PARALLELIZATION OF POPULATION-BASED EVOLUTIONARY ALGORITHMS FOR COMBINATORIAL OPTIMIZATION PROBLEMS

THÈSE N^o 2046 (1999)

PRÉSENTÉE AU DÉPARTEMENT D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDERALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Patrice Roger CALÉGARI
Magistère en informatique et modélisation, DEA d'informatique fondamentale, Université Claude Bernard, Lyon, France
de nationalité française

acceptée sur proposition du jury :

Prof. G. Coray, directeur de thèse
Prof. M. Cosnard, rapporteur
Prof. A. Hertz, rapporteur
Prof. D. Mange, rapporteur
Prof. D. Trystram, rapporteur
J.-F. Wagen, rapporteur

Lausanne, EPFL
1999

To my parents

Abstract

The objective of the present work is to make efficient parallelization of evolutionary algorithms (EA) easier in order to solve large instances of difficult combinatorial optimization problems within an acceptable amount of time, on parallel computer architectures.

No known technique allows one to exactly solve within an acceptable amount of time, such difficult combinatorial optimization problems (NP-complete). Moreover, traditional heuristics that are used to find sub-optimal solutions are not always satisfactory since they are easily attracted by local optima. Evolutionary algorithms (EA), that are heuristics inspired by natural evolution mechanisms, explore different regions of the search space concurrently. They are thus rarely trapped in a local optimum and are well suited to treat difficult combinatorial optimization problems. Their behavior can be improved by hybridizing (i.e., combining) them with other heuristics (EA or not). Unfortunately, they are greedy in computation power and memory space. It is thus interesting to parallelize them. Indeed, the use of parallel computers (with dozens of processors) can speed up the execution of EAs and provides the large memory space they require. It is possible to take benefit of the intrinsic parallelism of EAs (e.g., for the concurrent exploration of the search space) in order to design efficient parallel implementations. However each EA has its own characteristics and therefore a general rule cannot be defined.

This thesis starts with a description of the state of the art in which the different existing approaches and terminologies are outlined. The fundamental ingredients of EAs are then detailed and these ingredients are grouped by a classification tool called TEA (*Table of Evolutionary Algorithms*). This table is taken as a basis for the analysis of the criteria that influence the parallelization of EAs in order to define parallelization rules. The analysis considers especially the implementation of hybrid EAs on MIMD-DM[1] architectures. A notation of the granularity of parallel EAs is proposed. Further to this analysis, an object-oriented library named APPEAL (*Advanced Parallel Population-based Evolutionary Algorithm Library*) that applies the parallelization rules is designed and then used in order to experimentally validate these rules. During the experiments, different hybrid EAs are executed on a network of workstations in order to treat two problems: first the optimization of the best set of transceiver sites in a mobile radio network and second the classical graph coloring problem. Finally, a comparison of results and a discussion about future work conclude this thesis.

Key words: parallel computing, evolutionary algorithms, combinatorial optimization, taxonomy, object-oriented library, transceiver siting, graph coloring.

[1]MIMD-DM stands for Multiple Instruction stream, Multiple Data stream, Distributed Memory.

Version abrégée

Le but de ce travail de thèse est de faciliter la parallélisation efficace des algorithmes d'évolution (e.g., les algorithmes génétiques, les systèmes de fourmis, etc.) afin de résoudre, en un temps acceptable, de grosses instances de problèmes d'optimisation combinatoire difficiles sur des architectures parallèles.

Aucune technique connue ne permet de résoudre, en un temps acceptable et de façon exacte, les grosses instances des problèmes d'optimisation combinatoire NP-complets (aussi appelés "difficiles"). De plus, les heuristiques traditionnelles qui sont utilisées pour trouver des solutions approchées ne donnent pas toujours satisfaction car elles sont facilement attirées par les optima locaux. Les algorithmes d'évolution (AE), qui sont des heuristiques inspirées par l'évolution des systèmes biologiques, explorent simultanément différentes régions de l'espace de recherche. Ils sont donc peu sensibles à l'attraction d'un optimum local et conviennent bien pour résoudre des problèmes d'optimisation combinatoire difficiles. Leur comportement peut être amélioré en les hybridant (i.e., en les combinant) entre eux ou avec d'autres heuristiques. Malheureusement, ils sont gourmands en temps de calcul et en espace mémoire et il est donc intéressant de les paralléliser. En effet l'utilisation d'ordinateurs parallèles (comprenant des dizaines de processeurs) peut permettre d'accélérer leur exécution et de fournir l'espace mémoire important dont ils ont besoin. Il est possible de tirer profit du parallélisme intrinsèque des AE (e.g., la recherche simultanée de plusieurs solutions) pour en concevoir des implémentations parallèles efficaces, toutefois chaque AE possède ses propres caractéristiques et une règle générale ne peut pas être définie.

Cette thèse commence par un état de l'art du domaine mettant l'accent sur les différentes approches et terminologies existantes. La caractérisation de chacune des composantes fondamentales des AE est alors détaillée et ces composantes sont regroupées dans un outil de classification appelé TEA (tableau des algorithmes d'évolution). Ce tableau est utilisé comme base pour l'analyse des critères influençant la parallélisation des AE afin de définir des règles de parallélisation. L'analyse considère spécialement l'implémentation d'AE hybrides sur des architectures MIMD-DM[2]. Une notation de la granularité des AE parallèles y est entre autre proposée. Suite à cette analyse, une librairie orientée objet (nommée APPEAL) est conçue, puis utilisée pour valider expérimentalement les règles de parallélisation qui ont été définies. Différents AE hybrides sont ainsi exécutés sur un réseau de stations de travail pour traiter deux problèmes : le placement des antennes d'un réseau de téléphonie mobile, et le problème classique de coloration d'un graphe. Finalement, les résultats obtenus sont comparés et une discussion sur les suites à donner à ce travail conclut le rapport.

Mots clés: parallélisme, algorithmes d'évolution, optimisation combinatoire, taxonomie, conception logicielle orientée objet, planification de réseaux radio, coloration de graphes.

[2]MIMD-DM signifie machine à flot d'instructions multiple travaillant sur un flot de données multiple avec une mémoire distribuée.

It is only with the heart that one can see rightly.
What is essential is invisible to the eye.
Le Petit Prince, 1943.
Antoine de Saint-Exupéry (1900–1944)

Acknowledgments

I would like to thank Prof. Giovanni Coray, director of the computer science theory laboratory (LITH) at the Swiss Federal Institute of Technology (EPFL), who accepted to be my thesis supervisor. I also want to thank Dr. Pierre Kuonen, researcher in the same laboratory, who was my mentor. They both gave me a lot of fruitful advice for completing this thesis and contributed to the friendly atmosphere of the laboratory.

I would like to thank those who accepted to be referees and members of the jury of this PhD thesis: Prof. Michel Cosnard (director of the INRIA Lorraine research unit, Nancy, France), Prof. Denis Trystram (responsible for the parallel computing team at the modeling and computing laboratory, IMAG, Grenoble, France), Prof. Alain Hertz (professor at the chair of operational research at EPFL, and president of the Swiss operational research society), Prof. Daniel Mange (director of the logic systems laboratory at EPFL), and Jean-Frédéric Wagen (project leader at the Swiss telecommunication company Swisscom). I also want to thank Prof. Jacques Zahnd who accepted to chair the jury.

I am very grateful to those who accepted to read the first version of this manuscript and whose remarks permitted to improve it: Dr. Frédéric Vivien, Dr. Afzal Ballim, Dave Nespoli, and Dr. Frédéric Guidec whom it was a pleasure to share the same office and to work with for three years. I would like to thank Sophie Fallot-Josselin, Daniel Wagner, Jean-Michel Coupé, and Mahmoud El Husseini who I enjoyed to work with in the project STORMS[3]. I also want to thank Dr. Franck Nielsen whose theoretical point of view was rewarding, and Dr. Daniel Kobler who completed his PhD thesis within the same project as I, and whose collaboration allowed the exchange of original ideas.

I thank all those I love and who supported me. My parents who gave me the love for science and who are thus at the origin of this work, Anžela who gave me the power to complete it, and my brother Didier who designed the logo of the project LEOPARD[4].

This work is part of the project LEOPARD that was funded by the Swiss National Science Foundation (grants 2100–45070.95/1 and 2000–52594.97). It is a logical continuation of the project PAC (Parallelization of Combinatorial Optimization Algorithms) funded by the same foundation (grants SPP-IF 5003–034349, 1993–96). A part of this work was done within the European project STORMS that was framed in the 4^{th} ACTS Program (Advanced Communications Technologies & services) partially funded by the European Commission (AC016) and Swiss fund OFES (1995–98). The experiments were made on networks of workstations provided by EPFL.

[3]Project STORMS: Software Tools for the Optimization of Resources in Mobile Systems.
[4]Project LEOPARD: parallel population-based methods for combinatorial optimization.

Contents

Abstract iii

Version abrégée *(French abstract)* v

Acknowledgments vii

1 Introduction 1

2 State of the art 5

2.1 Combinatorial optimization problems 5

2.1.1 Definitions 5

2.1.2 Classes 7

2.2 Classical heuristics 8

2.3 Evolutionary algorithms 9

2.3.1 Genetic algorithm 11

2.3.2 Evolution strategy 12

2.3.3 Evolutionary programming 13

2.3.4 Ant colony system 13

2.3.5 Population-based incremental learning 16

2.3.6 Other evolutionary algorithms 16

2.3.7 Hybrid approaches 18

2.4 Parallel computing 20

2.4.1 Definition of a parallel algorithm 20

2.4.2 Parallel computer architectures 21

2.4.3 Parallel computing constraints 22

2.4.4 Classical parallel algorithm models 26

2.5 Classification of parallel EAs 28

3 Evolutionary algorithm mechanisms 31

3.1 The need for a proper parallelization 31

3.2 An original taxonomy of EAs 31

3.2.1 Motivation for parallelization 31

3.2.2 Background 32

3.2.3 Main ingredients of an EA . . . 32
3.2.4 The basic TEA . . . 36
3.2.5 Hierarchical ingredients . . . 37
3.2.6 Examples . . . 41
3.2.7 Extensibility . . . 44
3.3 About islands and topology . . . 44
3.3.1 Structured space phenomenon . . . 44
3.3.2 Island phenomenon . . . 45
3.3.3 Discussion . . . 45
3.4 An island-based genetic ant algorithm . . . 46
3.4.1 Motivation . . . 46
3.4.2 Description . . . 47

4 Parallelization of evolutionary algorithms 49
4.1 Parallelization analysis . . . 49
4.1.1 The architectural choice . . . 50
4.1.2 Levels of parallelization . . . 50
4.1.3 Influence of the main ingredients . . . 55
4.1.4 Other important criteria for parallelization . . . 58
4.1.5 Hybrid algorithms . . . 58
4.2 Case study . . . 60
4.2.1 Parallel island-based genetic algorithms . . . 60
4.2.2 Parallel island-based ant system . . . 62
4.2.3 Parallel island-based genetic ant algorithm . . . 66
4.3 A library for evolutionary algorithms . . . 68
4.3.1 Requirements . . . 68
4.3.2 Existing libraries . . . 69
4.3.3 Object-oriented model of APPEAL . . . 71
4.3.4 Implementation of APPEAL . . . 79
4.3.5 Current state and future evolution of APPEAL . . . 80
4.4 Alternative approaches to the parallelization of EAs . . . 81
4.4.1 Parallelization based on autonomous agents . . . 81
4.4.2 Asynchronous parallelization . . . 82

5 Transceiver siting application 85
5.1 Problem modeling . . . 85
5.1.1 Urban radio wave propagation simulation software . . . 85
5.1.2 Cells . . . 86
5.1.3 Examples of instances. . . . 87
5.1.4 Modeling of the service. . . . 88
5.1.5 Problem representation using set systems . . . 89
5.1.6 Hitting set and set cover problems . . . 89
5.2 Greedy-like algorithms . . . 90

5.3 Experimental conditions 91
5.3.1 Network configuration for speed-up measurements 91
5.3.2 Influence of islands on execution time 92
5.3.3 The choice of the number of generations 93
5.4 Parallel island-based genetic algorithms 94
5.5 Parallel island-based ant systems 98
5.6 Parallel island-based genetic ant algorithm 100
5.7 Quality of the results 101
5.7.1 Results 102
5.7.2 Influence of islands on the results 104
5.7.3 Results of other algorithms 104
5.7.4 Summary 105
5.7.5 Cooperation with other projects 106

6 Graph coloring application **107**
6.1 Definition of the problem 107
6.2 Examples of instances 108
6.3 Greedy-like algorithm 109
6.4 Parallel island-based genetic algorithms 109
6.5 Parallel island-based ant systems 111
6.6 Parallel island-based genetic ant algorithm 114
6.7 Quality of the results 115
6.7.1 Results 115
6.7.2 Summary 116

7 Conclusion **119**
7.1 Summary 119
7.2 Major contributions of this work 121
7.3 Perspectives 121

A Glossary and acronyms **123**
A.1 Glossary of usual evolutionary terms 123
A.2 Frequently used acronyms 124

B Demonstrations **125**
B.1 Theoretical efficiency with indivisible islands 125
B.2 Theoretical efficiency with partitioned islands 126

List of Algorithms **129**

Bibliography **131**

Index **141**

The last thing one knows when
writing a book is what to put first.
Pensées, 1670. Blaise Pascal (1623–1662)

Chapter 1

Introduction

Since the 50's, computer science and its attendant research fields have evolved very quickly. Hardware is faster and faster every year, and software allows one to solve problems that were unconceivable a few years ago. Four kinds of interests can be distinguished among research fields in computer science:

- the kind of problems that must be solved (optimization, numerical simulation, artificial intelligence, compilation, etc.),
- the class of algorithms that can be used (greedy, evolutionary, output sensitive, etc.),
- the methodologies for producing software (language, software engineering, implementation choices, etc.),
- the hardware that must be designed, constructed and/or used (architecture, chips, cost, etc.).

