction and multiplication to give a mean value of 0 and a variance of 1. The values were then plotted as four contour graphs: one graph for each problem instance, and one for the sum of the three sets of values. Figure 7.1 shows the color scale used in these contour graphs.

When the results were inconclusive, another grid of parameter values was picked intersecting the first, and the results were combined.

In the second iteration, parameter values were selected close to the local optima discovered during the first iteration. The process of running the experiments and processing the results was otherwise identical to that of the first iteration.

Figure 7.1: Color scale for the contour graphs that show results of parameter tuning. The numbers on the scale represent the distance of a data point from the mean, in standard deviations.

7.1.1 Iteration 1

Figure 7.2 shows the results of tuning ϵ and `convergenceThreshold`. It can be clearly seen that low values of ϵ cause low fitness values. For `convergenceThreshold` there is also a local optimum, although it is not as distinct as with ϵ.

In Figure 7.3 the results of tuning Y_0 and X_0 are shown. A local optimum for both variables can easily be discerned.

Figure 7.4 shows the results of tuning μ and γ. Unfortunately, a single local optimum does not exist here; a diagonal valley of low fitness values cuts across the graph. The larger local optimum, around $\mu = 0.8$ and $\gamma = 7$, was selected as the area for the second iteration.

The last graph, Figure 7.5, shows the results of tuning `epochGenerationLimit` and `intervalCheckEpoch`. From the graph it is evident that these parameters have a very subtle and rather unpredictable effect on the performance of the algorithm. Nevertheless, a small local optimum in the top-right corner of the graph was selected as the area for the second iteration.

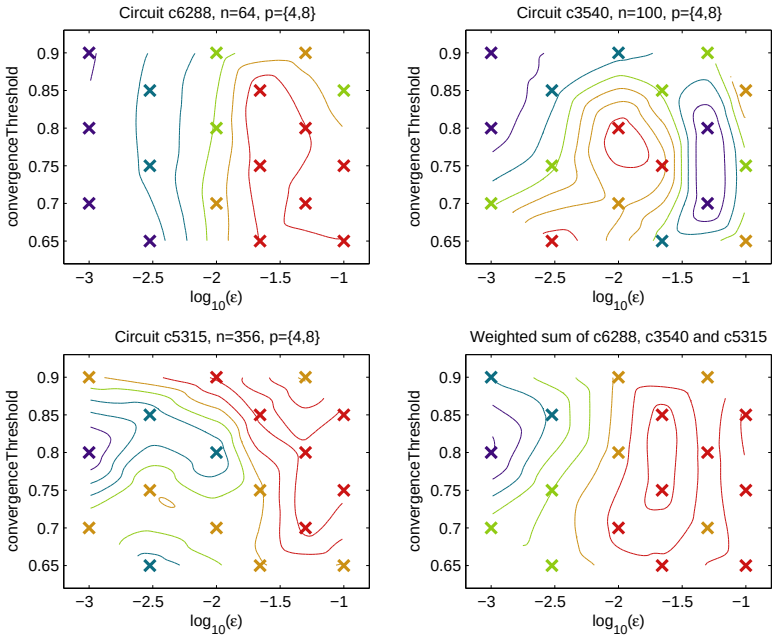


Figure 7.2: Parameter tuning of the parameters ϵ and `convergenceThreshold`. Values of ϵ were chosen with geometric spacing in the range suggested by Corno et al. [4] and are plotted logarithmically. Values of `convergenceThreshold` were chosen with linear spacing in the range suggested by Corno et al.

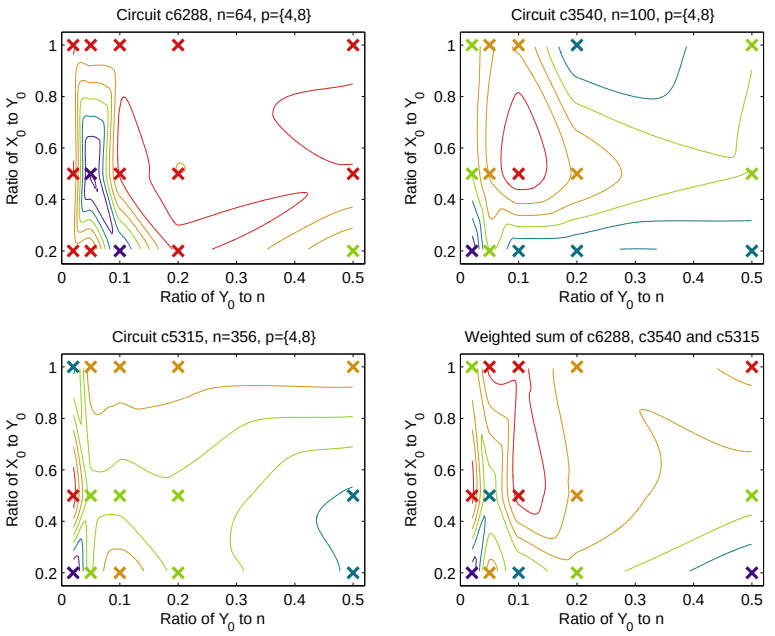


Figure 7.3: Parameter tuning of the parameters Y_0 and X_0. Since there is a strong dependency between these parameters and the number of genes n, the X-axis shows the ratio Y_0/n; on the Y-axis the ratio X_0/Y_0 is shown. Values of the two ratios were chosen with geometric spacing.

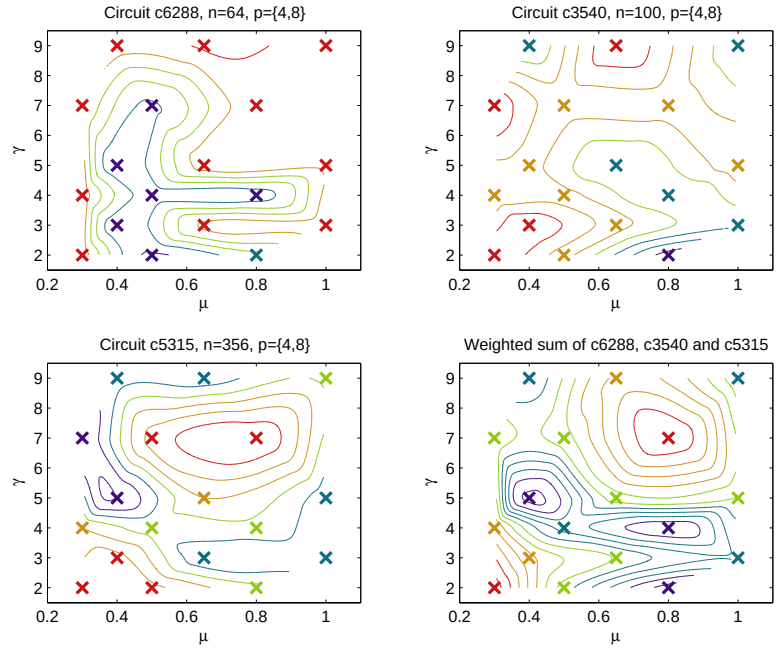


Figure 7.4: Parameter tuning of the parameters μ and γ. Values of μ were chosen with semi-linear spacing in the range from 0 to 1. Values of γ were chosen with geometric spacing, with the added constraint that they had to be integers.

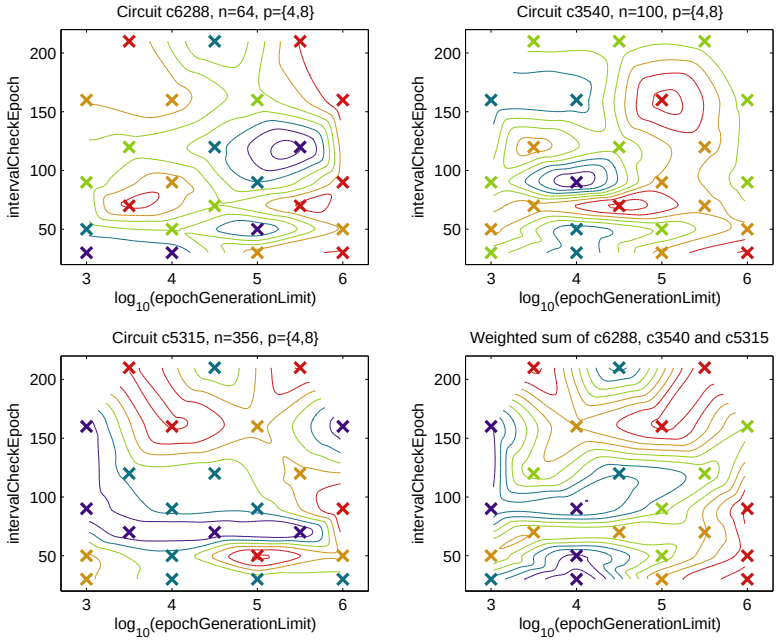


Figure 7.5: Parameter tuning of the parameters `epochGenerationLimit` and `intervalCheckEpoch`. Values of `epochGenerationLimit` were chosen with geometric spacing and are plotted logarithmically. Values of `intervalCheckEpoch` were also chosen with geometric spacing, but are plotted linearly. For each different value of `intervalCheckEpoch` the values of `intervalStoppingLocal` and `intervalStoppingGlobal` were also adjusted, but these are not shown on the graph.

7.1.2 Iteration 2

In the second iteration of parameter tuning, only five different combinations of parameter values were chosen, arranged like the *pips* on the "5" face of ordinary playing dice, with the center pip representing the values chosen during iteration 1. Results from the second iteration of parameter tuning are shown in figures 7.6, 7.7, 7.8 and 7.9.

The purpose of iteration 2 was primarily confirmation that the values from iteration 1 were really correct optima; the secondary goal was to perform tests in the area of the local optima at a resolution higher than in iteration 1. The results are straightforward: the five data points are shown along with a contour graph fitting these points. On the fourth quadrant of each figure, a black asterisk represents the final values of the parameters that were chosen. These values were then used in all further experiments with few exceptions. For reference, the exact values are given in Table 7.2.

Parameter	**Value**
ϵ	0.026
`convergenceThreshold`	0.80
Y_0/n	0.107
X_0/Y_0	0.590
μ	0.71
γ	6
`epochGenerationLimit`	100 000
`intervalUpdateParameters`	10
`intervalCheckEpoch`	170
`intervalStoppingLocal`	1200
`intervalStoppingGlobal`	1400

Table 7.2: Final parameter values after iteration 2 of parameter tuning

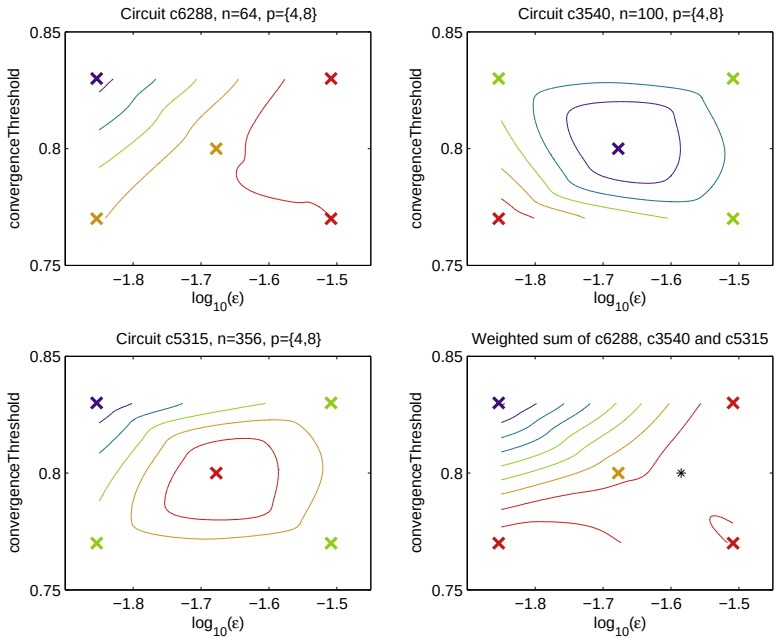


Figure 7.6: Parameter tuning of the parameters ϵ and `convergenceThreshold`. The black asterisk denotes the final parameter values chosen for further experiments.