The work presented in this thesis is at the cross-road of combinatorial optimization, evolutionary computation, object-oriented programming, and parallel computing.

Combinatorial optimization problems have existed for ages [82], but the first attempts to solve them with computers started only 30 years ago. At the beginning, only small problem instances were treated but since the 80's the growth of computation power has permitted solutions to complex problems in a larger search space than in the past. One of the biggest challenge that is given to computer scientists is to solve huge combinatorial optimization problem instances. Experience shows that constructive methods (such as greedy algorithms) are too easily trapped in local optima to solve such problems efficiently [6, 56]. Sequential approaches (such as simulated annealing or tabu search) behave much better than the latter but they converge slowly and are difficult to parallelize in order to be sped up using parallel architectures.

Evolutionary algorithms (EAs) are inspired by biology and natural evolution mechanisms. They can investigate several points of a search space concurrently and are thus rarely trapped in local optima. They are therefore well fitted to treat combinatorial optimization problems efficiently. However, they need a lot of computation power and even if the first EA was introduced by J. Holland in 1975 [63], real experimentations on such algorithms only started in the early 90's. A drawback of the youth of EAs is the lack of theoretical proof of their efficiency. Experience shows their good behavior but global studies are confused by the use of different terms to refer to the same notions in different EAs. The profusion of new terms that refer to well-known notions and techniques leads to an apparent lack of rigor and gives EAs a bad reputation. Definitions and state of the art of EAs for combinatorial optimization problems are given in Chapter 2.

The development of programs that contain more than 100,000 lines of code with complex data structures requires software engineering techniques. Object-oriented programming, that is one of the most popular [76], appeared with Simula in 1967 and Smalltalk in the 70's. It suffered from the lack of real object-oriented compilers that could support all of its concepts. The languages and compilers that appeared later in the 80's were lacking of maturity: they were generating slow programs (e.g., Eiffel [76]) or were not supporting most of object-oriented paradigms (e.g., C++ [95]). The first languages and compilers that could really be trusted for big projects appeared in the 90's (e.g., Eiffel) even if some of them still have important gaps (e.g., C++ [64]).

The reliability of hardware allows the design of parallel computers that run thousand of concurrent processors. The first computers that were designed using parallelism appeared around 1972 (e.g., the 8×8 array of processors ILLIAC IV [60]). Parallel computing implies research in graph theory, topology, programming language theory and computer architecture. The main advantages of processing a program on parallel supercomputers (or networks of workstations) are to speed up their execution and to benefit from a huge memory space (i.e., the sum of the memory of each processor). Until recently, it was mostly studied in order to perform intensive numerical simulations. The interest to deal with irregular problems (that cannot be modeled with regular matrices) only begun recently with the appearance of massively parallel supercomputers. Chapter 2 gives the necessary basics of parallel computing to understand the present work.

The dream would be to design an efficient program that is able to solve any huge combinatorial optimization problem very quickly. This is not realistic, however the increasing cooperation of different research communities makes it possible to profit from each other's knowledge in order to get as close as possible to this situation.

The objective of the present work is to make efficient parallelization of EAs easier. It is not to prove their efficiency but rather to give a clear view of their mechanisms in order to better understand how to parallelize them. The first stage is to extract the fundamental ingredients of well-known EAs, such as genetic algorithms (GA), scatter search (SC), evolution strategies (ES), and ant systems (AS), to define a unified taxonomy. In Chapter 3, the different ingredients are identified and used to propose a classification tool called the TEA (*Table of Evolutionary Algorithms*). This table is then taken as a basis for the analysis of the best way to implement an EA on parallel machines. This analysis considers especially MIMD-DM[1] computers that are more and more used because of their flexibility and their availability (a simple network of workstations can be used as a MIMD-DM machine). Chapter 4 offers a complete description of this analysis and presents the concepts of a software library that was designed from it. This object-oriented library, named APPEAL (*Advanced Parallel Population-based Evolutionary Algorithm Library*), implements most of the rules defined in preceding chapters in order to test them.

As test problems, two applications were chosen. They are used to evaluate speed-ups and to compare algorithms. First the transceiver siting application, a realistic problem related to telecommunications, is treated in Chapter 5. Second, the graph coloring application, a classical combinatorial optimization problem, is dealt with in Chapter 6. The first application makes it possible to apply a part of this work to the European project STORMS[2]. The second application is studied in order to check some experimental results obtained with the transceiver siting application as well as to verify the versatility of the library APPEAL. Finally, conclusion and perspectives are proposed in Chapter 7.

[1]MIMD-DM stands for Multiple Instruction stream, Multiple Data stream, Distributed Memory.
[2]STORMS stands for Software Tools for the Optimization of Resources in Mobile Systems.

There are always two people in every
picture: the photographer and the viewer.
Ansel Adams, photographer (1902–1984)

Chapter 2

State of the art

This chapter presents a quick overview of combinatorial optimization problems and a state of the art of evolutionary algorithms used to solve them. It ends with an introduction to parallel computing notions that are necessary to understand the next chapters.

2.1 Combinatorial optimization problems

2.1.1 Definitions

Many usual computational problems amount to searching for the best choice among different possibilities: what is the shortest path to visit a set of cities? what is the best position for antennas in a radio network? what is the best scheduling for a crew? This section defines the class of *combinatorial optimization problems* that includes all of them.

An *optimization problem* [82] is defined by a set Y and an objective function $f : Y \to \mathbb{R}$. The objective function f assigns to each candidate $y \in Y$ a value $f(y)$. The goal when solving an optimization problem $\mathsf{P}_\mathrm{o}(Y, f)$ is to find an *optimal solution* $y_\mathrm{opt} \in Y$ that minimizes the objective function f, that is $\forall y \in Y, f(y_\mathrm{opt}) \leq f(y)$. It can be noted that an optimal solution y_opt such that $f(y_\mathrm{opt}) = \min_{y \in Y} f(y)$ is not necessarily unique, and that since $\max f(y) = -\min(-f(y))$ restriction to function minimization is without loss of generality. When a neighborhood function $\mathcal{N} : Y \to \mathcal{P}(Y)$ is defined[1], a candidate y_m that verifies $\forall y \in \mathcal{N}(y_\mathrm{m}), f(y_\mathrm{m}) \leq f(y)$ is called a *local* optimal solution. An optimal solution is sometimes called a *global* optimal solution in order to avoid any confusion with a local one.

A *combinatorial problem* is defined by a search space S and a set of constraints $C = \{c_1, c_2, \ldots\}$ that formalizes the problem. A search space is a finite, or possibly countably infinite, set of elements s that are called *candidates*. A candidate that satisfies all the constraints c_i of a combinatorial problem $\mathsf{P}_\mathrm{c}(S, C)$ is said to be a feasible solution (or simply a solution) of $\mathsf{P}_\mathrm{c}(S, C)$ (it is called an infeasible solution of $\mathsf{P}_\mathrm{c}(S, C)$ otherwise).

[1] $\mathcal{P}(Y)$ is the set of subsets of Y.

The aim when solving a combinatorial problem $\mathsf{P}_\mathrm{c}(S, C)$ is to find a feasible solution $s' \in S$ for $\mathsf{P}_\mathrm{c}(S, C)$.

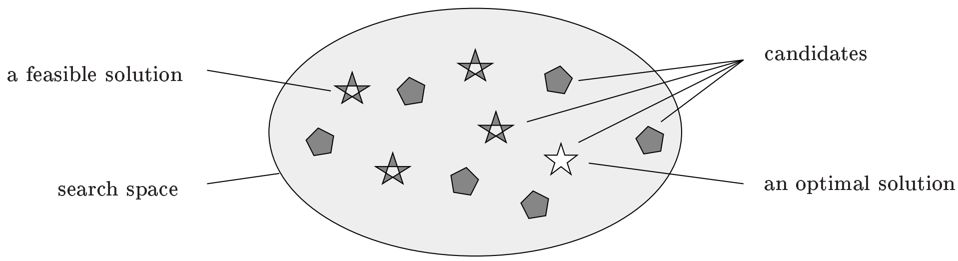


Figure 2.1: *Simple representation of a search space.*

Following the two previous definitions, a *combinatorial optimization problem* can be defined by a set of constraints C, a search space S, and an objective function $f : S \to \mathbb{R}$. The aim when solving a combinatorial optimization problem $\mathsf{P}_\mathrm{co}(S, C, f)$ is to find an optimal solution (according to the objective function f) among the feasible solutions that satisfy the constraints C in S. In other words, to solve $\mathsf{P}_\mathrm{co}(S, C, f)$ is equivalent to determine the set $X \subseteq S$ of all feasible solutions of $\mathsf{P}_\mathrm{c}(S, C)$, and to find $s_\mathrm{opt} \in X, f(s_\mathrm{opt}) = \min_{s \in X} f(s)$ (i.e., to solve $\mathsf{P}_\mathrm{o}(X, f)$). Figure 2.1 gives an informal representation of a search space when solving a combinatorial optimization problem.

Let us illustrate these definitions by taking the *traveling salesman problem (TSP)* as an example of combinatorial optimization problem. First let us define the following combinatorial problem:

- Given a graph G, find a path that visits each vertex of G exactly once.

Let us now add to this problem an objective function $f : \mathbb{N}^n \to \mathbb{R}$ that associates to each path its length. The TSP can then be defined:

- **TSP** (optimization version): Given a graph G, what is the shortest path that visits each vertex of G exactly once?

It can also be defined as:

- **TSP** (evaluation version): Given a graph G, what is the length of the shortest path that visits each vertex of G exactly once?

A combinatorial optimization problem whose solution (or answer) is either "yes" or "no" is called a *recognition (or decision) problem*. For example, the recognition version of the TSP is:

- **TSP** (recognition version): Given a graph G, is there a path of length shorter than ℓ that visits each vertex of G exactly once?

The description of a problem is not general and needs additional data to be solved. A problem together with input data define an *instance* of the problem. Reciprocally, a problem is the set of all its possible instances. For example, the TSP is a problem and the TSP together with a given graph (that represents the road map and the cities of Switzerland for example) is an instance of this problem. Although it is very important to distinguish between a problem and its *instances*, in the remainder of this report the distinction is not explicitly made when no ambiguity is possible.

2.1.2 Classes

A complete description of the different classes of combinatorial optimization problems can be found in [40, 82]. Here is a brief overview of the *P* and *NP* classes.

P is the class of recognition problems that can be solved by a polynomial-time algorithm. They are considered as *easy* problems. For example, P contains the graph connectedness problem:

- Is a given graph G connected?

NP (Non-deterministic Polynomial) is a richer class of recognition problems. For a problem to be in NP, the only requirement is: if x is an instance of the problem whose answer is "yes", then there must exist a concise certificate of x (i.e., of length bounded by a polynomial in the size of x) which can be checked in polynomial time for validity. It is proven that P $\subseteq$ NP, but nobody knows whether P = NP or not [40, 82]. It is however believed that NP $\not\subseteq$ P. This conjecture is one of the most prominent theoretical problems in computer science.

The most difficult problems of NP are called *NP-complete* problems. They have the following properties:

1. No NP-complete problem can be solved by any known polynomial algorithm.

2. If a polynomial algorithm can solve one NP-complete problem, then every NP-complete problem can be solved by a polynomial algorithm.

A *NP-hard* problem is a combinatorial optimization problem whose recognition version is NP-complete.

NP-complete (and NP-hard) problems are considered as computationally intractable. That is, an algorithm that could solve them requires an exponential amount of time. In the worst case, such an algorithm would need to enumerate every possible candidate of their search space. Consequently, only very small instances of such problems can be solved within a reasonable amount of time, and large ones are impractical. A *heuristic* is an algorithm that has absolutely no warranty to find an optimal solution but that has a "good" chance to find a "good" one. Such algorithms are needed to treat large instances of combinatorial optimization problems, hence sub-optimal solutions instead of exact ones.

2.2 Classical heuristics

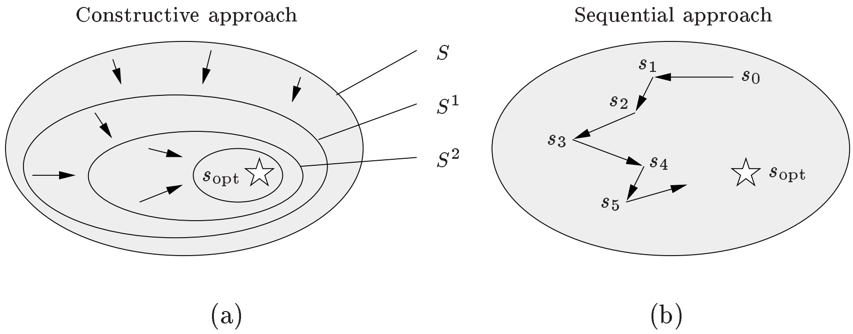


Figure 2.2: *Two traditional search principles for combinatorial optimization:*
(a) A solution is constructed by reducing the search space S such that
$S^i = \{s = (x_1, x_2, \ldots, x_n) \in S | x_1, x_2, \ldots, x_i \text{ are fixed}\},\ i \in \mathbb{N}$.
(b) Elementary modifications are repeatedly applied to a candidate $s_i \in S$.

Three general search principles are known for solving combinatorial optimization problems [32]. The first two are introduced hereafter and the third one is described in the next section.

The constructive approach, schematized in Figure 2.2(a), consists in taking an empty candidate $s_0 = ()$ and constructing a feasible solution by repeatedly choosing its components. A component x_i is added into a partial solution $s_{i-1} = (x_1, \ldots, x_{i-1})$ until a feasible solution s_n is constructed. This process can be seen as an iterative reduction of the search space. A part of a solution implicitly defines a set of solutions (all possible extensions of that given part). It may thus be viewed as a shorthand description of this set. Two examples of constructive approaches are given by greedy-like algorithms: Algorithm 11 (page 91), and Algorithm 12 (page 109).