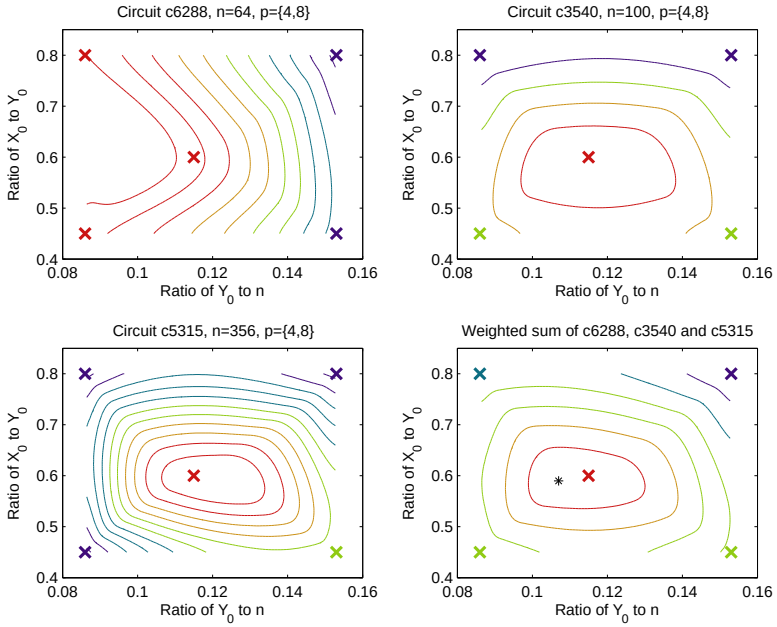


Figure 7.7: Parameter tuning of the parameters Y_0 and X_0. The black asterisk denotes the final parameter values chosen for further experiments.

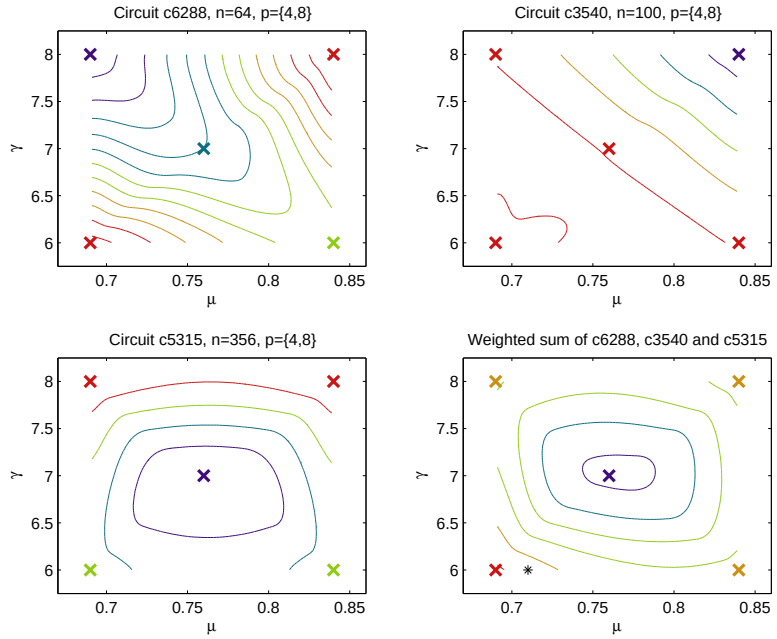


Figure 7.8: Parameter tuning of the parameters μ and γ. The black asterisk denotes the final parameter values chosen for further experiments.

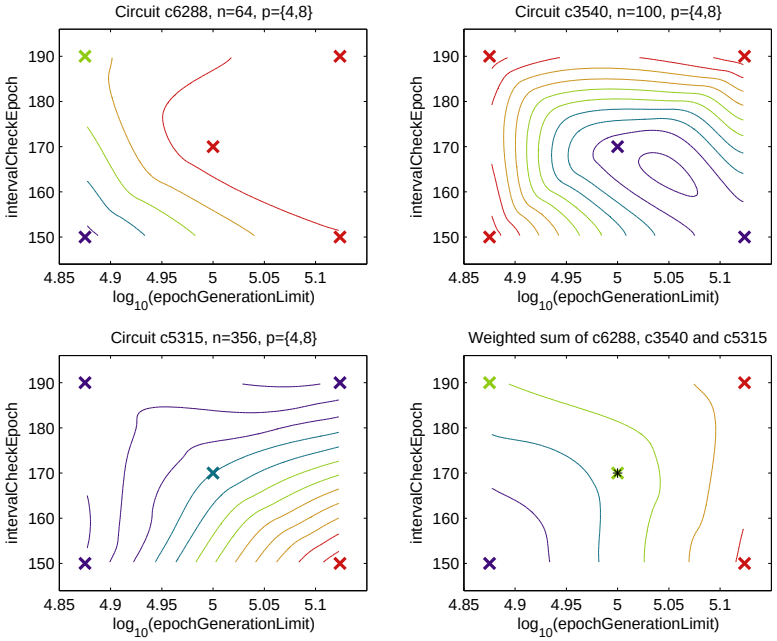


Figure 7.9: Parameter tuning of the parameters `epochGenerationLimit` and `intervalCheckEpoch`. The black asterisk denotes the final parameter values chosen for further experiments.

7.2 General performance metrics

Before moving on to the scalability experiments we will briefly demonstrate some general metrics of PSG's performance using the parameters found in the previous section. Section 7.2.1 shows what kinds of messages are exchanged between processors during a typical run of the algorithm, and in what proportions. In section 7.2.2 we illustrate the deterministic self-adaption strategy introduced in section 4.2.3 by showing how the value of ρ changes over time.

7.2.1 Communication

Figure 7.10 and Figure 7.11 show the communication characteristics for two runs of the algorithm. In these runs, the number of processors was set to 4 and the degree was set to 2. The problem instances were `c3540` and `c6288`, respectively. The bar graph in the bottom half of each figure shows the numbers of messages sent and received for each distinct message type used by the algorithm. These message types are:

`ALLELES`: The most important message of all, this is the message containing converged allele values, which is sent in a migration event. For each message sent, δ messages are received.

`LOCAL_STOP`: When a processor reaches its local stopping conditions, a `LOCAL_STOP` message is sent to p_0.

`GLOBAL_STOP`: When p_0 reaches the global stopping conditions, this message is sent to all other processors.

`LOCAL_SOLUTION`: Upon receiving `GLOBAL_STOP`, processors send their best individual found to p_0 inside a `LOCAL_SOLUTION` message.

`TERMINATION_TOKEN`: When p_0 has received $(p-1)$ `LOCAL_SOLUTION` messages, it sends out the `TERMINATION_TOKEN` to all other processors, informing them it is safe to disconnect the message loop and terminate.

The algorithm states that only `ALLELES` and `LOCAL_STOP` may be sent multiple times, all other messages are sent only once. In Figure 7.10, no processors reach their local stopping conditions and thus no `LOCAL_STOP` messages are sent. Figure 7.11 shows an interesting case where one processor, p_1, is lucky to reach its local stopping conditions early in the execution, and as a result it repeatedly sends `LOCAL_STOP` messages to p_0.

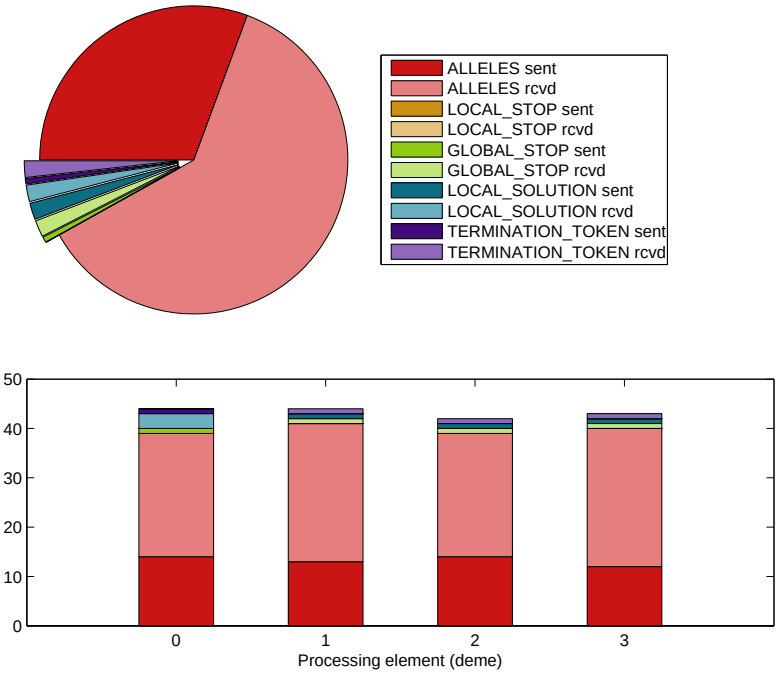


Figure 7.10: Visualization of the amounts of messages of distinct types exchanged in the system (with $p = 4$). The tested problem file was `c3540` ($n = 100$). The pie chart shows the aggregate amounts for all processors combined, the bar chart shows the individual amounts for each processor.

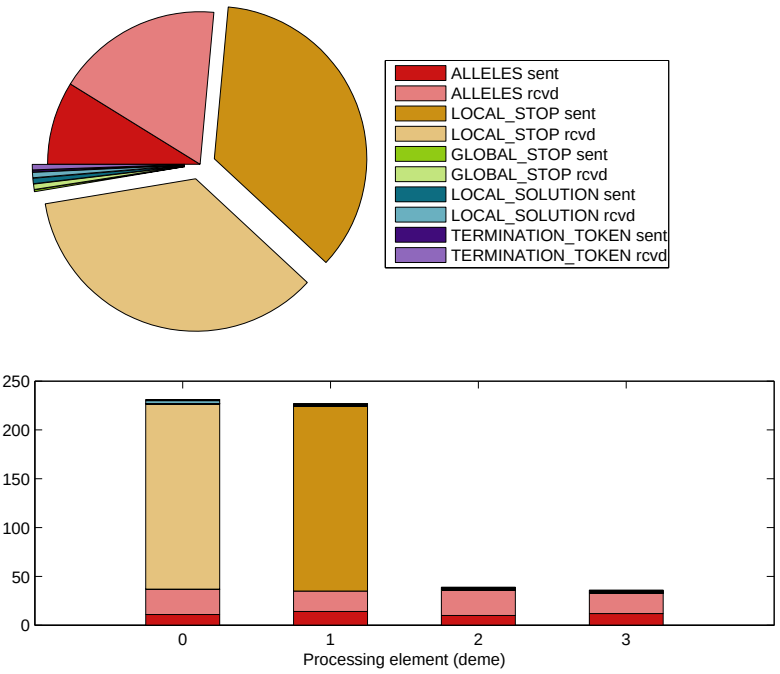


Figure 7.11: Visualization of the amounts of messages of distinct types exchanged in the system (with $p = 4$). The tested problem file was `c6288` ($n = 64$). The pie chart shows the aggregate amounts for all processors combined, the bar chart shows the individual amounts for each processor.

7.2.2 Self-adaption

In section 4.2 we describe a strategy for deterministic self-adaption of the mutation probability ρ, given by equation (4.6):

$$\rho(n, c, t) = \left(2 + \frac{n-2}{n}c + \frac{\sqrt{t}}{n}\right)^{-1}$$

Considering that n remains constant during execution of the algorithm, any changes in ρ are related to changes in c, the number of converged alleles, and t, the number of generations in the current epoch, which in turn depends on e, the number of epochs.

Figure 7.12 and Figure 7.13 plot the value of ρ along with c and e, where c is expressed as the percentage of genes that have converged to a specific allele. The figures refer to the same runs as described in Figure 7.10 and Figure 7.11 in the previous section.

In Figure 7.12, we see that c does not always increase over time; it can happen that some demes are forced to abandon their local optima in favor of more promising areas found in other demes. This is a normal behavior for an island model PEA. When this happens, the decreasing value of c leads to higher values of ρ, as can be seen from the graph.

Figure 7.13 shows p_1 reaching its local stopping conditions early in the execution when c approaches 100%. The graph also shows the wide diversity that can occur between demes, with p_0 and p_2 never converging more than 50% of their genes while the other demes appear to be more successful.

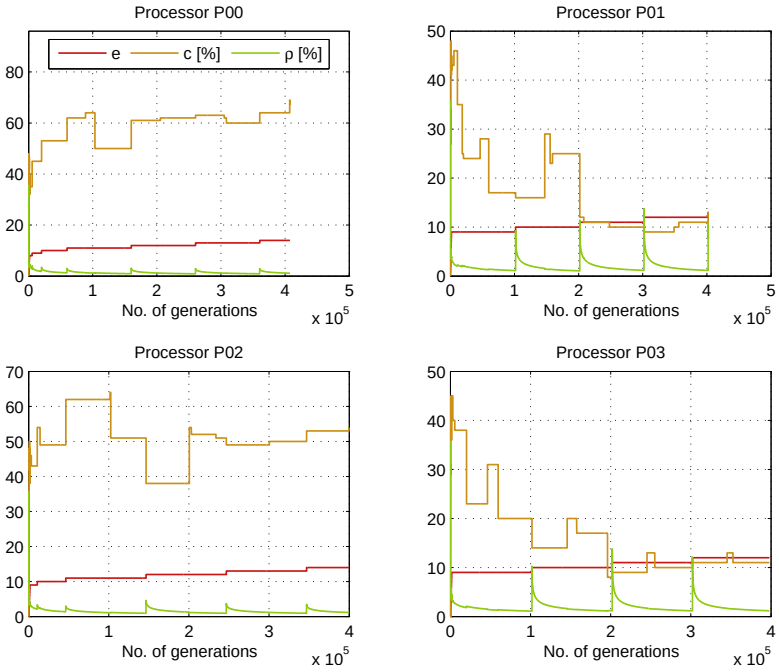


Figure 7.12: Visualization of the number of epochs e, the number of converged alleles c, and the mutation probability ρ for each processor in the system (with $p = 4$). The tested problem file was `c3540` ($n = 100$).

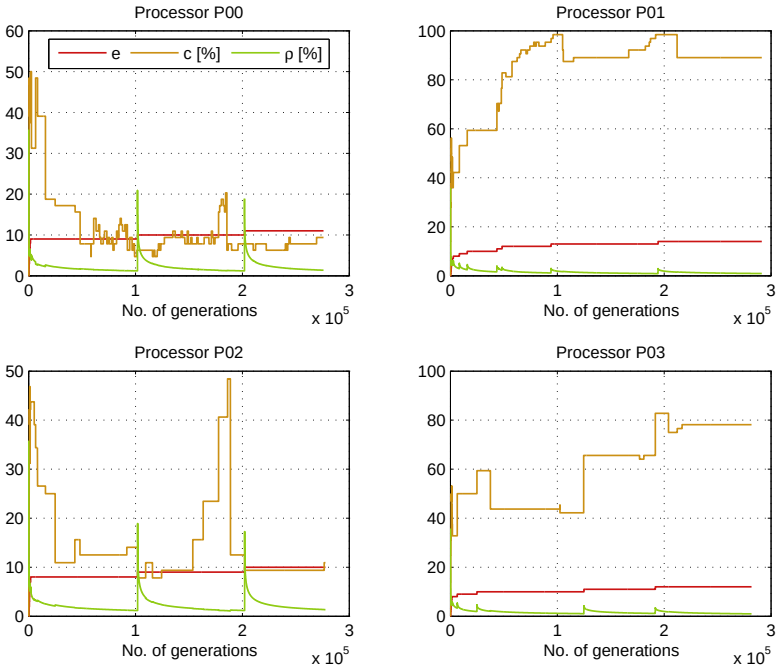


Figure 7.13: Visualization of the number of epochs e, the number of converged alleles c, and the mutation probability ρ for each processor in the system (with $p = 4$). The tested problem file was `c6288` ($n = 64$).

7.3 Scalability and speedup

In this section we will examine the parallel speedup of the algorithm in order to evaluate its scalability. In section 3.2.1 we highlighted some of the issues in measuring speedup for heuristic methods such as evolutionary algorithms, particularly the observation that when adopting a convergence-based approach to speedup, some PEA's might exhibit very poor speedups, with the convergence times sometimes even increasing as more PE's are added. However, such a PEA does make use of the extra resources, reflected by a higher solution quality. For such PEA's, we advocated the use of a solution-quality-based approach to speedup, measuring the time it takes different algorithm configurations to reach solutions of a certain quality.

Preliminary scalability testing revealed that PSG falls into this category; convergence-based speedup was virtually nonexistent, yet a correlation seemed to exist between the number of processors and the highest fitness found during execution. Therefore, in order to provide a more meaningful evaluation of PSG's scalability, a solution-quality-based approach to speedup was adopted.

Due to PSG's lack of an explicit population, a somewhat naïve stopping condition was used: execution terminates after the first individual with a fitness exceeding a certain threshold has been constructed. This is not fundamentally different from the approach in explicit-population PEA's, where execution terminates when the population's average fitness exceeds a threshold: at any generation, an individual in PSG is largely constructed from high-frequency alleles and can thus be seen as an "average" individual with, we assume, an average fitness.[1]

In section 3.1 we introduced equation (3.1), the general formula for parallel speedup.

$$S(n,p) = \frac{SU(n)}{T(n,p)}$$

In other words, parallel speedup is a ratio of two values; taking this into account, the scalability experiments are split into two phases: in phase 1 we will find the numerator $SU(n)$, and in phase 2 we will find the denominator $T(n,p)$.

7.3.1 Phase 1

When using a solution-quality-based approach to speedup, the first task is to set the fitness target to such a level that a sequential implementation can reach it in a reasonable amount of time. To determine these levels, a sequential algorithm was executed on each problem instance for 30 minutes, or until the maximal fitness value was found. Figure 7.14 shows the highest fitness values found during each run, along with the highest level of convergence reached. The fitness targets for each problem instance were then set slightly below these highest fitness values, to prevent rounding errors from unnecessarily prolonging execution.[2] Problem instances where no alleles converged at all (`c499` and `c1355`) were excluded from the rest of the experiments.

In order to find $SU(n)$ for each problem instance, the sequential algorithm was then executed once more, this time with the fitness target set as a local stopping condition. Taking the stochastic nature of the algorithm into account, each problem instance was run three times. The average wall-clock runtime of these three runs was taken as $SU(n)$. The results of these runs are displayed graphically in Figure 7.15.

[1] In real-world populations, the average individual does not always possess an average fitness; in many societies, a small minority of individuals holds a disproportionate amount of resources (food, breeding rights, money).

[2] In the implementation, fitness values use a floating-point representation.

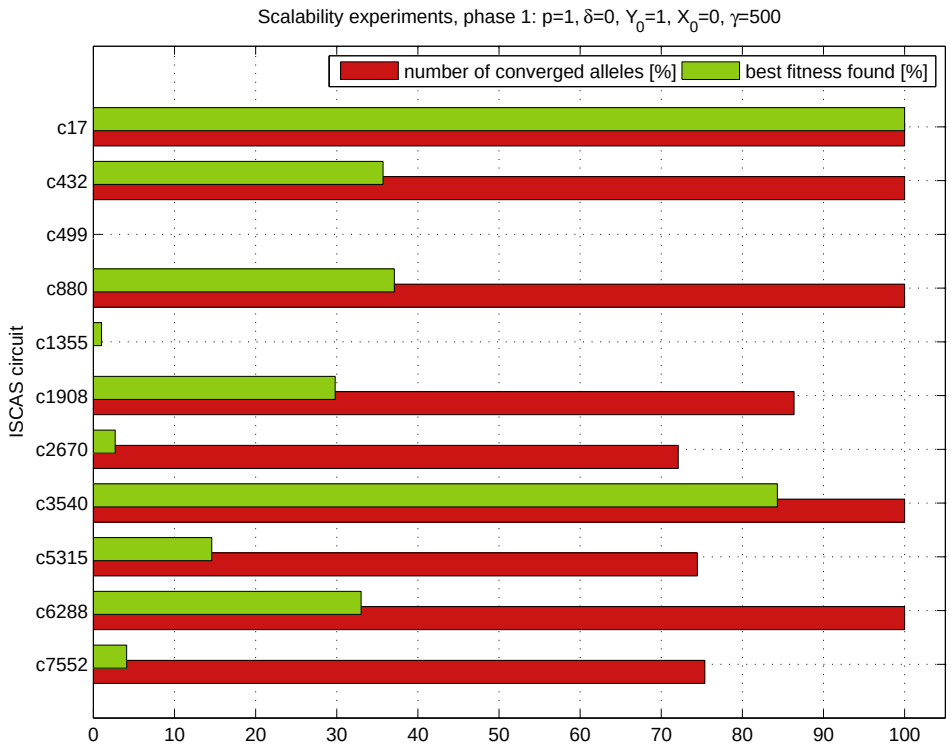


Figure 7.14: The highest fitness values, and number of converged alleles found during a 30-minute run of the sequential algorithm.

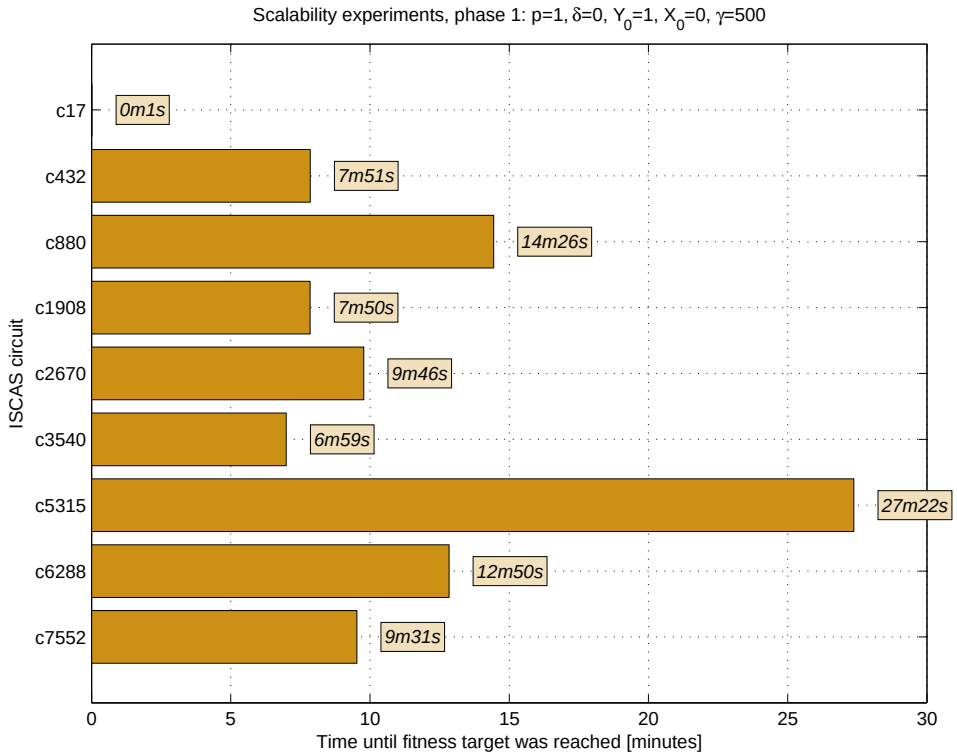


Figure 7.15: The average time $\mathrm{SU}(n)$ taken by a sequential configuration of PSG to reach the fitness targets shown in Figure 7.14.