Such a constructive method can be generalized to algorithms that make use of backtracking (like a branch and bound algorithm). In such algorithms, it is possible to enlarge the search space from time to time. For example, when a partial solution $s_i = (x_1, \ldots, x_i)$ is constructed, it is possible to backtrack to a previous one $s_j = (x_1, \ldots, x_j)$ with $j < i$ and to continue the constructive process from s_j.

The sequential approach (also called iterative approach, local search, or neighborhood search), schematized in Figure 2.2(b), starts with an initial candidate s_0 that can be chosen at random or built by a constructive algorithm. The process consists in repeatedly modifying a candidate s_i. More formally, let us define M_{s_i} the set of acceptable elementary modifications m at a candidate s_i, and $N_{s_i} = \{s' \in S \mid \exists m \in M_{s_i}, s_i \oplus m = s'\}$ the neighborhood of s_i. At each step of the process, the current candidate s_i is transformed into $s_{i+1} \in N_{s_i}$. The process stops when a termination criterion is met. A stop criterion can be one or a combination of conditions such as:

- a given number of modifications have been performed,

- the objective function $f(s_i)$ exceeds a lower bound B,
- a fixed time T_{stop} has passed,
- a local optimum is reached $(\forall s \in N_{s_i}, f(s) \leq f(s_i))$.

An example of the sequential approach is illustrated by the tabu search algorithm [44].

2.3 Evolutionary algorithms

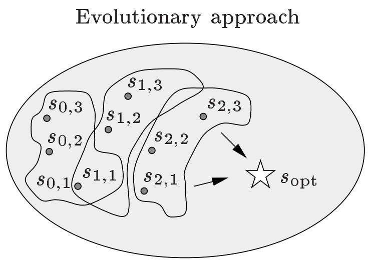


Figure 2.3: *A third search principle for combinatorial optimization: A subset P_{gen} of the search space S evolves in order to find an optimal solution among its members. Each iteration step is called a generation (index by gen):*
$P_{gen} = \{s_{gen,i} \in S; i \in [1, m], m = 3\},\ gen \in \mathbb{N}$

Let us define an *Evolutionary Algorithm (EA)* as a population-based algorithm. This means that its state at any time is a set of candidates (or solutions), in opposition to other algorithms whose state at any time is usually a single solution or a part of the solution that is being constructed (cf. previous Section). A sequential approach might be seen as a degenerate EA. In this case, the set of candidates would contain only one candidate. Similarly, a constructive approach might be seen as a degenerate EA. In this case, the set of candidates would be defined as the set of every possible extension of a partial solution. However, these two degenerate approaches are not considered as evolutionary approaches.

Most of EAs are inspired by biology and natural evolution mechanisms (evolution of species, social evolution of communities, etc.). That is why a specific "biological" vocabulary is commonly admitted in the evolutionary computation community (cf. Appendix A.1). For example, a candidate is called an *individual*, its encoding is called its *genotype* (or its *genome*) and its appearance (or meaning) is called its *phenotype*. The role of an EA is to control the *evolution* of a set of individuals that is called a *population*. During this evolution, individuals are created, modified, added, removed, etc. A *fitness* value is assigned to each individual, indicating how "good" the candidate modeled by the individual is for a given problem instance. The fitness function that computes this value is similar – and even often equal – to the objective function of an optimization problem.

Our definition of EAs is consistent with current publications [32] and with the definition given in the evolutionary computation FAQ [58]: *"EA is an umbrella term used to describe computer-based problem solving systems which use computational models of some of the known mechanisms of evolution as key elements in their design and implementation"*. It is however a little bit more general since the same reference restricts EAs to algorithms that *"share a common conceptual base of simulating the evolution of individual structures via processes of selection, mutation, and reproduction"*. For example, ant systems (that will be described in Section 2.3.4) do not make use of selection, mutation or reproduction operators to solve combinatorial problems. However they simulate the evolution of a population of individuals (called ants) inspired by social rules observed in real ant colonies and they can thus be considered as EAs.

A characteristic of EAs is their ability to visit (or explore) different regions of the search space simultaneously. This *exploration* is usually paired with the *exploitation* of the considered candidates. This counterbalancing concept aims at optimally using (or exploiting) information contained in the candidates. *Diversification* strategies are often used in order to favor exploration and thus to avoid a converging too quickly on the whole population in a same region. The notion of diversification derives from the tabu search literature [44], where it is contrasted with *randomization*: rather than seeking unpredictability, *diversification* seeks to attain an objective that implies a special form of order, such as maximizing a minimum weighted separation from chosen collections of points previously generated. On the other side, *intensification* strategies are used to act towards an improvement of the quality of the candidates. They favor exploitation.

In some algorithms, the whole population models a single candidate of a problem instance. Such algorithms have internal mechanisms similar to those of EAs as defined here: they handle a set of elements that can be viewed as a population of individuals. However, these elements model a part of a candidate instead of a complete candidate and only one single candidate is considered at once. These algorithms are thus fundamentally different and have none of the properties of EAs such as the capability of exploring different parts of the search space simultaneously. Consequently they are not considered in this work. An example of such an algorithm is given by the *emergent colonization* algorithm [70].

The result of an EA is the solution modeled by the individual with the best fitness value found in the population during the whole evolution. This individual is not always present in the population at the end of the evolution process since it does not necessarily participate in the evolution of the population. It must thus be memorized during the evolution. If the best individual is guaranteed to participate in the evolution (and thus to be present in the population at the end) then the EA is said to be *élitist*.

Little differences can sometimes be found here and there in the definition of evolutionary terminology (population, individual, etc.). A glossary of these terms is briefly given in Appendix A.1 and a complete analysis of their meaning is given in Section 3.2.

The first algorithm that was known as an EA is the Genetic Algorithm (GA) introduced by J. Holland in 1975 [63]. The following sections give a brief overview and the pseudo-code of the principal algorithms that are classified as EAs. Each algorithm is described with the terms and notations of its original definition (except for a few details that were adapted for homogeneity reasons).

2.3.1 Genetic algorithm

Genetic Algorithms (GA), introduced by J. Holland [63] in the 70's, are inspired by the genetic mechanisms of natural species evolution. In GAs, four phases can be identified (see Figure 2.4). Genotypes of individuals are generally encoded as chromosome-like

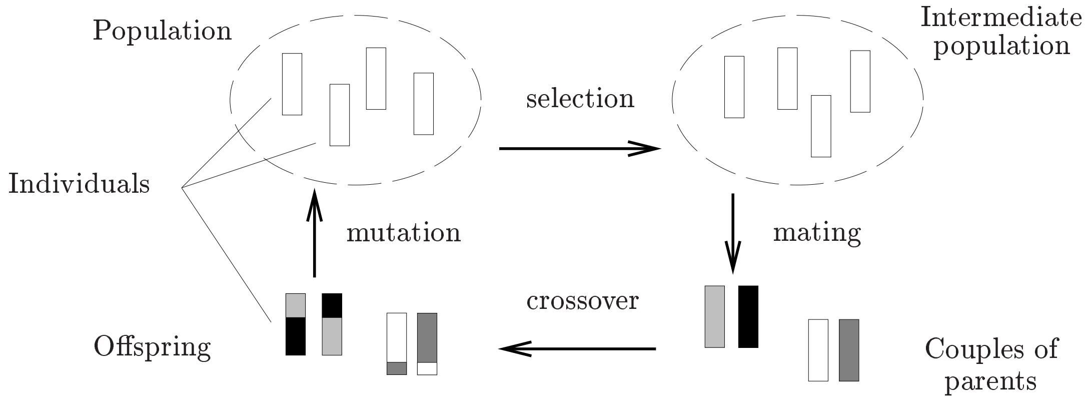


Figure 2.4: *The four phases of a genetic algorithm.*

bit strings. First, an initial population of individuals is generated (usually at random). An intermediate population is then created by selecting individuals according to their relative fitness value (a given individual can be selected several times). This may be perceived as an allocation of reproductive opportunities: the higher the fitness value of an individual, the likelier it is to be selected. When this intermediate population has been filled, it is emptied by taking individuals in pairs out of this population (each individual can be taken only once). On each of these pairs, a crossover operator is applied with a probability of p_c. It consists in exchanging some information between the two genotypes. This operator generates two new individuals by mating the two given ones. For example, the *one-point crossover* cuts two given bit strings at a same random position and recombines them by exchanging their ends, thus producing two new bit strings (or *offspring*). These offspring are put into a new intermediate population. If the crossover operator is not applied (which happens with probability $1-p_c$), the couple of individuals is put directly into the new intermediate population without changes. The selection and crossover operators compose the *cooperation step* of the GA. In the end, a mutation operator introduces noise in this population by randomly modifying some individuals. This second step of the GA, called the *self-adaptation step*, prevents premature convergence of the population. A common way to make this, is to take a mutation operator that flips the value of a randomly chosen bit of a bit string. This operator is then applied with probability p_m to each individual.

The individuals of the intermediate population replace all, or a part of, the individuals in the initial population. In a *generational replacement* GA, the intermediate population has the same size as the initial population, renewing the whole population in one generation. This is not the case in a *steady-state* GA: in such an algorithm, the size of the intermediate population is much smaller than the size of the initial population (for example only one or two couples). Moreover, the offspring do not necessarily replace their parents but can take the place of any other individuals (at random, or among the worst for example). The execution terminates after a predefined number of generations (typically twice the total number of individuals). More detailed descriptions of GAs can be found in [33] and [45]. Algorithm 1 summarizes the standard genetic algorithm described in Chapter 3 of [45].

Algorithm 1
(∗ Standard genetic algorithm ∗)
determine an initial population P at random
generation_count ← 0
repeat
    generation_count ← generation_count +1
    while $P_{\text{intermediate}}$ not full **do**
        select indi1 and indi2 in P
        (offsp1, offsp2) ← crossover(indi1, indi2) with probability p_c
        put offsp1 and offsp2 in $P_{\text{intermediate}}$
    for each individual in $P_{\text{intermediate}}$
        mutate(individual) with probability p_m
    P is updated with the individuals of $P_{\text{intermediate}}$
until termination condition is met

According to Holland [63] the number of *schemata* (i.e., similarity subsets) processed in one generation is proportional to the cube of the population size, hence the need of large population to perform a wide exploration of the search space.

Genetic programming (GP) is an extension of the genetic model into the space of programs (i.e., a candidate of the search space is a computer program). In this context, individuals are programs usually expressed as parse trees, and their fitness value is the result obtained when running this program. More information is available in [69].

2.3.2 Evolution strategy

The development of *Evolution strategies (ES)* started with I. Rechenberg and H-P. Schwefel in the 60's to solve hydro-dynamical problems [86, 3]. However, the first versions of ES were closer to *simulated annealing* than to an EA since they were handling only two individuals. The first ES that was really population-based appeared in the 70's and the (μ, λ)-strategy that is the state-of-the-art was introduced in 1977 [88]. The latter handles a population of μ parents and λ offspring. The offspring are created by combining and mutating parents, and at each generation the μ best offspring replace the parent population. In a variant, the $(\mu + \lambda)$-strategy, the μ best individuals (i.e., among parents and offspring) become the new parents of the next generation. Algorithm 2 summarizes the major components of an ES.

Algorithm 2
(∗ Evolution strategy ∗)
(∗ $Q = P_i$ for a $(\lambda + \mu)$-strategy, and $Q = \emptyset$ for a (λ, μ)-strategy with $\lambda > \mu \geq 1$ ∗)
1. determine an initial set P_0 of size μ
2. $i \leftarrow 0$
3. **repeat**
4. generate P'_i of size λ by combining and mutating individuals of P_i
5. select μ individuals in $P'_i \cup Q$ to put in P_{i+1}
6. $i \leftarrow i + 1$
7. **until** termination condition is met

In ES, an individual consists of up to two components, called *strategy parameters*, in addition to a real valued vectorial genotype $x \in \mathbb{R}^n$. These strategy parameters are variance $\sigma \in \mathbb{R}^{n_\sigma}$ and covariance $\alpha \in [-\pi, \pi]^{n_\alpha}$ of a generalized n-dimensional normal distribution, where $n_\sigma \in \{1, \ldots, n\}$ and $n_\alpha \in \{0, (2n - n_\sigma).(n_\sigma - 1)/2\}$. They determine the mutability of the individuals and can themselves be applied to evolutionary operators (mutation, etc.). The aim is to achieve the *self-adaptation* of parameters and the exploration of the search space simultaneously.

2.3.3 Evolutionary programming

Evolutionary programming (EP) is an EA similar to ES that was developed independently by L. J. Fogel [38] in the 60's. The only differences is the selection mode that is done stochastically via a tournament in EP whereas the worst solutions are removed in a deterministic way in ES. It is also mentioned in [58] that no recombination mechanism is used in EP whereas it can be used in ES.

2.3.4 Ant colony system

Ant System (AS) is a class of algorithms that were inspired by the observation of real ant colonies. Observation shows that a single ant only applies simple rules, has no knowledge and is unable to succeed in anything when it is alone. However, an ant colony benefits from the coordinated interaction of every ant. Its structured behavior (described as a "social life") leads to a cooperation of independent searches with high probability of success. ASs were initially proposed in [25, 28] to solve the well-known NP-hard TSP (cf. Section 2.1.1) that aims at finding the shortest closed tour that passes once by each vertex of a given graph.

A real ant colony is capable of finding the shortest path from a food source to its nest by using *pheromone* information: when walking, each ant deposits a chemical substance called *pheromone* and follows, in probability, a pheromone trail already deposited by previous ants. Assuming that each ant has the same speed the path which ends up with the maximum quantity of pheromone is the shortest.