7.3.2 Phase 2

In phase 2, the runtimes of the parallel implementation were measured. Letting f_T denote the fitness target found in phase 1, the algorithm's stopping conditions were set as follows:

- Each deme had a local stopping condition of the form $f \geq f_T$; in other words, the deme starts sending out `LOCAL_STOP` messages once an individual with a fitness exceeding the target is found.
- Processor p_0, responsible for checking the global stopping conditions, was set to initiate termination when at least one deme reaches its local stopping conditions. In other words, p_0 starts sending out `GLOBAL_STOP` messages after the first `LOCAL_STOP` message has been received. If no such message is received after some time limit (30 minutes) has elapsed, termination is initiated regardless.

In section 7.1 we left one parameter unaddressed: δ, the degree of the interconnection network. The goal of the scalability experiments was to analyze the correlation between speedup $S(n, p)$ and δ. A range of values for δ similar to the one used by Cao, et al. [13] was chosen. Specifically, four degrees of connectivity were examined:

1. $\delta = \frac{p}{2}$, the value designated as "usually optimal" by Cantú-Paz [12].
2. $\delta = (p - 1)$, or full connectivity; each deme sends migration packets to all others.
3. $\delta = 1$, the least connectivity; each deme sends a migratation packet to only one neighbor.
4. $\delta = 2$. Although differing only by one, this scheme is rather different from $\delta = 1$; when $\delta = 1$, any deme p_i sends migration packets to p_j, where $j = (i+1) \bmod p$. This means that p_j does not send migration packets to p_i. In the case of $\delta = 2$, two neighboring demes do send migration packets to each other: any deme p_i sends migration packets to p_j and to p_k, where $j = (i + 1) \bmod p$ and $k = (i - 1) \bmod p$.

The algorithm was executed on each of the nine ISCAS circuits from the first phase, using each of these four δ schemes. The number of processors p ranged from 2 to 24; other runtime parameters were set to the values found during the parameter tuning experiments described in section 7.1.

Wall-clock execution times were collected for each run. For each problem instance, these execution times were plotted as four series in the (p, T)-plane, one for each δ scheme. Dividing $SU(n)$, obtained in phase 1, by the values $T(n, p)$, speedup $S(n, p)$ was obtained. Speedups were again plotted as four series, this time with S on the y-axis.

Figures showing $T(n, p)$ and $S(n, p)$ for each tested problem instance are reproduced in Appendix A. Of special interest are the figures for `c17` shown in Figure A.1: due to the simplicity of this circuit, execution never took more than two seconds. Since time measurement in the implementation is done with a resolution of 1 second, $T(n, p)$ could not be measured accurately, and the values of $S(n, p)$ are not very meaningful. The graphs are reproduced solely for the sake of completeness.

A general notion of scalability for PSG may be obtained by producing a graph of aggregate speedup: the average values of $S(n, p)$ for each distinct value of p. In computing these values, we excluded the results of `c17` and so aggregate speedup is the average speedup across eight problem instances, with values of n ranging from 64 (`c6288`) to 466 (`c2670`). Aggregate speedup is shown in Figure 7.16.

The figure clearly shows that as more processors are added to the system, the algorithm finds a suitable individual in a reduced time. While speedup is not linear (in reality, it rarely is), it certainly is an increasing function for all δ schemes considered.

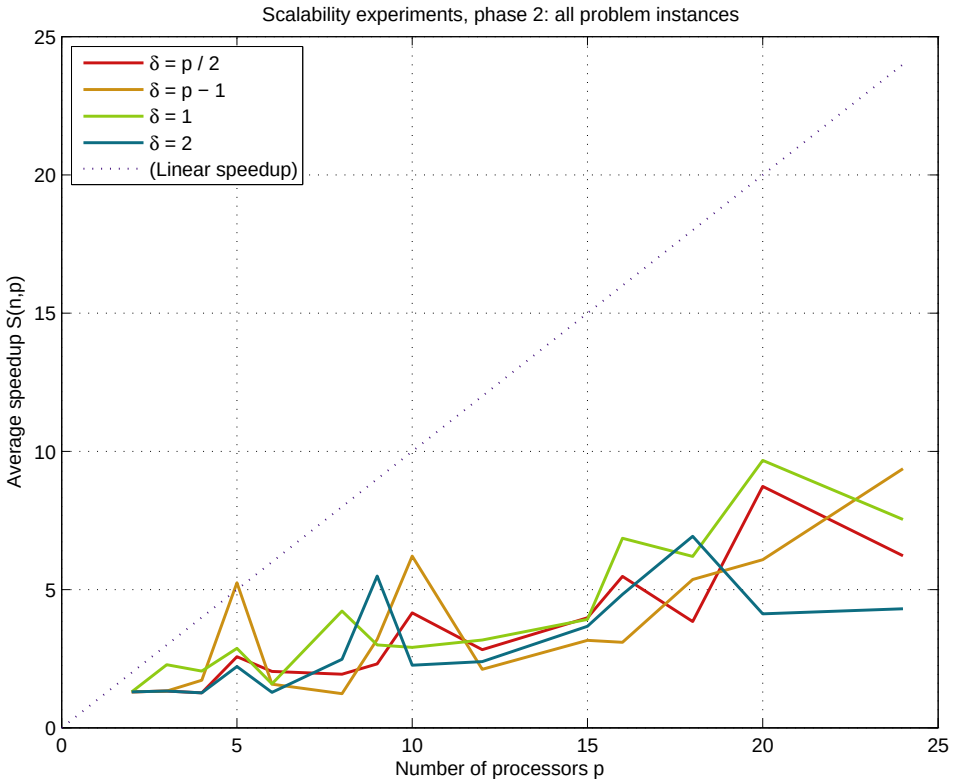


Figure 7.16: Speedup $S(n, p)$ of PSG for different combinations of p and δ for all ISCAS circuits tested. The dotted line denotes the theoretical linear speedup $S(n, p) = p$.

An interesting observation is the fact that optimal performance is often obtained when $\delta = 1$. Furthermore, the significant difference between the performance of the $\delta = 1$ and $\delta = 2$ schemes leads to the conclusion that in PSG, the topology of the network appears to play a larger role than in canonical genetic algorithms. This conclusion is based on the fact that while the $\delta = 1$ and $\delta = 2$ schemes do not differ greatly in their degree, they do differ in topology: when $\delta = 2$, a converged allele emigrating from any deme p_i may migrate back to p_i after just one epoch, whereas in the case of $\delta = 1$ the allele must go through at least $(p - 1)$ epochs before it finds its way back to p_i.

7.4 Comparison with random search

In this section we will examine how the algorithm performs when compared to a random search algorithm. Using terminology from evolutionary algorithms, a random search algorithm can be defined as an algorithm that, each generation, evaluates the fitness of a random individual. When execution terminates, the best individual found is returned.

PSG is a very flexible algorithm, and can be turned into a random search algorithm simply by setting ϵ to zero. This prevents the algorithm from adjusting the population's gene pool, leaving the VP in its initial state (all alleles are equally likely) for the length of the algorithm's execution. In section 2.2 we learned that when the Selfish Gene Algorithm constructs an individual, the individual's genotype will be made up of a combination of random and proportionate alleles. When the VP is in its initial state, there is no difference between a random and a proportionate allele, and all alleles are effectively random.

To make the comparison between PSG and random search meaningful, it was decided to let each algorithm execute for a fixed number of generations. For each problem instance, the generation target g_T was calculated as follows:

1. Execution reports from the scalability experiment were analyzed. The total number of generations executed was summed up and divided by the total number of seconds of wall-clock time. This gives the average number of generations per second $\bar{\nu}_g$.
2. g_T was set to $32 \cdot \bar{\nu}_g$, rounded to the nearest integer.
3. `intervalStoppingLocal` was set to $8 \cdot \bar{\nu}_g$, rounded upwards to the nearest multiple of `intervalUpdateParameters`.
4. `intervalStoppingGlobal` was set to $9 \cdot \bar{\nu}_g$, rounded upwards to the nearest multiple of `intervalUpdateParameters`.

By using this method, execution was expected to take around 32 seconds, with local and global stopping conditions being checked roughly every 8 and 9 seconds, respectively.

Further parameters of the experiment were as follows:

- The number of processors $p = 8$.
- For PSG, $\delta = \{1, 2, 4, 7\}$.[3] For random search δ is not relevant.
- For random search, $\epsilon = 0$.
- Each deme received a local stopping condition of the form $g \geq g_T$.
- p_0 received a global stopping condition of the form $\Sigma g \geq p \cdot g_T$.

Eight problem instances were thus tested: the nine from the scalability experiments minus `c17`, since all configurations are capable of reaching 100% fitness for `c17` within a few seconds. For each problem instance, one run of random search was performed, and four runs of PSG, each with a different value of δ. The highest fitness values reached during execution were collected and are represented in Figure 7.17 as ratios of solution quality.

[3] This corresponds to the four δ schemes discussed in the previous section.

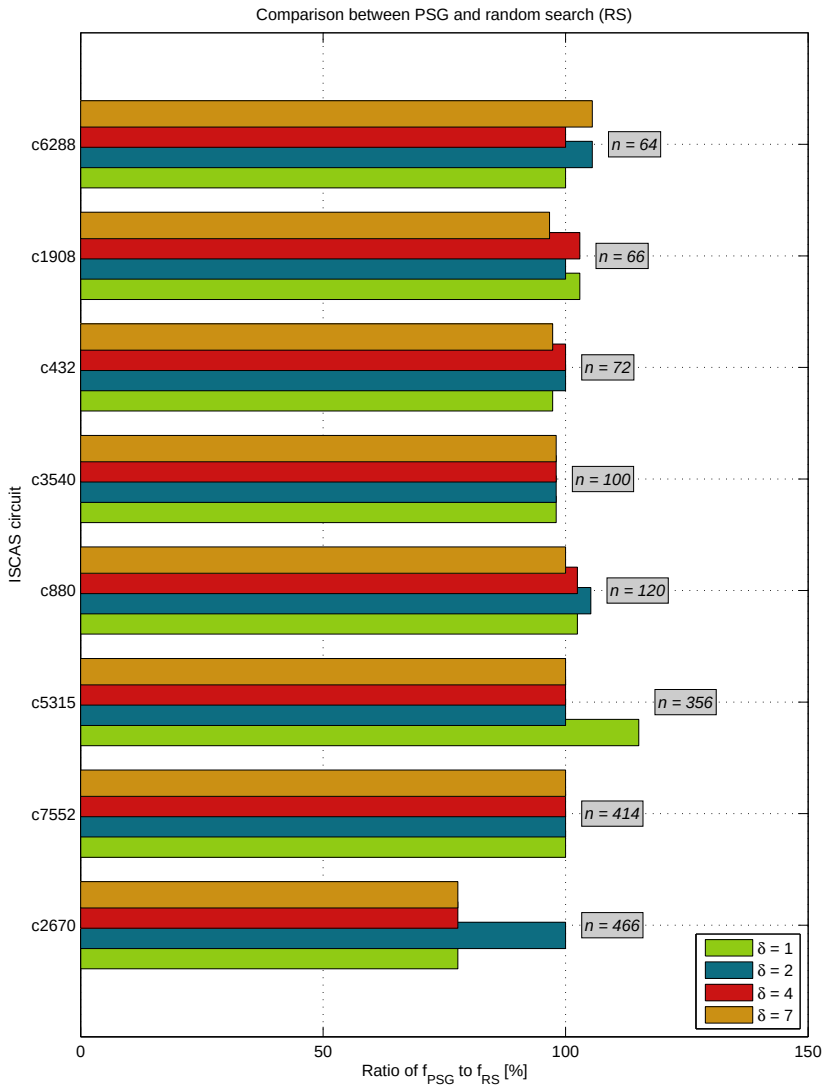


Figure 7.17: Comparison of solution qualities of PSG and random search (RS). The ratio f_{PSG}/f_{RS} is shown, where f_{PSG} and f_{RS} are the best fitness values found by PSG and random search, respectively. Additionally, the number of inputs n is shown for each problem instance.