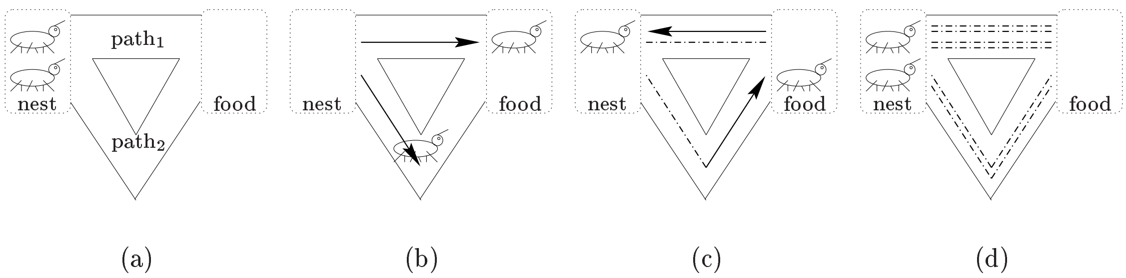


Figure 2.5: *Behavior of an ant colony.*

This process is illustrated in Figure 2.5: at the beginning, ants have no indication on the length of the paths between their nest and the food source, each path is then taken at random by half of the ants in average (Figure 2.5(a)). After one unit of time ants that took path$_1$ have arrived while the others are half-way. They all deposit pheromone trails on their path (Figure 2.5(b)). The first ants are more likely to go back to the nest by their own initial way since no pheromone is deposited at the extremity of path$_2$ yet. After two units of time the first ants are back while the others arrive. The pheromone density is double on path$_1$ (Figure 2.5(c)). From now on, each ant that leaves the net prefers to take path$_1$ that receives thus more and more pheromone (Figure 2.5(d)). Consequently, the shortest path found is path$_1$.

In fact, when an ant must choose a direction to take, the choice is made according to its *visibility* (or knowledge) of the problem, and according to the *trails*. Algorithm 3 is directly inspired by this behavior.

Three different algorithms of the AS class were introduced. They only differ by the quantity of pheromone an ant leaves when it walks on path$_{ij}$, an edge from a node i to a node j. Let us note $\Delta\tau_{ij}^k$ the quantity of pheromone left by an ant k on path$_{ij}$. In the *Ant-quantity* algorithm, $\Delta\tau_{ij}^k$ is a constant. In the *Ant-density* algorithm, $\Delta\tau_{ij}^k$ is inversely proportional to the length of path$_{ij}$. In the *Ant-cycle* algorithm, $\Delta\tau_{ij}^k$ is inversely proportional to the complete tour length done by ant k. In this last case, $\Delta\tau_{ij}^k$ can only be computed once ant k has finished its complete tour, whereas $\Delta\tau_{ij}^k$ is known during its moves in the two former cases. The trail left on path$_{ij}$ by a colony is the normalized sum of the trails left by every ant of the colony on this path during one generation:

$$\Delta\tau_{ij} = \frac{\Delta\tau_{ij}^*}{\sum_l \Delta\tau_{il}^*} \text{ with } \Delta\tau_{ij}^* = \sum_k \Delta\tau_{ij}^k \tag{2.1}$$

Algorithm 3
(* ANT SYSTEM *)
1. initialize the trails
2. cycle $\leftarrow$ 0
3. **repeat**
4. cycle $\leftarrow$ cycle +1
5. **for** each ant
6. construct a solution s_a using trails and visibility
7. evaluate the objective function at s_a
8. update the trails
9. **until** cycle $\geq$ max_cycle

A random-proportional *state transition rule* is used to decide which node an ant must visit from a node i at a time t. The transition probability p_{ij} for an ant to go from node i to node j depends on its visibility $\eta_{ij} = (1/\text{length of path}_{ij})$, and on τ_{ij} the intensity of the pheromone trail:

$$p_{ij}(t) = \frac{[\tau_{ij}(t)]^{\alpha}[\eta_{ij}(t)]^{\beta}}{\sum_{l \in \text{allowed}} [\tau_{il}(t)]^{\alpha}[\eta_{il}(t)]^{\beta}} \text{ with } \tau_{ij}(t+n) = \rho \cdot \tau_{ij}(t) + \Delta\tau_{ij}(t, t+n) \tag{2.2}$$

where $(1-\rho)$ represents the evaporation of trails, α the importance given to the trails, and β the importance given to the visibility. The time unit $(t = 1)$ corresponds to one move of an ant (i.e., $(t = n)$ corresponds to the time needed to complete a tour of length n). It can be noticed that if $\alpha = 0$, ants loose their sense of smell (they do not use trails anymore) and thus the algorithm follows a greedy-like rule. Detailed description and definitions of the different terms can be found in [26, 27].

Ant Colony System (ACS) was introduced later in [34] to improve AS. ACS differs from AS by three main aspects:

- the global updating rule is applied only to edges which belong to the best ant tour,
- a local pheromone updating rule is applied,
- when constructing an ant, the state transition rule provides a way to exploit more or less the accumulated knowledge of the problem. This state transition rule that decides about the node h that an ant must visit from a node i is:

$$h = \begin{cases} \arg\max_{j \in \text{allowed}}\{[\tau_{ij}]^{\alpha}.[\eta_{ij}]^{\beta}\} & \text{if } q \leq q_0 \quad \text{(exploitation)} \\ g & \text{otherwise} \quad \text{(biased exploration)} \end{cases} \tag{2.3}$$

  where q is a random number uniformly distributed in $[0, 1]$, $q_0 \in [0, 1]$ is a parameter, and g is selected according to the probability distribution given in Equation 2.2.

2.3.5 Population-based incremental learning

The *Population-Based Incremental Learning (PBIL)* algorithm is presented in [5] as an abstraction of the standard GA and as a combination of evolutionary optimization and hill-climbing. It is however more similar to an AS than to a GA. It does not use crossover operations and creates a new population at each generation by sampling a probability vector Pr. This vector is updated at each generation with high-evaluation solutions (i.e., individuals with good fitness value) encoded as bit-strings. The main steps of the PBIL together with the formula used to update Pr are shown in Algorithm 4. The objective is to have a probability vector which, when sampled, generates high-evaluation solutions with high probability.

Algorithm 4
(∗ PBIL ALGORITHM ∗)
(∗ $length$ is the size of the vectors Pr and $solution[k], \forall k$. ∗)
(∗ LR is the learning rate. ∗)
(∗ $N_{samples}$ is the number of samples considered at each cycle. ∗)
(∗ N_{update} is the number of solutions to update Pr from. ∗)
initialize the probability vector ($\forall i \in [1, length], Pr[i] = 0.5$)
repeat
    for $k = 1$ **to** $N_{samples}$
        generate $solution[k]$ according to probabilities Pr
        evaluate $solution[k]$
    sort vectors $solution[k]$ from best to worst according to their fitness
    for $j = 1$ **to** N_{update}
        for $i = 1$ **to** $length$
            $Pr[i] \leftarrow Pr[i] * (1 - \mathrm{LR}) + solution[j][i] * \mathrm{LR}$
until termination condition is met

2.3.6 Other evolutionary algorithms

The basic *Scatter Search (SC)* introduced in [43] is a population-based algorithm that is not commonly considered an EA. It has however all the characteristics of an EA and controls the evolution of a population of "points". It can be summarized by Algorithm 5.

Also usually not considered as an EA, the *adaptative memory* algorithm introduced in [87] is based on a population of solutions that is enhanced during an adaptation process. It was initially proposed to solve vehicle routing problems. First, it creates an initial population of constructed solutions. This first step is then followed by a probabilistic diversification and intensification loop. Algorithm 6 gives the main idea of this algorithm.

It could be argued that after the initial step the algorithm is based on a population of "parts of solutions" and not on a population of "solutions". However, since only uninteresting parts of solutions are cancelled while non-trivial (i.e., pertinent) ones are

Algorithm 5
(∗ SCATTER SEARCH ∗)
1. determine an initial set P_0 of points
2. $i \leftarrow 0$
3. **repeat**
4. $i \leftarrow i+1$
5. determine a set T_i of points by linear combinations of points in P_{i-1}
6. transform the points in T_i to get a set F_i of feasible solutions
7. improve the solutions in F_i to get a set D_i of points
8. select $|P_0|$ points in $P_{i-1} \cup D_i$ to form P_i
9. **until** termination condition is met

Algorithm 6
(∗ ADAPTATIVE MEMORY ∗)
1. determine an initial set P of solutions with a local search algorithm
2. **repeat**
3. generate a new solution s by combining parts of solutions of P
4. **if** s is an infeasible solution **then** repair s
5. improve s with a local search algorithm
6. add the non-trivial parts of s in P
7. **until** termination condition is met

kept, it can be assumed that the property of the population is that of a population of "solutions" in which useless information is not encoded. Line 6 could thus be replaced by "add s in P" while line 3 would be "generate a new solution s by combining pertinent parts of solutions of P".

2.3.7 Hybrid approaches

Definition

Many studies have been done to improve the quality of the results obtained with EAs (and especially GAs [102]). One of these techniques consists in making several algorithms work together in order to profit from the best characteristics of each of them. The resulting algorithm is then called a *hybrid algorithm* by analogy with the biological hybridization of two complementary living organisms. In the most general framework EAs can be hybridized with any other algorithm (even with other EAs).

A hybrid algorithm that uses both traditional methods (cf. Section 2.2) and evolutionary techniques is also known as a *memetic algorithm* [78, 79]. This naming comes from R. Dawkin's biological term *meme*: memes are genetic encoding that can be improved by the people that hold them (i.e., individuals evolve their genetic heritage during their life) whereas genes are set once for all at birth (i.e., genetic changes are only possible during the reproduction process).

Taxonomy

Recently, E.-G. Talbi described a taxonomy of hybrid meta-heuristics[2] in [97]. His taxonomy makes a distinction between design and implementation issues.

Design issues are used in order to classify the way meta-heuristics are hybridized. First the taxonomy distinguishes the kind of interactions that associates the meta-heuristics. Second it distinguishes the kind of dependency that links the meta-heuristics. Third it distinguishes if the meta-heuristics are identical (i.e., homogeneous) or not. Fourth it distinguishes if they all explore the same search space or if each of them treats a different part of the problem. And finally, it distinguishes if they all treat the same problem or not. These design issues and their notations are summarized in Table 2.1.

Implementation issues depend on the execution model of the algorithm, that is the machine for which the algorithm was designed (and implemented). They are described at the end of the chapter in Section 2.5.

Examples

When several populations evolve independently they are called *islands* (or *demes* [46]). The independent EAs – one per island – cooperate by exchanging individuals that *migrate*

[2] The term *meta-heuristic* refers to EAs and traditional heuristics that can be applied to different problems, in opposition to heuristics that are designed to solve a specific problem.

Abbreviation	Meaning	Description
L	Low-level	It addresses the functional composition of a single optimization method. A given method of a meta-heuristic is replaced by another method of a meta-heuristic.
H	High-level	The different meta-heuristics are self-contained. There is no direct relationship to the internal workings of a meta-heuristic.
R	Relay	A set of meta-heuristics is applied one after the other, each using the output of the previous as its input, acting like a pipeline.
C	Co-evolutionary	Many parallel agents cooperate. Each agent carries out an independent search and exchange information with the others.
hom	homogeneous	All the combined algorithms use the same meta-heuristic.
het	heterogeneous	Different meta-heuristics are used.
par	partial	The problem is decomposed into sub-problems, each having its own search space.
glo	global	Every algorithm searchs in the whole search space.
spe	specialist	Each algorithm solves a different problem (for example, one can optimize parameters of another).
gen	general	All algorithms solve the same problem.

Table 2.1: *Talbi's classification of hybrid meta-heuristics (design issues) as it is described in [97].*

from one island to another. For example, an island-based GA that was introduced in [98] runs independent GAs on distributed islands positioned on a hypercube. The algorithm is thus a "High-level Co-evolution Hybrid" that executes homogeneous algorithms, each of which solving the same problem in the same search space. It is classified as:

HCH(GA)(hom,glo,gen).

The hybrid algorithm used by D. Levine in his thesis [74] includes three kinds of hybridization. First, it is based on a GA whose population is improved by a local search (LS) algorithm at each generation. The embeded LS algorithm carries out independent searches to improve individuals, hence a classification as a "Low-level Co-evolution Hybrid" that executes heterogeneous algorithms (GA and LS):

LCH(GA(LS))(het,glo,gen).

Second, the initial population of this algorithm is created by a greedy heuristic (GH). The output of the greedy-like heuristic is used as input to the LCH GA and no other interaction occurs. The classification of the resulting “High-level Relay Hybrid ” algorithm is:

HRH(GH+LCH(GA(LS))(het,glo,gen))(het,glo,gen).

Third, this algorithm is run on independent islands (as in the previous example). The classification of the complete hybrid algorithm is thus finally:

HCH(HRH(GH+LCH(GA(LS))(het,glo,gen))(het,glo,gen))(hom,glo,gen).

2.4 Parallel computing

Parallelism may appear at many levels in computers, from the multiple microprocessor registry access to concurent process management. I only consider here multi-processor algorithms (and programs) that are written to execute on several *Processing Elements (PEs)*. One objective of parallelizing an algorithm is to speed up its execution by distributing computation on several PEs. In the case where each PE has a local (or private) memory space, a large distributed memory space is provided. A second objective is to process larger data than it is possible to store on a single sequential computer memory. The simultaneous availability of several cooperating PEs can be a warranty of robustness for fault tolerant systems. This latter property can also be taken as an objective. Parallel computing is a wide field whose fairly complete state of the art can be found in [22, 30, 9, 4]. This section only presents the notions related to parallel computing that are necessary for understanding the next chapters.

2.4.1 Definition of a parallel algorithm

The Arab mathematician al'Khwarizmi (790 – c. 850), who is at the origin of the word *algebra*, wrote a text on Hindu-Arabic numerals. The Latin translation of this text *“Algoritmi de numero Indorum*[3]*”* gave rise to the word *algorithm* deriving from his name in the title. The notion of algorithm evolved during the years and is nowadays often related to computers. An algorithm can have slightly different definitions from one reference to another. The two following definitions are representative of the most complete ones commonly found in the literature. Even if their terms differ, they are equivalent.

- An algorithm is a prescribed set of well-defined rules or instructions for the solution of a problem, such as the performance of a calculation, in a finite number of steps[4].
- An algorithm is a set of explicit, finite, and precisely determined rules, the step-by-step application of which to a complex problem will yield a solution or optimal result[5].

[3]The English translation is “Al-Khwarizmi on the Hindu Art of Reckoning”
[4]Oxford dictionary of computing, 1996.
[5]Cambridge Encyclopedia, 1995.

It is however possible to find more vague definitions that do not approach the termination notion. They simply define an algorithm as:

- A specific procedure for a computer that will either accomplish some tasks or solve a problem. It is roughly equivalent to a computer program[6].

- A procedure or a set of rules for calculation or problem-solving[7].

The notion of termination is sometimes generalized to algorithms such as EAs whose stop criterion depends on the algorithm evolution: "If a potential infinite number of steps is required, the process can still qualify as an algorithm if a stopping rule based on solution accuracy can be given"[8].

The definition of a "parallel algorithm" is also not clear in the literature. It is sometimes missing in some dictionaries and it is often succinct:

- A parallel algorithm is any algorithm in which computation sequences can be performed simultaneously[8].

- A parallel algorithm is an algorithm in which several computations are carried out simultaneously[9].

More formally, a parallel (resp. sequential) algorithm can be defined as a series of instructions that follows a partial (resp. total) order, and that transforms input data (number, files, machine state, etc.) into output data in a finite time [30, 71]. The notion of partial order implies that some instructions may not be ordered and thus could be executed simultaneously. The way the instructions are processed (by a computer, a supercomputer or by hand) is *a priori* not a concern. Any parallel algorithm can be changed into an equivalent sequential one by totally ordering its instructions. It can be noted that such a sequentialized algorithm is then not unique. It can be done by choosing an arbitrary order that satisfies the dependencies of the instructions, or by simulating the time on a given architecture based on theoretical model of parallelism (e.g., PRAM [30] or BSP [101]). Conversely, parallelizing a sequential algorithm into an equivalent parallel one comes to replace some of its ordered instructions by partially ordered ones without changing its behavior. Such a transformation is not straightforward and it is made even more difficult by the search of an "efficient" parallel algorithms for given parallel computer architectures.

2.4.2 Parallel computer architectures

Flynn [37] classified parallel computers accordingly to their control capacit and data flow mechanisms:

[6]Glossary of computing terms, 1998.
[7]The Oxford English dictionary, 1993.
[8]Academic Press dictionary of science and technology, 1992.
[9]McGraw-Hill dictionary of scientific and technical terms, 1994.

- Single Instruction stream, Single Data stream (*SISD*) computers are classical sequential computers.
- Single Instruction stream, Multiple Data stream (*SIMD*) computers execute synchronously the same instruction on every PE simultaneously. Different PEs can however handle different data.
- Multiple Instruction stream, Multiple Data stream (*MIMD*) computers execute different instructions (and even different algorithms) asynchronously on different data.

Parallel computers also divide into two categories depending on their memory architecture.

- *Shared memory (SM) architecture* computers have a unique large memory that can be accessed by every PE. Communication between PEs is done through memory accesses by writing and reading information in the common memory. Memory access problems can be solved at two levels. They can be solved by hardware construction, or by system routines. In the latter case, the architecture is said to be a *shared virtual memory* (SVM).
- *Distributed memory (DM) or message passing architecture* computers have distinct memory units for each PE. Each of these memories can only be accessed by the PE it belongs to. Communication between PEs is done by exchanging messages through channels. These channels are implemented by buses organised according to a given map, called its topology. A regular topology is usual (e.g., a grid, a torus, a hypercube, a ring, a k-ring, a tree, etc.). A router sometimes enhances this basic topology architecture by optimizing direct point-to-point communication links.

Because of the high price of parallel supercomputers and because of the increasing availability of computer networks, workstations linked by a bus (e.g., ethernet, FDDI, etc.) are more and more often used to run parallel programs. A *Network Of Workstations* (NOW) is thus considered as a MIMD-DM computer. A network of homogeneous worstations is rather referred to as a *Cluster Of Workstations* (COW). The topology of such networks (or clusters) is often irregular and hidden. From the programmer's point of view it can be considered as a complete graph. However, the communications are much slower than if a complete graph was physically implemented.

2.4.3 Parallel computing constraints

Problem of defining the speed-up

"It is naively admitted that a task can be done faster by a team than by a single worker. The problem is that a lot of time can be wasted within a team because of waits, chats, and misunderstandings." This simple remark could summarize the problem of speeding

up programs by taking advantage of parallelism. The notion of speed-up is however not easy to define.

Sequential time, noted t_{seq}, is the time required by one processing element (PE) to execute a program that solves a problem instance. *Parallel time* on p PEs, noted t_p, is the time needed to solve the same problem instance with p PEs[10]. It is implicit that the same PEs must be used in both cases to allow any comparison (networks of heterogeneous processors are discussed later). t_{seq} and t_p can be theoretical, measured, or estimated. The *speed-up* evaluates the speed gained by taking advantage of parallelism:

$$\mathrm{S}(p) = \frac{t_{seq}}{t_p} \tag{2.4}$$

The *efficiency* measures the fraction of time a typical PE is effectively used during a parallel execution:

$$\mathrm{E}(p) = \frac{t_{seq}}{p \times t_p} = \frac{\mathrm{S}(p)}{p} \tag{2.5}$$

The concept of speed-up (and of efficiency) has multiple variations: since the optimal sequential time t_{seq} is unknown, it is sometimes defined in a different manner. For example, the following definitions can be found in the literature [9, 30, 22, 4, 57]:

1. t_{seq} and t_p are the execution times of exactly the same parallel program P when running on 1 PE and on p PEs. P is parameterized by p the number of PEs and it implements exactly the same algorithm for every value of p. In other words, $t_{seq} = t_{p=1}$.

2. t_{seq} is the execution time of the best (i.e., fastest) sequential program known while t_p is that of the best program executed on p PEs. The two algorithms are likely to be different in both cases.

3. t_{seq} is the execution time of a benchmark program that is used as a reference (even if faster programs are known). It can be the program the most commonly used during a given period for example.

4. t_{seq} and t_p are the execution times of two programs that implement exactly the same algorithm.

5. t_{seq} is estimated by extrapolation of $t_{p=n}, \ldots, t_{p=m}$ where $n, m \in \mathbb{N}$, $2 \leq n < m$.

The first definition seems to be the most appropriate to study the parallelization of algorithms. It is chosen for the remainder of this reports. The second and third definitions are subject to debate since they require first to elect the best (resp. the most commonly used) program for solving a given problem. Moreover the criteria used to determine this reference program can change from a problem to another. The fourth definition is

[10]Often, no hypothesis is made on the memory space available. In this thesis, it is assumed that the total memory space is p times larger on p PEs than on one PE.

a generalization of the first one. Indeed, it is sometimes impossible to run a program compiled for a parallel computer on a single processor, an equivalent program (often the same code compiled with different compilers or compilation options) must then be used. Sometimes, the input data are too large to be stored in the memory accessible by a single processor. The memory of every PE may be needed to run the parallel application. In this case, the fifth definition makes it possible to draw a speed-up graph anyway.

The number of PEs can sometimes depend on the size n of the problem. In that latter case, the previous definitions are still valid provided that p be replaced by $p(n)$. If the p PEs are heterogeneous t_{seq} is the execution time of the program on the fastest PEs of the heterogeneous network. PEs can be heterogeneous because of technical differences or just because some of them are already used to process other tasks when executing the program (in a multi-user environment, for example).

Since unpredictable external factors (such as operating system processes) can sometimes influence an execution time t_i, it is usually not determined on a single run but it is a statistical result of several runs (minimal or average time after fixed number of runs for example). This is especially true with randomized algorithms as they explicitly make use of a random generator (unpredictable by definition) [81]. EAs are strongly randomized algorithms. Two different executions of the same program with the same input data (i.e., the same instance of a problem) can thus have very different execution times.

In every algorithm, there are parts that are inherently sequential and some others that are parallelizable. Let us note r the ratio of sequential parts on the total computation. The speed-up that can be obtained after parallelizing such an algorithm is bounded by Amdahl's law [1]:

$$\mathrm{S}(p) \leq \frac{1}{r + \frac{1-r}{p}} \leq \frac{1}{r}, \forall p \tag{2.6}$$

The efficiency of a parallel program is also altered by eventual synchronizations of PEs, by the management of the remote PEs (such as their identification by unique labels), and by the time overhead due to communications between PEs. Figure 2.6 shows the shape of a typical speed-up graph.

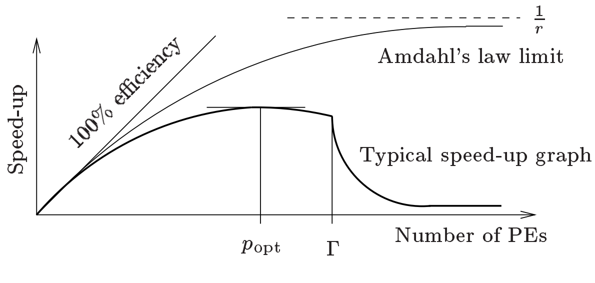


Figure 2.6: *The shape of a typical speed-up graph and of Amdahl's law (see Equation 2.6).* p_{opt} *is the number of PEs that gives the best speed-up,* Γ *is the maximum parallelism degree of the algorithm, and* r *is the ratio of sequential parts on the total computation.*

Communication load

The *communication load* is the part of time due to communications between PEs during the execution of a parallel program. It includes the time needed to prepare and send messages (data or instructions) as well as the latency of the parallel computer to initialize a connection between PEs. The communication load depends on the size and quantity of the exchanged messages (i.e., the number of PEs p and the algorithm structure).

The analysis of communication load in an algorithm requires a model. For example, the linear latency model is often used to evaluate the communication time overhead t_{com}:

$$t_{\mathrm{com}} = \beta + L\tau \tag{2.7}$$

where β is the latency, L is the length of the message and τ is the bandwidth.

When the bandwidth of channels is not large enough to communicate a desired amount of data, a time overhead is added. This phenomenon, that is hard to predict [81], is called *contention*.

Scalability

Roughly speaking, a parallel algorithm has a good *scalability* if it can execute efficiently (with a low communication load) on many PEs. For example, let us define Γ the maximum parallelism degree, that is the maximum number of instructions that can be executed simultaneously in a parallel algorithm. If its speed-up graph is increasingly monotone as long as the number of PEs is less than Γ and if Γ is large enough then the scalability of the algorithm is excellent (see Figure 2.6). More information using accurate models can be found in [57, 55].

Granularity

When a sequential algorithm is parallelized, it is partitioned (implicitly or explicitly) into tasks $T_1, \ldots, T_n$. The number and the size of these tasks define the granularity (or graininess [4]) of the parallel algorithm that is created. A *fine-grain* parallelism approach consists in partitioning the algorithm in a lot of small tasks whereas a *coarse-grain* parallelism approach consists in partitioning it in only a few larger tasks. The choice of the granularity is usually closely tied to the choice of the parallel computer characteristics (architecture, number of PEs, etc.).

Dependency

The tasks of a parallel algorithm are partially ordered. Some of them can therefore be linked by a dependency relationship, that is if task T_j depends on task T_i then task T_j cannot execute before task T_i is finished. This simple and intuitive rule can sometimes lead to efficiency problems. Indeed, when many tasks depend on many others they are likely to waste a lot of time waiting for all the latter to finish.

Task mapping

PEs that run small tasks are likely to finish before those that process large computations. An appropriate distribution of the tasks on the PEs is thus necessary (but not sufficient) to minimize the waiting time of the PEs (i.e., in order to maximize the efficiency of a parallel program in the sense of Definition 2.5). A good *task mapping (or allocation)* is difficult to achieve in practice because of unpredictable factors:

- The computation load of a task is not always known in advance.
- The number of tasks can vary unpredictably during the execution.
- Some PEs can run faster than some others because of their technological properties.
- Some PEs can sometimes be slowed down by tasks that are run by the system or by other users.

Three policies of task allocation are possible depending on when the allocation and the number of tasks are determined. If they are both determined and fixed at compile time by the programmer, the task mapping is *static*. If they can be changed at run-time the allocation is *adaptive*, and if the number of tasks is fixed at compile time while the allocation is changed at run-time, it is *dynamic* [97]. In the two latter cases, a *load balancing* algorithm is necessary to allocate the tasks at run-time. If it is called several times to update the mapping during the execution, it is *dynamic* [30]. Dynamic load balancing is necessary for programs that run tasks of heterogeneous and unpredictable size. In some particular cases a static allocation is sufficient. For example, if a homogeneous cluster of workstations is dedicated to a single user, the task mapping allocates to each PE a task of the same size (i.e., with the same amount of computation to process). The mapping of several tasks on a same PE is time consuming. So if the number of tasks is greater than the number of PEs, tasks can be merged[11] into larger ones in order to speed up the program. This is especially true when the tasks access the same data that can then be loaded only once on each PE.

2.4.4 Classical parallel algorithm models

Parallel algorithms are often the result of the same approaches and thus follow the same schemes. This section presents the classical models necessary to describe most of parallel algorithms. These models are not mutually exclusive and are often merged when designing a single parallel algorithm.

Pipeline model

Historically, the pipeline was used early as a parallel model. It is now part of the VLSI technology and is rather considered as an indispensable technique for speeding up any

[11] This requires that the dependencies between the tasks are satisfied in the resulting task.

part of hardware (or software) than as a parallelization method. It is the application of a simple observation: if an algorithm $\mathcal{A}$ can be expressed as[12] $\mathcal{A}_k \circ \ldots \circ \mathcal{A}_1 \circ \mathcal{A}_0$ then for $i \in [0, k-1]$ the result of $\mathcal{A}_i$ is used as input for $\mathcal{A}_{i+1}$. In that case, if each $\mathcal{A}_i$ is run on a remote PE i then the result available on PE i can be useful for the next PE $(i+1)$ to begin its work. In practice, a pipeline can be represented as a computation chain linked by communication channels that are used to send data flow or control instructions from PE i to PE $(i+1)$ (see Figure 2.7). These channels are not necessarily physical channels but may be simulated via a shared data space (memory or file system) that can be accessed by all PEs.