The results are not as impressive as one would have hoped; PSG rarely outperforms random search, and if it does, it does so by only a few percent. However, it should be noted that for the problem instances considered, the time available for execution in this experiment (less than one minute) is far less than the time needed for PSG to converge and yield its greatest benefit. Moreover, Corno et al. already demonstrated that the Selfish Gene Algorithm behaves as a random search algorithm for its first phase of execution [4].

Summing up, the experiments in this section neither confirm nor deny the hypothesis that PSG can outperform a random search algorithm. For that, further experiments would be needed, where there is more time available for execution. Unfortunately, organizational limits imposed on the computing cluster prevented such experiments from being performed.

7.5 Real fault coverage

In this section, we will examine the *real* fault coverage of the cellular automata that PSG evolves. In section 6.2.1 we described two methods of evaluating a TPG: full fault-simulation and fault vector covering. The latter method, which is the one used by PSG, is not as accurate as the former, since no fault simulation is performed. In this section, we will perform this fault simulation, using the cellular automata evolved by PSG as TPG's.

To determine the real fault coverage on the benchmark circuits, the first task is to find the cellular automata that are most likely to achieve the highest fault coverage. To find them, the results of the scalability experiments were analyzed; for each circuit, the "winning" cellular automaton was chosen as the individual yielding the highest fitness value. If multiple individuals all had the highest fitness value, the winner was chosen as the individual produced by the algorithm that achieved the highest level of convergence. For each circuit, the fitness value of the fittest individual is shown in Figure 7.18, along with the configuration of the algorithm that evolved this individual. At this point, it is worth repeating the definition of fitness in the Cellular Automata Problem: the fitness value of an individual refers to the percentage of fault vectors for the given circuit that are covered by test vectors from the cellular automaton.

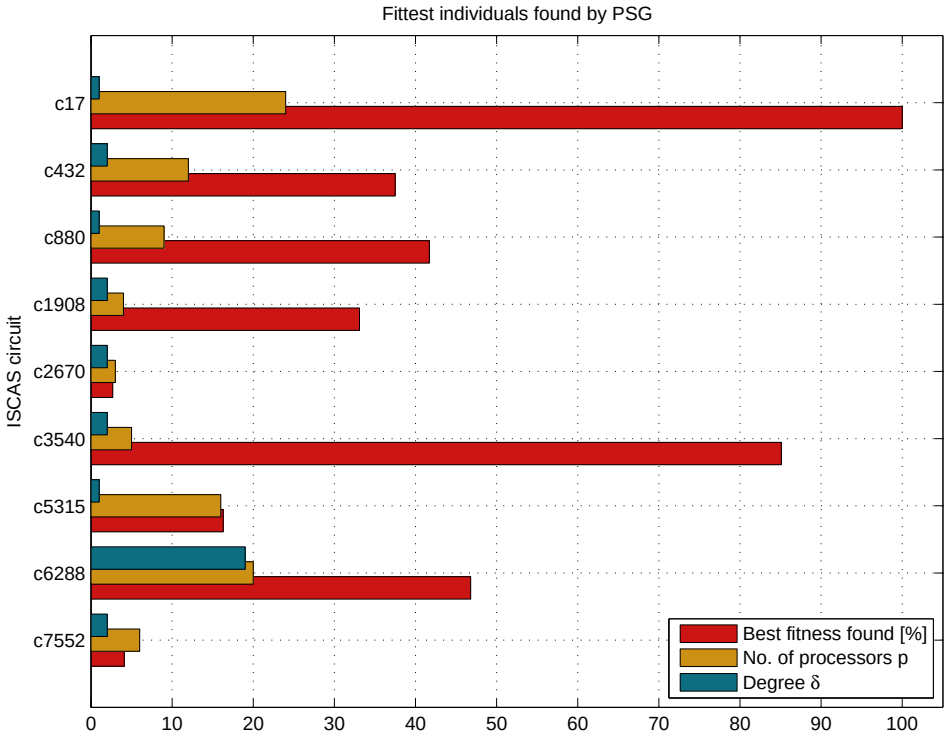


Figure 7.18: Best fitness values found by PSG during scalability testing, along with the configuration that found the fittest individual.

Fault coverage was obtained by fault-simulating each circuit, using I vectors extracted from the winning CA as the set of input vectors V_T, $|V_T| = I$. Due to computational constraints, the highest number of input vectors considered was one million ($I_{max} = 10^6$).

Denoting ΣF as the total number of faults introduced by the fault simulator, and F_{UD} as the lowest number of undetected faults, fault coverage C_F was calculated according to the following formula.

$$C_F = \frac{\Sigma F - F_{UD}}{\Sigma F} \tag{7.1}$$

In addition to fault coverage, the minimal number of input vectors I_C needed for achieving maximal fault coverage was also determined. I_C is minimal in the sense that when the I_C'th element from V_T is removed, more faults will go undetected.

Results from the fault coverage experiment are given in Table 7.3. Figure 7.19 shows this information graphically. Fault coverage is very high for all circuits, with two circuits, `c17` and `c880`, reaching 100% fault coverage. For these circuits, we may compare the values of I_C to the range of values presented in [29].[4] For `c17`, I_C is expected to fall within the range 2–33, with a statistical average of 4. Our value (12) is above the average, but still well within the range. For `c880`, the range is 2500–75 000, with an average of 13 000. Our value of 8868 is well below the average number of input vectors needed to fully test the circuit.

Circuit name	**Undetected faults** F_{UD}	**Fault coverage** C_F	**Minimal number of vectors** I_C
`c17`	0	100%	12
`c432`	4	99.2%	826
`c880`	0	100%	8868
`c1908`	9	99.5%	3611
`c2670`	137	93.9%	946 695
`c3540`	137	96.0%	20 482
`c5315`	59	98.9%	2831
`c6288`	50	99.4%	70
`c7552`	214	97.2%	912979

Table 7.3: Real fault coverage obtained by cellular automata evolved by PSG.

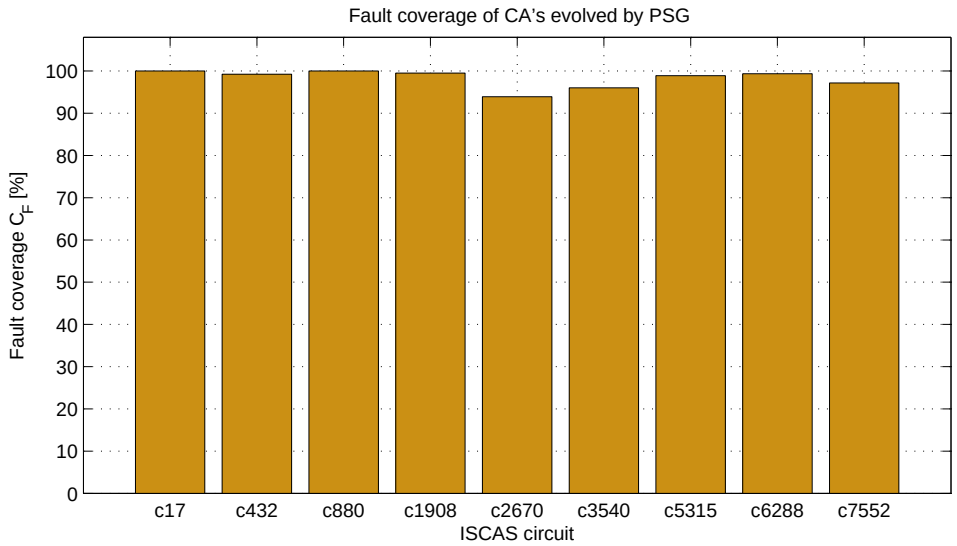


Figure 7.19: Real fault coverage obtained by fault-simulating the ISCAS benchmark circuits, using cellular automata evolved by PSG as the TPG.

[4] It should be noted that these ranges were established using an LFSR as the test pattern generator.

8 Conclusions

In this chapter, we will review the development and analysis of the Parallel Selfish Gene Algorithm, presenting the knowledge gained, the lessons learned, and the questions that remain unanswered. First, some general conclusions will be given, followed by the definitive verdict on scalability and efficiency in the second section. The final section presents a critical view of the results of the work, and states some open questions and suggestions for further work.

8.1 General conclusions

In section 2.2 the Selfish Gene Algorithm (SG) is introduced, while section 5.1 describes the process of implementing this algorithm. The main conclusion here is that despite its relative anonymity, SG is a solid algorithm. Corno et al. already showed the algorithm to be capable of handling the 0/1 Multiple Knapsack Problem [4] and the Cellular Automata Problem [6]; the implementation of SCP-SG shows that the algorithm can also solve the Vertex Cover Problem, while the implementation of PSG confirms that the algorithm is capable of evolving high-quality cellular automata.

Furthermore, the implementation of PSG demonstrates that the "island model" of parallel evolutionary algorithms described in section 3.3.3 can be successfully applied to a non-standard, implicit-population algorithm such as SG.

Taking a closer look at the Cellular Automata Problem, we conclude that the fault vector covering method of TPG evaluation is a good alternative to the much more time-consuming fault simulation approach. This evokes the analogy of the differences between heuristic search algorithms (evolutionary algorithms, simulated annealing, among others) and exhaustive methods such as depth-first search; heuristic search methods are capable of finding near-optimal solutions to a problem in a fraction of the time needed for an exhaustive search.

Another lesson learned from the Cellular Automata Problem is that elementary cellular automata are good test pattern generators for BIST circuitry. This confirms the findings of Corno et al. [6] and Fišer [29]. Moreover, the cellular automata evolved by PSG have only four of the 256 CA rules available to them, yet yield a fault coverage upwards of 93% for all benchmark circuits considered.

8.2 Scalability and efficiency

In section 3.1.1 we introduced the general concept of scalability, defined as the ability to utilize increasing numbers of processors effectively. By this definition, PSG was shown to scale reasonably well: an increase in the number of processors translates to either a reduction in execution time, or an increase in solution quality.

In section 3.2.1 two approaches to measuring speedup of PEA's are presented: the convergence-based approach and the solution-quality-based approach. Since there is no strict consensus on which of these should be used, both were attempted.

When convergence-based speedup is considered, PSG shows no signs of time savings when more processors are utilized. This is in no way a bad sign, since the solution quality attained by high-p configurations does tend to exceed that of the sequential implementation.

The situation is different when the solution-quality-based approach to speedup is adopted. In this case, the algorithm does show clear time savings for higher-p configurations, and the speedup $S(n, p)$ is an increasing function of p. Aggregate speedup is depicted in Figure 7.16; this is perhaps the most important graph in the document, and so it is reproduced here once more.