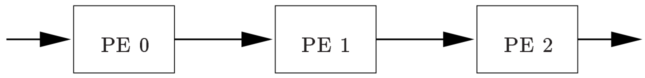


Figure 2.7: *A pipeline approach. Arrows show communications between PEs.*

The aim is that all PEs be busy simultaneously as long as possible. Let us suppose that n data sets must be processed on a pipeline with k stages. Once the k first data sets are loaded in the pipeline, PE i processes data set j while PE $i+1$ processes data set $j-1$. A speed-up of up to

$$\mathrm{S}^{\text{maximum}}_{\text{on pipeline}} = \frac{nk}{k+n-1} \tag{2.8}$$

can be gained. It tends toward k when when n tends to infinity, hence a good efficiency when many data sets are to be processed.

Partitioning model

The partitioning model [85] consists in sharing the computation among PEs, unlike the pipeline model in which PEs assume different duties. Each PE processes a part of the problem that is divided into subproblems. Sub-solutions are combined to produce the final results. Such a model implies thus a minimum of synchronization among PEs.

Asynchronous model

Asynchronous algorithms (also called *relaxed algorithm* [85]) are characterized by the ability of PEs to process the most recent available data without waiting for each other. This is only possible on MIMD computers. The expected efficiency is usually better than that of algorithms that need to synchronize their PEs. A drawback is the complexity of designing and implementing such a model when parallelizing algorithms that are given in synchronized form. It is in fact rarely possible and fully new asynchronous algorithms inspired by given synchronous ones are usually proposed.

[12]Notation: $\mathcal{A}_1 \circ \mathcal{A}_0 = \mathcal{A}_1(\mathcal{A}_0)$.

Farmer/worker model[13]

A classical way to parallelize an algorithm is to give the control of the algorithm to a single PE, called the *farmer*, and to let it distribute the computation among the other PEs, called the *workers*. Figure 2.8 shows such a farmer/worker approach. Since the control and the data processing are separated, this approach is quite easy to implement, and it is robust as long as the farmer PE is not involved in a crash. Moreover, the centralized control of the algorithm makes the task mapping less tricky than in the general framework. The worker PEs can thus be fairly loaded and have a good chance to finish after an equivalent amount of time. However, a *bottleneck* is often difficult to avoid when a lot of worker PEs exchange information simultaneously with the unique farmer PE.

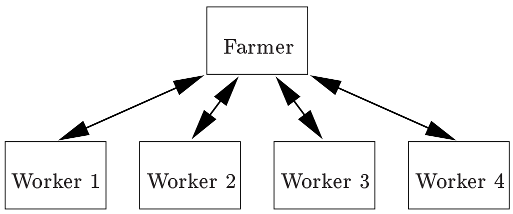


Figure 2.8: *A farmer/worker approach. Arrows show communications between PEs.*

2.5 Classification of parallel EAs

An attempt of classification for parallel EAs was introduced by F. Hoffmeister in [62]. It classifies parallel ESs and GAs into six categories deduced from the following two criteria:

1. The synchronization (named "interaction scheme" in [62]): Synchronous (S) or Asynchronous (A).

2. The parallel model (named "extent of recombination and selection" in [62]): Master/Slave (MS), Parallel Populations (PP), or Parallel Individuals (PI).

It also classifies sequential GAs and ESs that are split into two categories: synchronous and asynchronous[14].

Hoffmeister's classification confounds parallel EAs based on a distributed population with hybrid island-based EAs: the PP category includes all of them. The population of a parallel EA can indeed be distributed on remote PEs, hence a set of distributed *sub-populations* that could be considered as islands. However, this set of sub-populations

[13] This model was originally known as the master/slave model and was recently renamed in order to be politically correct.

[14] The meaning of a sequential asynchronous algorithm is however not explained and seems to be a mistake.

models a single population in the algorithmic sense while the islands of an island-based EA evolve independently and cooperate. The choice of using one population or several islands and the choice of distributing a population for parallelization reasons should thus be distinguished clearly.

In fact, misuses of language are frequent in the literature. For example, the term *Parallel Genetic Algorithm (PGA)* [84, 46] is often used in order to actually refer to Island-based Genetic Algorithm (IGA). The term *distributed genetic algorithm* [7, 99] is also often used to refer to IGA.

The implementation issues described in the Talbi's taxonomy (cf. Section 2.3.7) give the list of characteristics that are required by its classification scheme in order to describe a parallel algorithm: the task mapping (static, dynamic, or adaptive), the computer architecture (SIMD or MIMD), the memory architecture (shared or distributed) and the PE homogeneity (homogeneous or heterogeneous). These issues give thus a concise way to describe the implementation properties of hybrid meta-heuristics (on sequential, parallel or specific computers). They represent the main choices that must be made to give a backbone of a parallel meta-heuristic and are among the first criteria to set when parallelizing an EA.

I am always doing that which I can not do,
in order that I may learn how to do it.
Pablo Picasso, artist (1881–1973)

Chapter 3

Evolutionary algorithm mechanisms

This chapter enumerates the main ingredients that characterize evolutionary algorithms and presents a classification tool based on these ingredients. The use of this tool is illustrated with classical EAs at the end of the chapter.

3.1 The need for a proper parallelization

Section 2.3 showed that EAs are all characterized by the ability to explore several regions of a search space concurrently and that this ability is due to the evolution of several almost independent individuals (also called ants, candidates, solutions, etc.). The number of regions that are simultaneously explored by an EA is tied to the number of its individuals and many individuals are thus usually required to achieve a "good" (i.e., wide enough) exploration. Since the management of many individuals is highly time and memory consuming, and since the amount of independent processing required for the evolution of individuals suggests an intrinsic parallelism, parallel versions of EAs are of great interest. However EAs cannot all be efficiently parallelized in the same way because each of them uses its own mechanisms, that is why the study of the parallelization of EAs appears to be an appealing challenge. The goal of the next section is to identify the mechanisms of EAs in order to give specific rules for their parallelization.

3.2 An original taxonomy of EAs[1]

3.2.1 Motivation for parallelization

The multitude of different algorithms classified as EAs and their sometimes unclear definitions prevent us from giving any general parallelization rule and it is thus necessary to identify the fundamental ingredients of such algorithms.

[1]The content of this section is a joint work published in [12] and [67] with little modifications. The first version of this taxonomy was presented at *ismp97* [68].

This section presents an attempt to classify EAs. It shows how they can be described in a concise, yet comprehensive and accurate way. First, the fundamental ingredients of EAs are identified and explained. Second, these ingredients are interpreted at different abstraction levels. Then, a new classification scheme relying on a table, called the *Table of Evolutionary Algorithms* (TEA), is introduced. It distinguishes between different classes of EAs by enumerating their fundamental ingredients. Finally, possible uses of the TEA are illustrated on typical EAs.

3.2.2 Background

At the beginning of EA history, there was no ambiguity about what GAs were [63]. Later, however, different ingredients were added to enhance GAs' performances, leading to algorithms which substantially differ from their original principles [33, 45]. These algorithms are still often named GAs. Moreover, it is common to find a GA described with the same pseudo-code as an EA in the literature [58]. Although the difference between these two classes of algorithms is usually explained, it seems that the distinction is often not clear. One of the risks such a situation leads to is that identical things are made several times under different names. This is the reason why, despite the existing *ad-hoc* tutorials, a systematic means of describing the main ingredients of EAs in a short-hand way is a challenging task to investigate.

An example of the above mentioned risk is given by scatter search and GAs. As explained in [43], a number of evolutions of GAs used for solving optimization problems were basic elements of the initial scatter search framework, and have therefore been "rediscovered".

The literature very often focuses on the efficiency, or the utility, of some kind of operators. For example, many articles compare the use of different crossover operators in a GA (classical one-point crossover, multi-point crossover [92], uniform crossover [96], etc.). The important points concerning the structure of the algorithms are then too often ignored. This could be schematized by saying that the interest is more in the implementation of the EAs than in the mechanisms of the algorithms themselves. For example, in order to understand the "philosophy" of an algorithm, the fact of using or not using a mutation operator in a GA may appear more important than the way it is actually done in a particular implementation.

3.2.3 Main ingredients of an EA

Population

The population being the primary source of the exploration mechanism in EAs, its size can be viewed as a measurement of exploration capacity. The population size is thus an important ingredient of an EA that can either be constant or change during the evolution. Individuals that exchange information in an EA are called the *parents* and newly created or modified individuals are called the *offspring*. The exchange of information is realized by operators that usually depend on the considered EA. For example in GA, this is done by the crossover operator (cf. Section 2.3.1). Offspring are created using information coming from several individuals present in the current population. The *number of parents* participating in the creation of an offspring is an important ingredient of an EA since it defines how much information is merged at once. For example, in GAs the number of parents is constant and equal to 2, but there exist other EAs in which the number of parents can vary during the execution of the algorithm.

In addition to the parents, the creation of the offspring may also use some global information about the history of the population. This information represents the context in which the algorithm evolves, and is called *history of the population.* This context is generally handled by the population, and is updated with respect to the past of the population. The term *history of the population* includes information, taking into account the evolution, that cannot be gathered by looking at the current state of the population, but would need the historical account of the last generations. An example for this is the trails handled in ASs. Another example would be given by GAs in which the probability that is associated with each operator would be updated by taking into account the results obtained during the last couple of generations.

The frequency of use of the information sources (called *exchange rate*) is also an important feature since it determines the amount of information exchanged on average. For example, if a little information is exchanged often or if a lot is exchanged rarely, the global exchange of information is in the same range.

The *information sources* of an offspring are described by three ingredients, the number of parents, the history of the population (noted $h_{\text{Population}}$), and the exchange rate.

Neighborhood

An important feature concerning the exchange of information between individuals is the limitation of the number of individuals which are allowed to make exchanges. To answer that question, a neighborhood function[2] $\mathcal{N} : P \to \mathcal{P}(P)$ can be set for the population P. The neighborhood function associates to each individual e a subset $\mathcal{N}(e)$ of P called its neighborhood.

An operator that is applied to an individual $e \in P$ can then only choose another individual taken from $\mathcal{N}(e)$. This neighborhood function can take the form of a directed graph: a vertex is associated with each individual and an arc from an individual i_1 to an individual i_2 is introduced if $i_2 \in \mathcal{N}(i_1)$. In some case individuals are not aware of each other (i.e., they do not exchange information explicitly) P is then *unstructured.* In every other case, a population can be viewed as a connected set of individuals that defines a *structured space* (i.e., a topology). If the operator can be applied to any combination of individuals (as in the classical GA) then the structure of the space is a complete graph: $\forall e \in P, \mathcal{N}(e) \cup \{e\} = P$.

If an operator is applied to more than two individuals at once, the same neighborhood

[2] $\mathcal{P}(P)$ is the set of the subsets of P.

function can be used. The only supplementary requirement is a given order in which these individuals must be chosen. For example: whether they must all be in the neighborhood of the first one, or each one must be in the neighborhood of the last individual that was chosen, or any other rule.

Individual history

As explained at the beginning of this section (page 33), information about the history of the population may be stored during the run of the algorithm. A similar information may exist at the individual level: each individual has some evolving information that does not concern the problem being solved but how the individual behaves in a certain situation (what is its mutation rate for example). This information is the history at an individual level, called *history of the individual*, and is noted $h_{\text{Individual}}$. Here again, the history covers information that cannot be determined by the current state of the individual, but would need a description of its state during the previous generations. Notice that this notion also applies to a generational replacement algorithm. Indeed, the history of a newly created individual is then defined on the basis of the history of its parents. An example of such a history is given by ES (cf. Section 2.3.2).

Evolution of a population

Usually, the evolution of the population is achieved through a succession of evolution steps called *generations*. If the whole population can be changed from one generation to another[3], an evolution step is said to be a *generational replacement*. If only a part of the population is changed from one generation to another, the evolution is said to be *steady state*. With the use of parallel programming for the EAs, *asynchronous* evolution has appeared. In this last case, each individual is continuously changed without checking whether the others are also changed. For example in an asynchronous GA, the next individual that is to be replaced by an offspring can still be used between the beginning and the end of the creation of the offspring.

Solution encoding

There exist many ways to encode individuals. Even if chromosome-like strings are often used, the encoding method is mainly determined by the information that should be treated (i.e., exchanged, improved, modified, etc.) by individuals. The kind of information that is exchanged is on a higher level, whereas the way to do it (a crossover operator description, for example) may be considered on a lower level. The information exchange operator, and its coding, will be determined on the basis of the kind of information that has to be exchanged. However the kind of information to exchange in order to have an efficient EA depends on the problem considered. Indeed, once one has decided what information is important to exchange, the basic blocks of information, one can choose

[3] With the possible exception of one or two individuals of the population.

any encoding method and design the information exchange operator with respect to this encoding and the information blocks. Of course, the choice of an encoding together with an operator that exchanges information may be very inefficient on a computer, but the choice of this pair is only an implementation issue. Since encoding is highly problem-dependent[4], this feature will not be included in this general descriptive TEA.

For example, let us take a real valued vector v that represents a colored graph ($\forall i, c[i]$ represents the color index of vertex i), and let us take an individual *indiv* whose genotype is encoded by v. Let us define now a mutation operator $\mathcal{M}$ that changes the value of a vector component at random. The fact that the color of a vertex of the colored graph represented by *indiv* is changed by mutating it with $\mathcal{M}$ must not be seen as a consequence of the vectorial encoding. It is in fact an algorithmic choice: an appropriate operator $\mathcal{M}'$ can be designed to have the same effect on any individual whose genotype represents a colored graph whatever the encoding is.