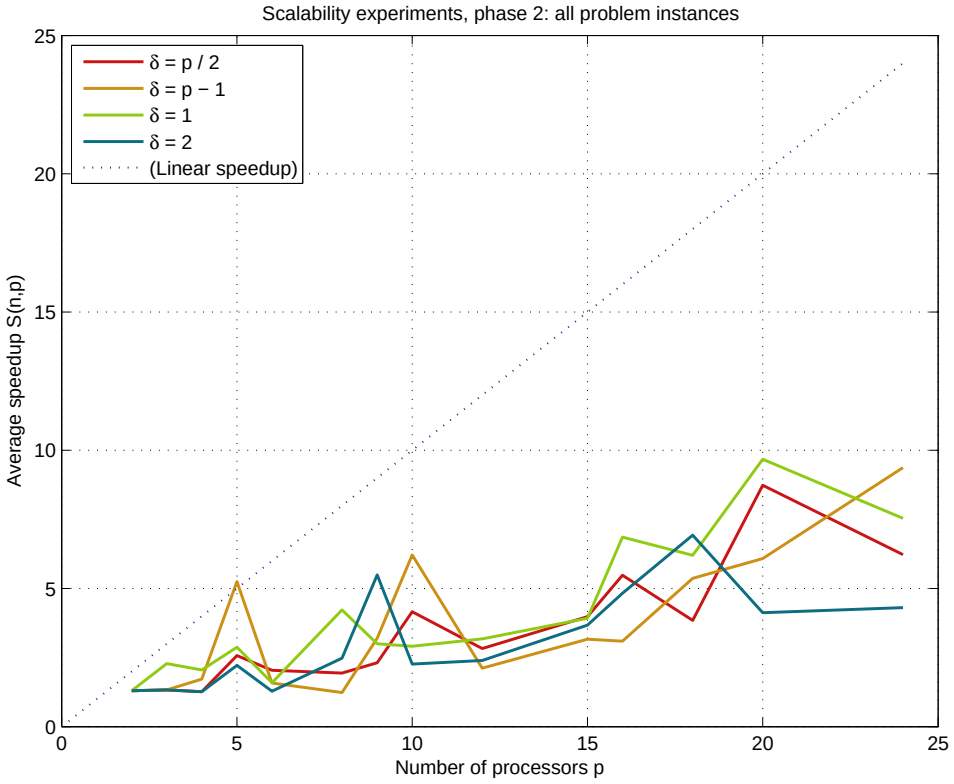


Figure 7.16: Speedup $S(n, p)$ of PSG for different combinations of p and δ for all ISCAS circuits tested. The dotted line denotes the theoretical linear speedup $S(n, p) = p$.

Speedup was found to depend somewhat on the degree of the interconnection network; however, this dependence does not fully correspond to the behavior predicted by the numerical analysis given in [12].

The final conclusion is that PSG is a processor-efficient algorithm. The extension of the Selfish Gene Algorithm onto multiple processors gives users the ability to reduce execution time or increase solution quality.

8.3 Discussion

The main unanswered question is whether there is a correlation between the level of convergence and solution quality. Many PEA's display such a correlation, and it is often taken as an assumption when analyzing algorithms numerically [3, 12].

PSG does not appear to exhibit such a correlation. In fact, the results of convergence-based speedup testing can neither confirm nor deny the hypothesis. Further research would be needed to analyze the dependence between convergence and solution quality for algorithms derived from the Selfish Gene Algorithm.

The uncertainty of this matter affects the assessment of the deterministic self-adaption schedule descibed

in section 4.2.3. Like the methods from which it is derived, this method assumes that the hypothesis holds and that there is a clear relation between convergence and solution quality. When the hypothesis cannot be confirmed, the mechanism's foundation is weakened.

It should also be noted that no comparisons were made between the self-adaption mechanism and any other methods of setting the mutation probability. An objective evaluation of the self-adaption strategy cannot be made until further research provides results of such a comparison.

PSG was compared head-to-head with a random search algorithm. The results of this comparison are less than favorable. The question why PSG was not able to outperform the random search algorithm remains unanswered. A hypothesis was stated that the lack of a performance gain was due to the limited execution time available for the experiment. However, further experiments would be needed to confirm this.

The last issue concerns speedup again. The degree of connectivity resulting in the highest values of speedup does not correspond to the value accepted as "usually optimal" by both experimental evaluation [13] and numerical analysis [12]. More research would be needed to examine this issue in-depth. Of particular interest is the question to what extent the numerical analysis of general PEA's may be applied to the Parallel Selfish Gene Algorithm.

9 Bibliography

[1] C.G. Langton et al. *Artificial Life*. Addison-Wesley, 1989.

[2] R. Dawkins. *The Ancestor's Tale: A Pilgrimage to the Dawn of Life*. Houghton Mifflin, Boston, 2004.

[3] T. Bäck, D.B. Fogel, and Z. Michalewicz. *Evolutionary Computation 1: Basic Algorithms and Operators*. CRC Press, 2000.

[4] F. Corno, M.S. Reorda, and G. Squillero. The selfish gene algorithm: a new evolutionary optimization strategy. In *ACM symposium on Applied Computing*, pages 349–355, 1998.

[5] R. Dawkins. *The Selfish Gene*. Oxford University Press, 2nd edition, 1989.

[6] F. Corno, M.S. Reorda, and G. Squillero. Exploiting the selfish gene algorithm for evolving cellular automata. In *International Joint Conference on Neural Networks (IJCNN'2000)*, pages 577–581, 2000.

[7] P. Tvrdík. *Parallel Systems and Algorithms*. ČVUT, Prague, 2nd edition, 1999.

[8] F.P. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1975.

[9] A.D. Bethke. Comparison of genetic algorithms and gradient-based optimizers on parallel processors: Efficiency of use of processing capacity. Technical Report 197, University of Michigan, Logic of Computers Group, 1976.

[10] J.H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[11] M. Tomassini. Parallel and Distributed Evolutionary Algorithms: a Review. In K. Miettinen, P. Neittaanmäki, M.M. Mäkelä, and J. Périaux, editors, *Evolutionary Algorithms in Engineering and Computer Science*. Wiley, 1999.

[12] E. Cantú-Paz. *Efficient and Accurate Parallel Genetic Algorithms*. Kluwer Academic Publishers, 2000.

[13] H. Cao, J. Yu, L. Kang, and R.I.B. McKay. An experimental study of some control parameters in parallel genetic programming. *Neural, Parallel & Scientific Computations*, 11(4):377–394, 2003.

[14] E. Alba and J.M. Troya. A survey of parallel distributed genetic algorithms. *Complexity*, 4(4):31–52, 1999.

[15] P.R. Grant. *Ecology and Evolution of Darwin's Finches*. Princeton University Press, 1999.

[16] P. Liu, F. Lau, M.J. Lewis, and C. Wang. A New Asynchronous Parallel Evolutionary Algorithm for Function Optimization. In *Parallel Problem Solving from Nature (PPSN VII)*, pages 401–410, 2002.

[17] P.B. Grosso. *Computer Simulations of Genetic Adaptation: Parallel Subcomponent Interaction in a Multilocus Model*. PhD thesis, University of Michigan, 1985.

[18] T. Bäck. Self-Adaptation in Genetic Algorithms. In *First European Conference on Artificial Life*, 1992.

[19] T. Bäck and M. Schütz. Intelligent Mutation Rate Control in Canonical Genetic Algorithms. In *International Syposium on Methodologies for Intelligent Systems*, pages 158–167, 1996.

[20] B.T. Skinner, H.T. Nguyen, and D.K. Liu. Performance study of a multi-deme parallel genetic algorithm with adaptive mutation. In *2nd International Conference on Autonomous Robots and Agents (ICARA 2004)*, pages 13–15, 2004.

[21] ISO/IEC. International Standard 14882, Programming Languages – C++. Technical report, International Organization for Standardization, 2003.

[22] Boost. *Boost C++ Libraries*. http://www.boost.org/.

[23] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*. http://www.mpi-forum.org/.

[24] Kitware. *CMake Cross Platform Make*. http://www.cmake.org/.

[25] D. Roundy. Darcs: Distributed Version Management in Haskell. In *ACM SIGPLAN workshop on Haskell*, pages 1–4, 2005.

[26] R.M. Karp. Reducibility Among Combinatorial Problems. *Complexity of Computer Computations*, 43:85–103, 1972.

[27] D. Gleich. Erdős-Rényi Random Graphs: Seeing the Giant Component, 2006. http://stanford.edu/~dgleich/demos/matlab/random_graphs/erdosreyni.html.

[28] A. Dharwadker. The Vertex Cover Algorithm, 2006. http://www.geocities.com/dharwadker/vertex_cover/.

[29] P. Fišer and H. Kubátová. Pseudorandom Testability – Study of the Effect of the Generator Type. *Acta Polytechnica*, 45(2):47–54, August 2005.

[30] S. Wolfram. *Cellular Automata and Complexity: Collected Papers*. Addison-Wesley, 1994.

[31] F. Brglez, D. Bryan, and K. Kozminski. Combinational profiles of sequential benchmark circuits. In *International Symposium on Circuits and Systems*, pages 1929–1934, 1989.

[32] VLSI Research Group, Department of Computer Science and Engineering, Faculty of Electrical Engineering, ČVUT. *Atalanta-M 1.1*, 2005. http://service.felk.cvut.cz/vlsi/prj/Atalanta-M/.

[33] H.K. Lee and D.S. Ha. Atalanta: an Efficient ATPG for Combinational Circuits. Technical Report 93-12, Dept. of Elect. Eng., Virginia Polytechnic Institute and State University, 1993.

[34] H.K. Lee and D.S. Ha. HOPE: an Efficient Parallel Fault Simulator for Synchronous Sequential Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15:1048–1058, 1996.

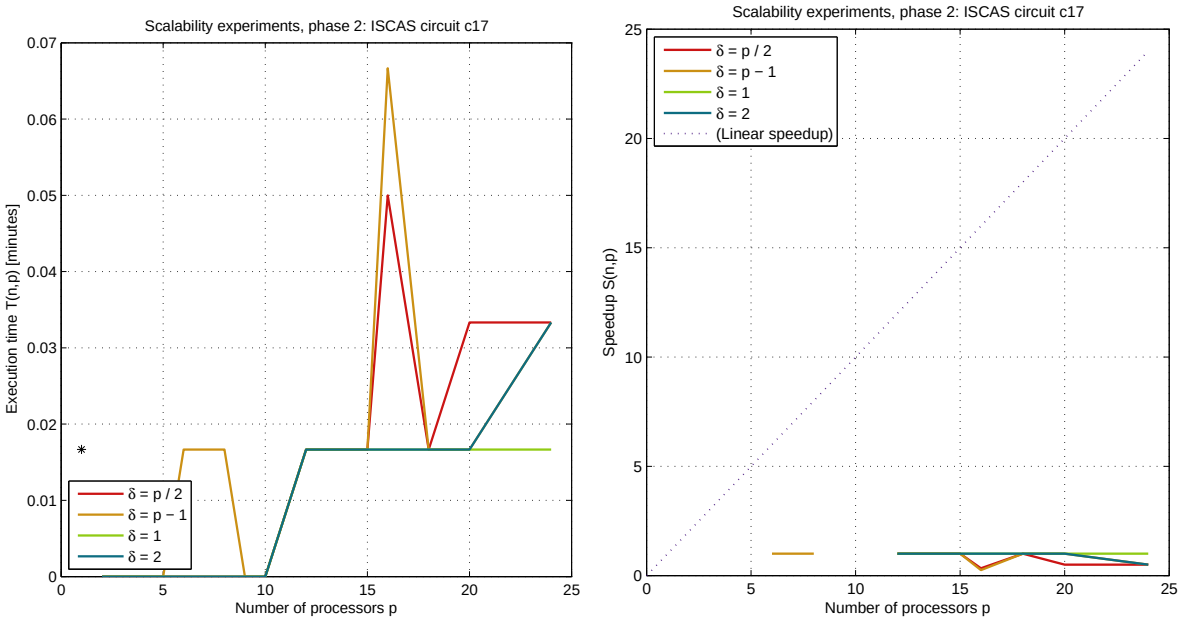
[35] K. Cattell and S. Zhang. Minimal Cost One-Dimensional Linear Hybrid Cellular Automata of Degree Through 500. *Journal of Electronic Testing*, 6(2):255–258, 1995.

A Detailed results of the scalability experiment

In section 7.3.2 we describe phase 2 of the scalability experiments, where we determine $T(n, p)$ and $S(n, p)$ for several ISCAS benchmark circuits using four distinct schemes for setting δ. In this section we will reproduce the concrete results. For each problem instance there are two figures: one shows $T(n, p)$ and $SU(n)$, the other shows $S(n, p)$. Table A.1 shows which figure corresponds to which problem instance.

ISCAS circuit	**No. of genes** n	**Graph of** $T(n, p)$	**Graph of** $S(n, p)$
c17	10	Figure A.1a	Figure A.1b
c432	72	Figure A.2a	Figure A.2b
c880	120	Figure A.3a	Figure A.3b
c1908	66	Figure A.4a	Figure A.4b
c2670	466	Figure A.5a	Figure A.5b
c3540	100	Figure A.6a	Figure A.6b
c5315	356	Figure A.7a	Figure A.7b
c6288	64	Figure A.8a	Figure A.8b
c7552	414	Figure A.9a	Figure A.9b

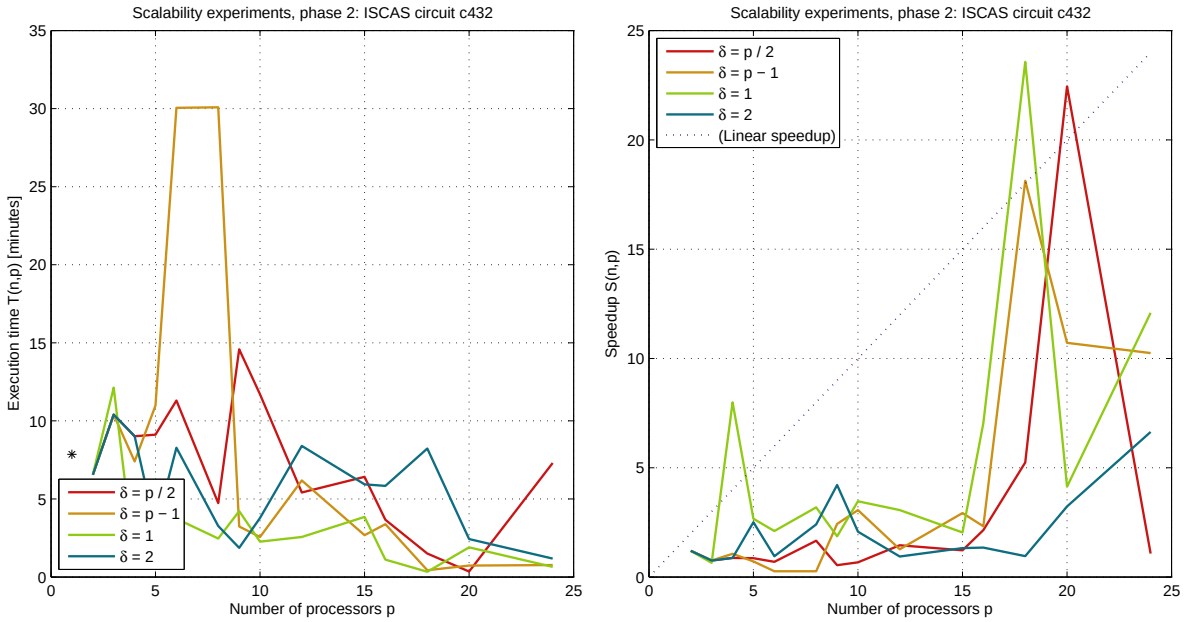
Table A.1: ISCAS'85 benchmark circuits and their scalability graphs.



(a) Execution time $T(n, p)$.

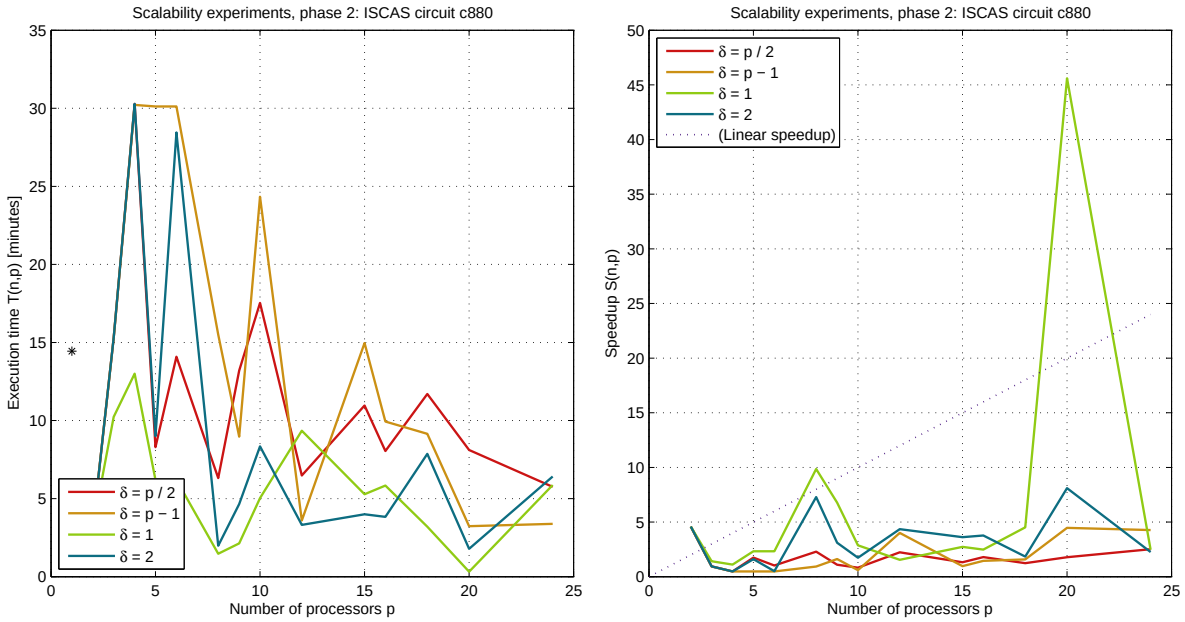
(b) Speedup $S(n, p)$.

Figure A.1: Execution time $T(n, p)$ and speedup $S(n, p)$ of PSG for different combinations of p and δ on ISCAS circuit c17. In the left figure, the black asterisk denotes $SU(n)$, the time taken by the sequential implementation. In the right figure, the dotted line denotes the theoretical linear speedup $S(n, p) = p$.



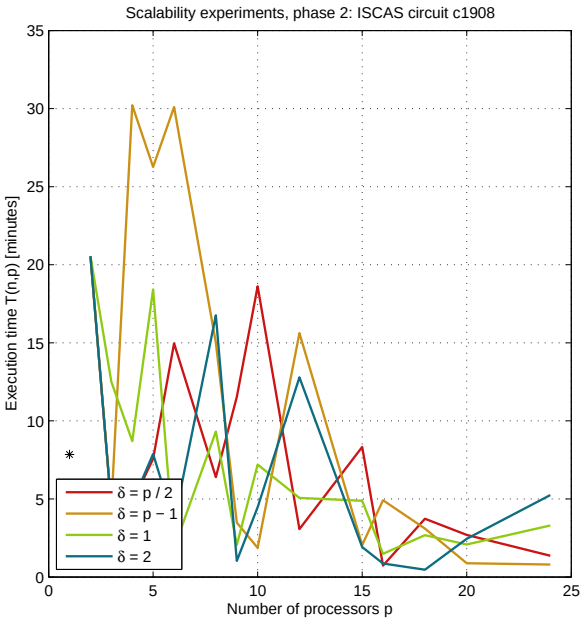
(a) Execution time $T(n, p)$. (b) Speedup $S(n, p)$.

Figure A.2: Execution time $T(n, p)$ and speedup $S(n, p)$ of PSG for different combinations of p and δ on ISCAS circuit c432. In the left figure, the black asterisk denotes $SU(n)$, the time taken by the sequential implementation. In the right figure, the dotted line denotes the theoretical linear speedup $S(n, p) = p$.

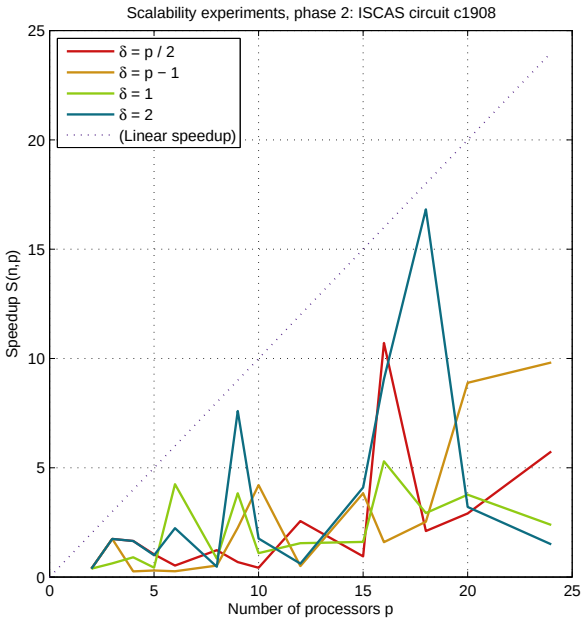


(a) Execution time $T(n, p)$. (b) Speedup $S(n, p)$.

Figure A.3: Execution time $T(n, p)$ and speedup $S(n, p)$ of PSG for different combinations of p and δ on ISCAS circuit c880. In the left figure, the black asterisk denotes $SU(n)$, the time taken by the sequential implementation. In the right figure, the dotted line denotes the theoretical linear speedup $S(n, p) = p$.

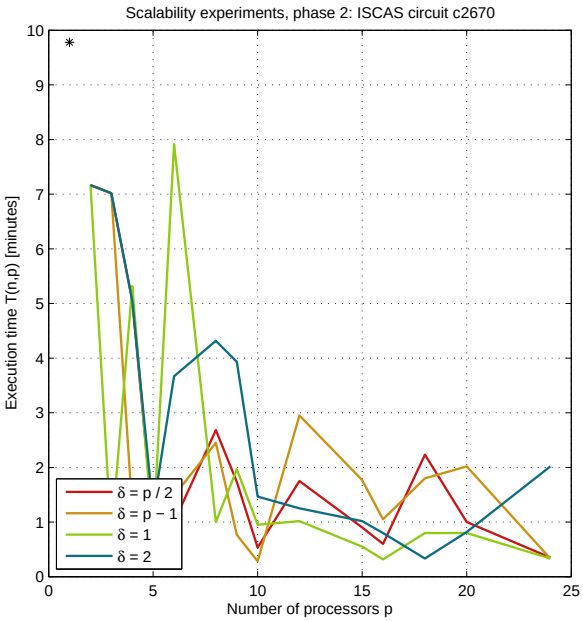


(a) Execution time $T(n, p)$.

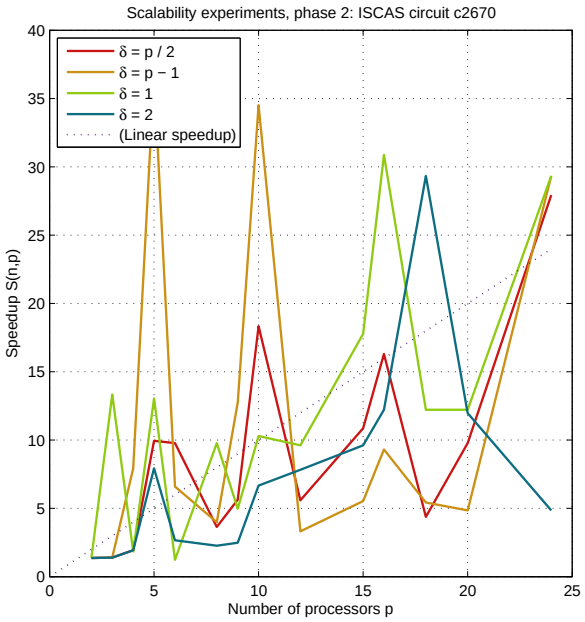


(b) Speedup $S(n, p)$.

Figure A.4: Execution time $T(n, p)$ and speedup $S(n, p)$ of PSG for different combinations of p and δ on ISCAS circuit `c1908`. In the left figure, the black asterisk denotes $SU(n)$, the time taken by the sequential implementation. In the right figure, the dotted line denotes the theoretical linear speedup $S(n, p) = p$.