Nevertheless, the encoding method has an impact on the behavior of an algorithm. It is important to describe these effects. Depending on the encoding method and the information exchange mechanism, newly created or modified individuals can represent infeasible solutions. An infeasible solution is a candidate that is not a solution to the considered problem. It should be noticed that some EAs can, by using convenient encoding and information exchange, avoid the creation of infeasible solutions. Individuals representing infeasible solutions can be *killed, repaired,* or *penalized.* Individuals are killed when they are deleted, or replaced by other new individuals. Individuals are repaired when they are transformed so as to represent a solution to the problem (no improvement of the solution is expected by the transformation, only its feasibility is concerned). When an individual is penalized, its fitness value acquires a penalty that can depend on the distance between the candidate represented and a feasible solution.

Individual improving

A way to bring significant improvements in the results obtained by an EA is to use local heuristic techniques such as hill-climbing or tabu search at some stage of the computation [42]. An *improving algorithm* is any change applied to a single individual, without using information of other individuals, in order to improve its fitness value. The improving algorithm can be a simple operation or a more sophisticated combinatorial algorithm (e.g., tabu search, simulated annealing). In the latter case the global algorithm is said to be hybrid.

Noise

One of the major problems encountered with combinatorial algorithms is the premature convergence of the solution towards a local optimum. In order to steer individuals away from local optima or some more complex regions of attraction, EAs introduce some *noise* (or randomization) in the population. This noise can be generated by randomly

[4] But it could be introduced in a problem-specific table for EAs.

perturbing some individuals, as the mutation operator does in a GA for example. The only requirement is that this noise has unexpected results on the fitness of an individual, in the sense that it does not necessarily improve it.

3.2.4 The basic TEA

In order to be aware of the principles of an EA compared to another EA, the ingredients that characterize them must be easily readable. The creation of a one-row table that allows such comparisons is therefore proposed: the TEA (Table of Evolutionary Algorithms). The main idea of the TEA is to have one column per ingredient developed in this section. In each cell, an entry, that can be a number or abbreviated information, gives the necessary indication for the corresponding criteria. Table 3.1 shows such a table whose cells are filled as follows:

$\lvert$Population$\rvert = $ cst	structured population	information sources	infeasible	$h_{\text{Individual}}$	improving algorithm	noise	evolution
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)

Table 3.1: The basic TEA

(1): A 'Yes' or a 'No', depending whether the size of the population is constant. A size range can be written instead of the 'Yes', if it is constant.

(2): A 'Yes' or a 'No', depending whether the population is structured. Classical topologies can be written instead of the 'Yes': *ring, grid, torus, hcube* (hypercube) and *compl* (complete graph).

(3): The number of parents for each offspring (nothing if this number is not fixed). The abbreviation $h_{\text{Population}}$ can be added if the history of the population is used. If the information is not entirely exchanged at each generation between every parent, an exchange rate can be added into brackets: for example, (0.5) means that individuals exchange information every two generations, or that only half of them exchange information at each generation in average, or even that they exchange only half of their information.

(4): One of the four abbreviations: *nvr* (when infeasible individuals can *never* appear), *pen* (when the infeasible individuals are *penalized*), *rep* (when the infeasible individuals are *repaired*) or *die* (when the infeasible individuals are killed).

(5): A 'Yes' or a 'No', depending if the history of the individuals is used by the algorithm.

(6): A 'Yes' or a 'No', depending if an improving algorithm is applied to the individuals or not.

(7): A 'Yes' or a 'No', depending if noise is used or not.

(8): One of the three abbreviations: *gr* (when generational replacement is used), *ss* (when steady state is used) or *as* (when asynchronous mode is used).

The TEA does not provide, nor does it replace, algorithm pseudo-codes. It merely informs about the algorithm's key elements. The primary goal of the TEA is to compare the principles of EAs, as opposed to comparing their performances. It should never be forgotten that its aim is not to explain the details of a given algorithm. It may however be used to describe algorithm classes, or to compare the characteristics of two algorithms *a priori* considered to be different.

For a first example, the very simple genetic algorithm described by Algorithm 1 at page 12 is used. The basic TEA associated with this algorithm is shown in Table 3.2.

$\|Population\| = cst$	structured population	information sources	infeasible	$h_{Individual}$	improving algorithm	noise	evolution
Yes	compl	$2(p_c)$	nvr	No	No	Yes	gr

Table 3.2: The basic TEA for a standard genetic algorithm

3.2.5 Hierarchical ingredients

Further description levels

Ingredients that were described in the previous section concern one population of individuals. Some other description levels may however be considered, in order to describe several populations, or even sets of populations. This section shows how the notions explained in 3.2.3 can be understood at other description levels.

Usually, the notion of parents is only used when a new individual is created by combining information of other individuals. However, this notion can be generalized to any exchange of information. For example, consider the case where several populations (or islands) are used. An island obtained by selecting a collection of individuals from two islands I_1 and I_2 can be considered as the offspring of the parent islands I_1 and I_2.

In fact, most of the ingredients of the previous section remain correct if "individual" is replaced by "island". With this in mind, the previous section can describe another level of an algorithm using islands. In order to look at an EA in this manner, one must define an element e and a set S of such elements at each level. At a given level, the EA works on a homogeneous set of elements e. Let us take an island-based GA (IGA) as example (cf. Section 2.3.7). An IGA inspired by the standard GA described in Algorithm 1 is given by Algorithm 7. The islands are virtually positioned on an oriented ring, and migrations are only allowed along that ring. Every time a new generation is computed, a copy of the best individual (i.e., with the greatest fitness value) ever met by each island is sent to the next island on the ring. Each island thus receives a new individual that replaces one of its individuals selected randomly (another policy would be to replace the individual with the lowest fitness value).

Algorithm 7
(* ISLAND-BASED GENETIC ALGORITHM (IGA) *)
determine k initial islands $[P^0, \ldots, P^{k-1}]$
generation_count $\leftarrow$ 0
repeat
    generation_count $\leftarrow$ generation_count +1
    for each island i
        while $P^i_{\text{intermediate}}$ not full **do**
            select indi1 and indi2 in P^i
            (offsp1, offsp2) $\leftarrow$ crossover(indi1, indi2)
            put offsp1 and offsp2 in $P^i_{\text{intermediate}}$
        mutate each individual in $P^i_{\text{intermediate}}$
        $P^i \leftarrow P^i_{\text{intermediate}}$
    if generation_count is multiple of m
        then the best individual of each island P^i migrates to $P^{(i+1) mod\ k}$
until termination condition is met

At the lowest level, in such an algorithm an element e is an individual. Individuals are grouped to form islands (the sets S). Since the IGA works on each island independently, let us consider a level where an element e is an island (the set S of the previous level) and where these islands are grouped to form an *archipelago* (the new set S). In the case of the IGA, only one archipelago is considered, but a process that clusters archipelagi into *meta-archipelagi* could be imagined. In such a case, the same reasoning as for the previous level can be applied. Therefore, the previous section remains valid almost without modification for all levels. Figure 3.1 gives an example of the use of structured spaces for a population and for an archipelago. In the IGA case, the information exchange operator, corresponding to the crossover operator at the individual level, can be migration at the island level, as mentioned above. A definition for the "fitness value" of an island can be the mean fitness value of the individuals in this island. Thus, an improving algorithm can improve this mean fitness value (without using the other islands). The only ingredient of the previous section that cannot be easily generalized to an upper level, is the feasibility of an element: it is not clear what an infeasible island can be. But the possibility is left for a suitable definition needed in future developed EAs.

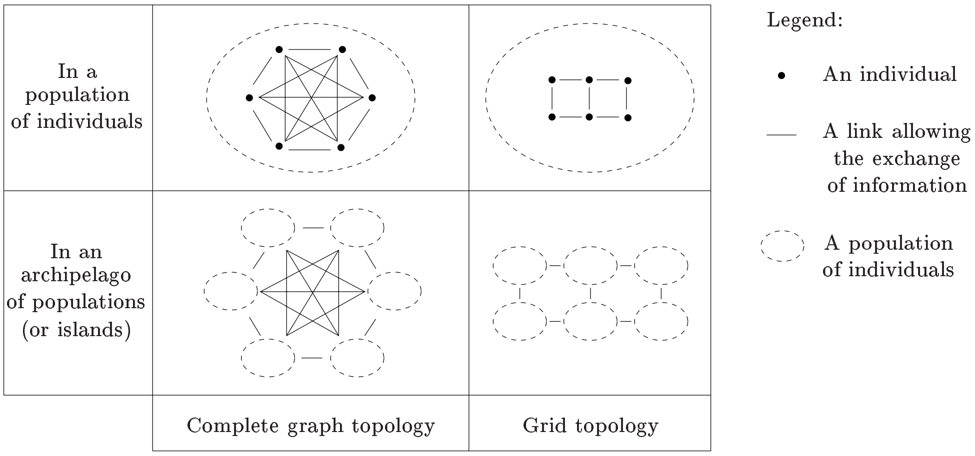


Figure 3.1: *Structured spaces defined by individuals in a population and by islands in an archipelago. In this example, a complete graph and a grid topology are chosen to structure the space.*

Even if they do not exist yet, EAs can be imagined with even more levels. The IGA can for example be extended to a three-level algorithm, an archipelago-model GA, in which there are several archipelagi. Algorithms with more than two levels do not necessarily give better results, but they can enter in the above described framework.

Since, as explained in 3.2.3, an individual represents a candidate to the problem instance considered, the individual level can be seen as the basic level of an EA. The level in which an element is a set of individuals can be seen as a level above this basic level. More generally, if the elements e of a given level l are more elaborate than the elements e' of a level l', the level l can be considered as being higher than l'. If the elements $e' \in S'$ are direct components of the elements e, then $e = S'$ and $l' = l + 1$.

The classification table

Let us see now the complete classification table, based on the basic TEA introduced in 3.2.4. In this extended table, one row is filled per description level. An additionnal column is inserted on the left side in order to name the description level of each row. Table 3.3 shows such a table with two description levels.

$S(e)$ Set of elements e	$\lvert S\rvert = \text{cst}$	structured S	information sources	infeasible e	h_e	improving algorithm	noise	evolution
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)

Table 3.3: TEA: the table for evolutionary algorithm classification

Column (0) names a description level by making explicit the set S of elements e concerned in the corresponding row. For example, at the lower level in a IGA, "Island(Individual)" would mean that in the first row the elements e are the "Individuals" grouped into "Islands".

The TEA is filled as explained in 3.2.4 for the basic TEA, except that sets and elements are now considered instead of populations and individuals. For example, columns (2), (3), and (5) are now filled with:

(2) A 'Yes' or a 'No', depending if the set S is structured or not. Classical topologies can be written instead of the 'Yes': *ring, grid, torus, hcube* (hypercube), and *compl* (complete graph).

(3) The number of parents for each offspring (nothing if this number is not fixed). The abbreviation h_S is added to the number if the history of the set S is used.

(5) A 'Yes' or a 'No', depending if the history h_e of the elements e is used by the algorithm.

In the case the corresponding ingredient has no sense at a given level, a cell contains the character '/'. It should be rare.

In standard EAs, the one-to-one relation holds between the levels of the algorithm and the rows of the table. But the TEA is more flexible: several rows are possible for a given level. For example, two different types of islands can be used and can be described by a row named "Island1(Individual)" and a row named "Island2(Individual)", grouped with "Archipelago(Island1, Island2)".

In order to improve the results, algorithms often use some kind of diversification in one of the ingredients. For example, one can imagine that the size of the population is constant most of the time, but that it is decreased from time to time and then brought back to its original value. If taken literally, one should put a 'No' in the column entitled "$|S| = \text{cst}$". But since, the overall idea is to have a constant population, a special symbol Δ (for Diversification) may be associated with a 'Yes' in this column. Table 3.4 shows how to describe a population whose size is decreased every now and then. How exactly the diversification is done can be commented beside the table.

$S(e)$ Set of elements e	$\lvert S\rvert = \text{cst}$	…
Population(Individual)	Yes Δ	…

Δ The population size changes when…

Table 3.4: *Example of the use of the* Δ*-feature.*

A large part of the place taken by the table is due to the labeling of the columns. Thus, a condensed version of the table was introduced here. This short version does not contain the labels. Each row of the table can then be represented by its ingredient pattern, that can be seen as its "finger prints":

$$(0)\ [(1)(2)](3)(4)(5)[(6)(7)(8)]$$

where (i) is the content of cell (i). To keep an easy reading of this compact version, the Yes's and the No's are replaced by capital Y and N. The empty character that can be put in cell (3) when the number of parents is not fixed is replaced by a '␣'. The abbreviations appearing in the cells (2),(4) and (8) are kept in lowercase. If there is a Δ in a cell, it can be put as index to the corresponding Y, N or abbreviation in this compact version. It can be noted that the first two cells (in the first pair of square brackets) concern directly the sets S, the next three cells concern directly the elements e, and the last three cells (between the square brackets) are related to the evolution "policy".

3.2.6 Examples

TEA descriptions associated to typical EAs are given as examples in this section.

Standard genetic algorithm

Let us again use Algorithm 1 whose TEA was presented in 3.2.4, but consider the TEA in its final shape. The TEA associated with this standard genetic algorithm is shown in Table 3.5, and its compact form is: Population(Individual) [Ycompl]2(p_c)nvrN[NYgr].

Island-based genetic algorithm

The second example is a generational replacement island-based GA, that uses migration on an oriented ring every m generations (see Algorithm 7 at page 38). If the fitness of an island is defined as the mean fitness of the individuals in this island, then the TEA associated with this algorithm is shown in Table 3.6. The Information sources noted "$2(1/m \cdot 1/|\text{Island}|)$" means that 2 parents (i.e., 2 islands) exchange information every m

$S(e)$ Set of elements e	$\|S\| = \text{cst}$	structured S	information sources	infeasible e	h_e	improving algorithm	noise	evolution
Population(Individual)	Yes	compl	$2(p_c)$	nvr	No	No	Yes	gr

Table 3.5: TEA of a standard genetic algorithm

$S(e)$ Set of elements e	$\|S\| = \text{cst}$	structured S	information sources	infeasible e	h_e	improving algorithm	noise	evolution
Island(Individual)	Yes	compl	$2(p_c)$	nvr	No	No	Yes	gr
Archipelago(Island)	Yes	ring	$2(1/m \cdot 1/\|\text{Island}\|)$	/	No	No	No	gr

Table 3.6: TEA of an island-based genetic algorithm

generations and that the amount of information exchanged (compared to the size of an entire island) is $1/|Island|$ (i.e., one individual).
The compact form of this TEA is:

$$\left| \begin{array}{l} \text{Island(Individual) [Ycompl]}2(p_c)\text{nvrN[NYgr]}, \\ \text{Archipelago(Island) [Yring]}2(1/m \cdot 1/|\text{Island}|)\text{/N[NNgr]}. \end{array} \right.$$

Scatter search

The third example is the basic scatter search [43] that is summarized by Algorithm 5 shown on page 17. The TEA associated with this algorithm is shown in Table 3.7.