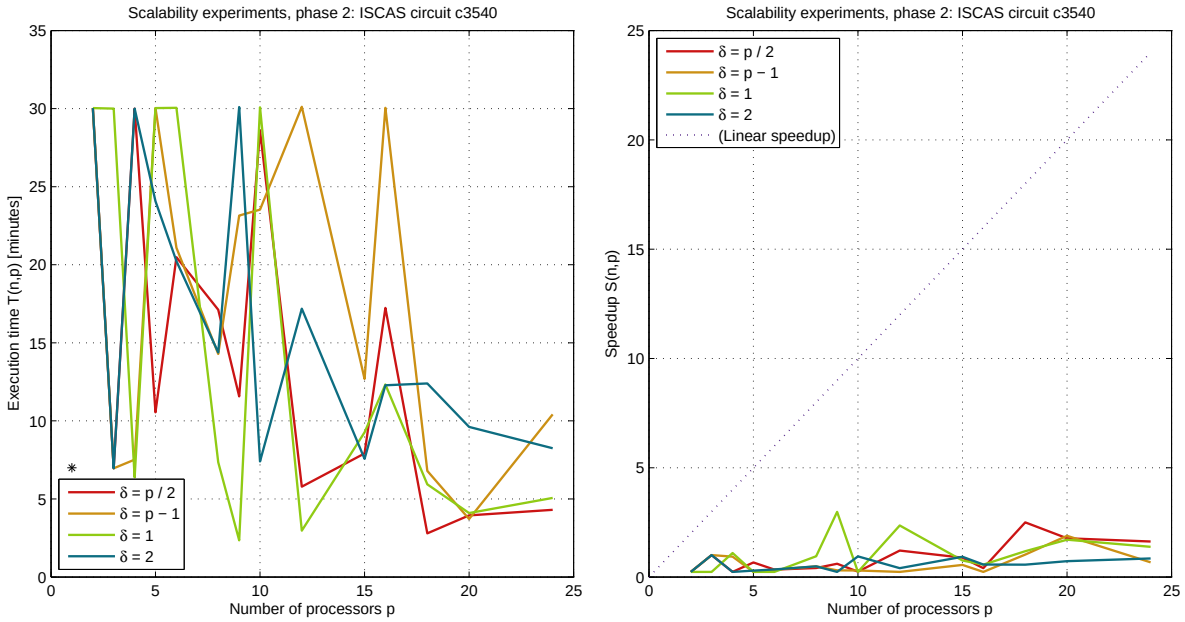


(a) Execution time $T(n, p)$.



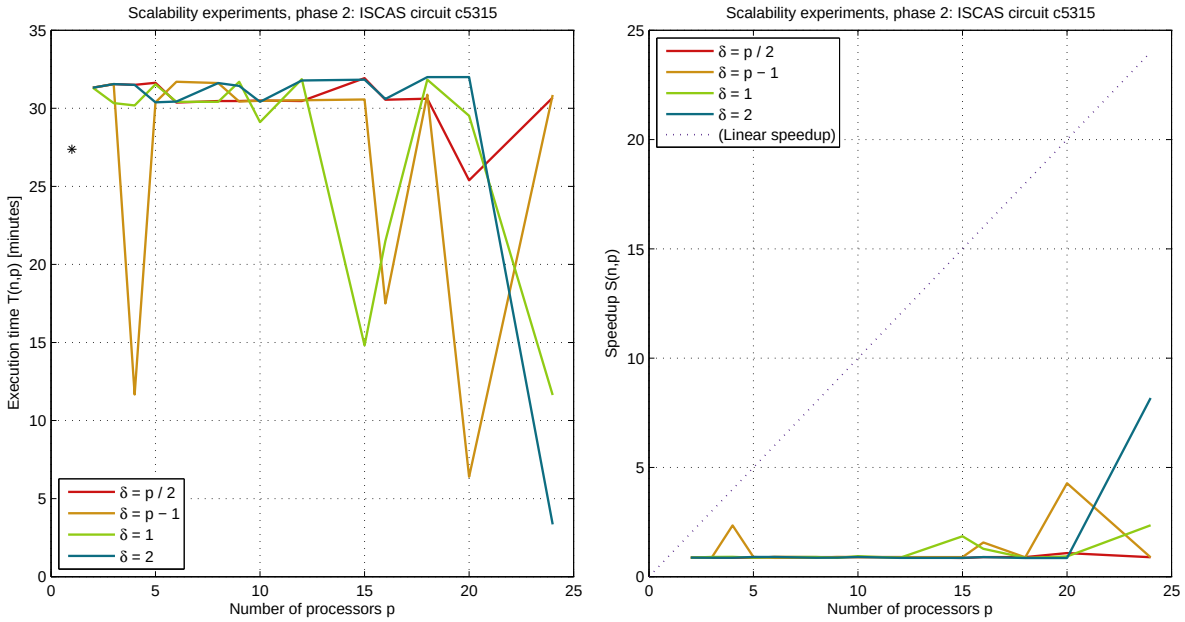
(b) Speedup $S(n, p)$.

Figure A.5: Execution time $T(n, p)$ and speedup $S(n, p)$ of PSG for different combinations of p and δ on ISCAS circuit `c2670`. In the left figure, the black asterisk denotes $SU(n)$, the time taken by the sequential implementation. In the right figure, the dotted line denotes the theoretical linear speedup $S(n, p) = p$.



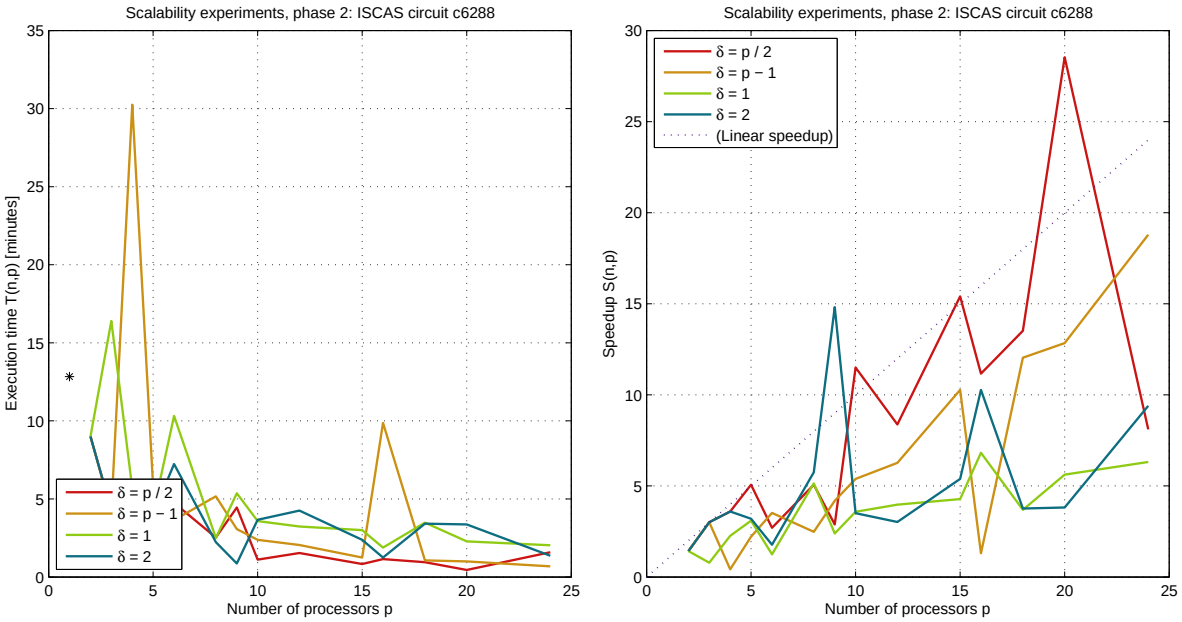
(a) Execution time $T(n, p)$. (b) Speedup $S(n, p)$.

Figure A.6: Execution time $T(n, p)$ and speedup $S(n, p)$ of PSG for different combinations of p and δ on ISCAS circuit c3540. In the left figure, the black asterisk denotes $SU(n)$, the time taken by the sequential implementation. In the right figure, the dotted line denotes the theoretical linear speedup $S(n, p) = p$.



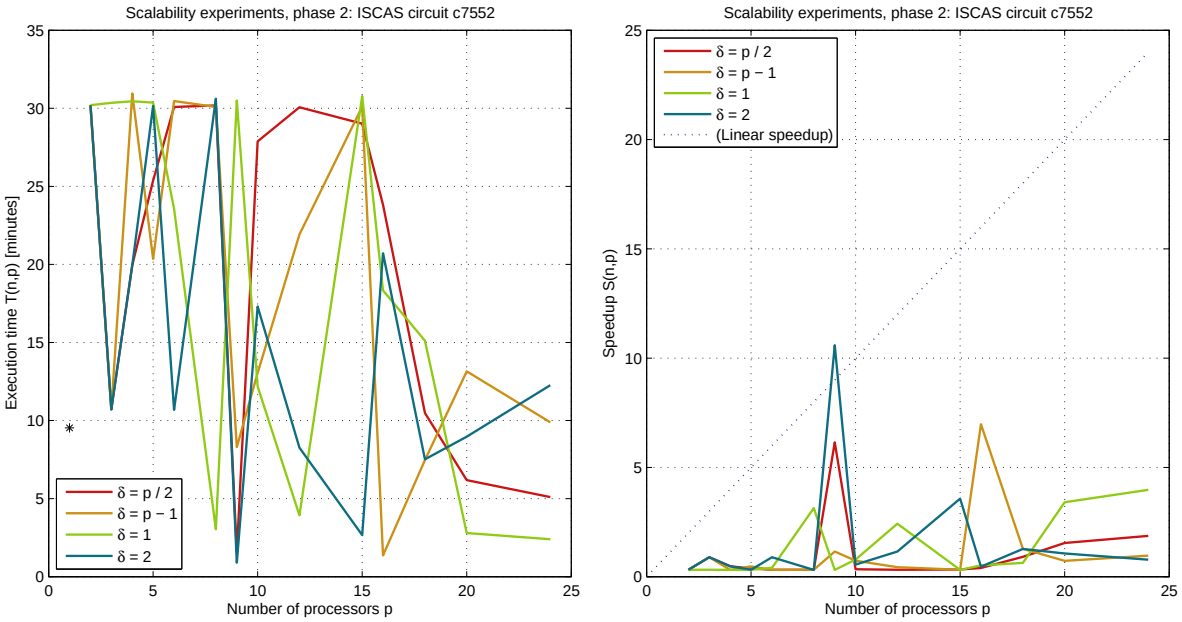
(a) Execution time $T(n, p)$. (b) Speedup $S(n, p)$.

Figure A.7: Execution time $T(n, p)$ and speedup $S(n, p)$ of PSG for different combinations of p and δ on ISCAS circuit c5315. In the left figure, the black asterisk denotes $SU(n)$, the time taken by the sequential implementation. In the right figure, the dotted line denotes the theoretical linear speedup $S(n, p) = p$.



(a) Execution time $T(n, p)$. (b) Speedup $S(n, p)$.

Figure A.8: Execution time $T(n, p)$ and speedup $S(n, p)$ of PSG for different combinations of p and δ on ISCAS circuit c6288. In the left figure, the black asterisk denotes $SU(n)$, the time taken by the sequential implementation. In the right figure, the dotted line denotes the theoretical linear speedup $S(n, p) = p$.



(a) Execution time $T(n, p)$. (b) Speedup $S(n, p)$.

Figure A.9: Execution time $T(n, p)$ and speedup $S(n, p)$ of PSG for different combinations of p and δ on ISCAS circuit c7552. In the left figure, the black asterisk denotes $SU(n)$, the time taken by the sequential implementation. In the right figure, the dotted line denotes the theoretical linear speedup $S(n, p) = p$.