$S(e)$ Set of elements e	$\|S\| = \text{cst}$	structured S	information sources	infeasible e	h_e	improving algorithm	noise	evolution
Population(Point)	Yes	compl	$2..\|S\|$	rep	No	Yes	No	ss

Table 3.7: TEA of a basic scatter search

Ant system

The fourth example is an ant system. Algorithm 3 (page 15) gives a sketch of such an algorithm. The corresponding TEA is shown in Table 3.8. Notice that the only information source for an ant is the history of the population (called trails in ant systems). Indeed, during the construction of a solution, an ant does not use the solutions provided by some given ants, but uses exclusively a combination of values obtained by the whole population during a certain number of cycles.

$S(e)$ Set of elements e	$\lvert S\rvert$ = cst	structured S	information sources	infeasible e	h_e	improving algorithm	noise	evolution
Colony(Ant)	Yes	No	0 h_S	nvr	No	No	No	gr

Table 3.8: TEA of an ant system

PBIL

The last example is the Population-Based Incremental Learning (PBIL). Its TEA shown in Table 3.9 is the same as that of an ant system. The two algorithms are indeed based on the same concepts. The only differences between them are:

- the possibility for ants to use *visibility* (i.e., specific information about the problem),
- the obligation for solutions in a PBIL to be encoded as bit-strings,
- the size of the probability vector P (τ in an AS) that must have the same size than the solution vectors in a PBIL,
- the computation of P (or τ) that needs only the best solutions in a PBIL (and all ants in an AS).

PBIL could be seen as a particular case of AS. Indeed, an AS with the following constraints works as a PBIL:

- ants are encoded into bit-strings,
- visibility is not used ($\beta = 0$),
- the function that updates τ does not take every ant into account.

This justifies the fact that PBIL and AS have the same TEA description.

$S(e)$ Set of elements e	$\|S\| = \text{cst}$	structured S	information sources	infeasible e	h_e	improving algorithm	noise	evolution
Population(solution)	Yes	No	0 h_S	nvr	No	No	No	gr

Table 3.9: TEA of a PBIL

3.2.7 Extensibility

The TEA has been designed in order to be extensible to yet unknown classes of EAs. It makes explicit the hierarchical notion of "level" in EAs; therefore, the TEA can also be helpful for the design of new classes of EAs. Indeed, by exploring many different ways to fill out the TEA, one may discover new classes of EAs by their descriptive characterization. In the end it should be stressed that the TEA is not a static tool. In concert with the possible evolution of EAs, the TEA may evolve in order to take into account new ingredients that will eventually be discovered as useful in EAs.

3.3 About islands and topology

Studies of parallel EAs often tackle the notion of islands and the notion of structured space (introduced in 3.2.3). The reason is that these two notions appeared with the first studies of the parallelization of GAs [47, 98, 19, 20]. This section discusses them from an algorithmic point of view by removing every reference to parallel implementations (the relationship between these notions and parallel EAs is presented in next chapter).

3.3.1 Structured space phenomenon

As defined on page 33, the elements of an EA can be connected in order to restrict their exchange of information within a given neighborhood. The topology hence defined influences the behavior of the algorithm since it controls the information propagation speed. The effects of a grid topology on a GA are observed in [93]. It is shown that species (i.e., individuals with identical genotypes) spontaneously appear in some regions to form small clusters. These clusters grow or shrink during the evolution according to exchange of information (i.e., the crossover) at the edges. An increase of the neighborhood size results in a higher *selection pressure*[5] (i.e., a faster convergence of the algorithm).

[5]When a phase of an EA requires that individuals be selected, the importance of the restrictions on this selection phase is measured by the *selection pressure*. If the choice is large, the pressure is low and the exploration of the search space is thus favored (cf. Section 2.3). If the choice is restricted, the pressure is high and the exploitation of the search space is then favored.

3.3.2 Island phenomenon

It is observed that the execution of several independent GAs on islands (cf. Section 2.3.7) gives better results than that of one GA on a single population with the same total number of individuals [99, 94, 16]. It is also observed that an island-based GA with migration outperforms one without migration [23].

The role of migration is to exchange information from one island to another. It has been shown that it is equivalent to the GA crossover operator at the abstraction level of the population. The migration rate (i.e., the quantity of individuals that migrate at once) and migration time-scale (i.e., the frequency of migration) determine the amount of information exchanged between islands. If this amount is too large or too small the performance is degraded (as observed in [99]).

A decrease of the population size results in a higher selection pressure (i.e., a faster convergence of the algorithm). Islands permit independent evolution of several populations, hence a simultaneous *almost independent* exploration and exploitation of the search space. A first theoretical investigation of allocation of trials to schemata by PGA (in fact island-based GA) is given in [84]. It is however not clear why an IGA evolves towards increasingly better fitness values when the number of islands increases. A study of the optimal number and size of islands is made in [46] for some specific cases (isolated islands, perfect mixing of the information produced on each islands, etc.). According to [19] that refers to different papers, the interesting topologies are those with a medium diameter (the ring is however presented as a good candidate in spite of its rather large $\frac{length}{2}$ diameter).

3.3.3 Discussion

The island and the topology phenomenon is not clearly explained but some ideas can be proposed. The study of the space structure phenomenon and of the island phenomenon are closely tied. Indeed, they both increase the selection pressure by isolating individuals in a neighborhood that is opened in the first case and closed in the second.

When an island is small, its convergence is fast and once all of its individuals are almost identical the behavior of the EA on the island is mainly similar to that of a traditional sequential algorithm (cf. page 8) that only searches in the neighborhood of a candidate. An island-based EA could thus be compared in a first approximation to a simultaneous execution of sequential algorithms.

Let us take now a structured space of islands that contain unstructured populations. If the number of islands is increased while the total number of individuals is kept constant, the size of each island decreases. In the limit case, each island has one individual that can only exchange information with the individuals on neighbor islands. It results in a structured space of individuals that is mapped on the topology that connects islands. This shows that the study of topology and of islands is tied.

A grouping concept whose function operates a bit differently from the function of islands derives from creating clusters of individuals that share particular features [42]. This gives rise to strategies where “within cluster” operations yield forms of intensification, while “across cluster” operations yield forms of diversification. The use of such *clustering* can be described in the TEA in the *structured S* column since it can be assimilated to a neighborhood function whose topology changes with time.

These considerations could have an important impact on the characteristics of EAs designed in the future. However, so far, there is no proof on the properties of islands and structured spaces. The island model has however some advantages on the structured space model: it is less sensitive to parameter settings since it enables different islands to have different parameters [99] (or even different EAs).

3.4 An island-based genetic ant algorithm

The aim of this section is to create a new hybrid EA that will be used in the remainder. The aim is to experiment some parallelization rules on a new EA on which there is no *a priori* knowledge.

3.4.1 Motivation

GAs are rarely trapped in local optimum but they are sometimes too “blind” to find good regions of the search space even when they seem easy to find. GAs are thus often hybridized[6] with traditional heuristics in order to outperform the standard GA (dozens of them are listed in [97]). However these traditional heuristics can attract the whole algorithm in a bad region of the search space.

In an AS, individuals (or ants) are constructed by using “pure” traditional techniques (the visibility of ants) and evolutionary knowledge (the trails) as a source of information. When the visibility of ants is given a high importance[7], the behavior of an AS is thus similar to that of a local search but without its drawbacks since it benefits from EA properties.

The exchange of information from one individual to another is one of the main mechanism in GAs while the global information about the population that is gathered and used by every individual is the heart of ASs.

The use of both mechanisms in a unique EA could be fruitful since such a hybrid EA could have the efficiency of a hybrid (greedy, GA) without being attracted in a bad region of the search space. In a hybrid (greedy, GA), the greedy-like algorithm can be used to determine the initial population of the GA, to improve its final population, or to improve its intermediate population by modifying (or replacing) some individuals at each generation. All these approaches result in a low-level hybridization. As mentioned in the previous section the use of islands (a high-level co-evolution technique) often results in EAs with good performance. This hybridization technique is thus a good alternative to

[6] Cf. definition in 2.3.7.

[7] This is done by setting a higher value to β than to α (cf. Section 2.3.4).

hybridize a GA and an AS. Moreover, the resulting hybrid EA would be a good test-candidate to study the parallelization of an atypical EA because of the many different EA ingredients[8] it would use.

3.4.2 Description

The *genetic ant algorithm* proposed here is an island-based EA with heterogeneous islands, named IGAA. It introduces no new notions but uses those of EAs discussed previously. Each island is independent and has its own population that evolves according to its own rules: one island contains individuals that evolve as ants in an AS (cf. Algorithm 3) and other islands contain individuals that evolve as in a GA (cf. Algorithm 1). Migration occurs after each generation along an oriented ring that connects the islands. Algorithm 8 gives the scheme of this hybrid EA and Table 3.10 shows its associated TEA. In the Talbi's taxonomy[9], it is noted **HCH(GA,AS)(het,glo,gen)**.

Algorithm 8
(* ISLAND-BASED GENETIC ANT ALGORITHM (IGAA) *)
determine k initial islands $[P^0, \ldots, P^{k-1}]$
initialize the trails on island P^0
generation_count $\leftarrow 0$
repeat
generation_count $\leftarrow$ generation_count $+1$
for each island i
if $i = 0$
then (* island is P^0: the evolution is based on an AS *)
for each individual (or ant)
construct a solution using trails and visibility
update the trails
else (* island is P^i with $i \neq 0$: the evolution is based on a GA *)
while $P^i_{\text{intermediate}}$ not full
select indi1 and indi2 in P^i
(offsp1, offsp2) $\leftarrow$ crossover(indi1, indi2)
put offsp1 and offsp2 in $P^i_{\text{intermediate}}$
mutate each individual in $P^i_{\text{intermediate}}$
$P^i \leftarrow P^i_{\text{intermediate}}$
the best individual of each island P^i migrates to $P^{(i+1) mod\ k}$
until termination condition is met

The parallelization of this algorithm is presented in 4.2.3. It is then used for experiments and the results obtained are discussed in Sections 5.6 and 6.6.

[8] EA ingredients are introduced at the beginning of the present chapter.
[9] Cf. Section 2.3.7.

$S(e)$ Set of elements e	$\lvert S\rvert$ = cst	structured S	information sources	infeasible e	h_e	improving algorithm	noise	evolution
Population(Individual)	Yes	compl	$2(p_c)$	nvr	No	No	Yes	gr
Colony(Ant)	Yes	No	$0\ h_S$	nvr	No	No	No	gr
Archipelago(Population,Colony)	Yes	ring	$2(\frac{1}{\lvert\text{Population}\rvert})$	/	No	No	No	gr

Table 3.10: TEA of IGAA

A common mistake people make
when trying to design something
completely foolproof is to underestimate
the ingenuity of complete fools.
The hitchhiker's guide to the galaxy.
Douglas Adams (1952–)

Chapter 4

Parallelization of evolutionary algorithms

This chapter presents a new approach to parallel EAs based on the algorithmic ingredients that were introduced in the previous chapter with the TEA. The aim is to parallelize sequential EAs without changing their behavior, and thus without altering or improving the quality of their results.

A notation for the granularity of parallel EAs is introduced and parallelization rules are given by interpreting the TEA ingredients. These rules make it easier to parallelize any given EA. Three EAs are then taken as examples and parallelized. Finally an object-oriented library is designed. It will permit to implement the resulting parallel EAs in order to validate the parallelization rules.

4.1 Parallelization analysis

The parallelization of an EA first requires to identify independent tasks that can potentially execute simultaneously, in order to distribute them on different PEs. The analysis of the dependencies between the tasks and of the amount of communication they need to exchange gives some hints on the way to parallelize the algorithm. A compromise must then be found between a maximization of the number of tasks and a minimization of the overhead due to the parallelization (because of communications, synchronizations, etc.). This section lists different parallelization rules that can be induced from the characteristics of EAs listed in the previous chapter. It can be noted that these rules are thus only based on algorithmic characteristics and should apply for any combinatorial optimization problem. The parallelization rules are finally applied to three different sample EAs.

Choices have to be made concerning the parallel architecture, the granularity, the communication load, and the parallel algorithm model.

4.1.1 The architectural choice

SIMD computers imply a synchronous behavior. They usually have small memories on each PE and they are almost not considered anymore by supercomputer vendors. MIMD computers, that are more and more available, are thus preferred as target platforms for the parallel algorithms that are discussed further on. SIMD computers can however have some advantages in specific cases that will be mentioned when necessary.

Shared memory architectures are supposed to solve all the data allocation problems at system or hardware level (cf. Section 2.4.2). The aim is to free programmers of any complex message passing problem. Nevertheless, in practice they are less flexible than distributed memory architectures because memory access is exclusively controlled by the system and the hardware: memory is usually physically distributed (hence possible memory contention), memory accesses might be sequentialized in some cases, etc. MIMD-DM computers are thus chosen in order to study the parallelization of EAs because of their flexibility (they can simulate other architectures if necessary) and increasing availability (cf. Section 2.4.2).

PRAM [30] and BSP [101] models are well-known theoretical parallel programming model. Their main objective is to give a formal framework to parallel programmers, but they do not exactly correspond to any existing parallel computer. Therefore, none of these theoretical models are chosen for the present work. Instead, a virtual MIMD-DM computer with p PEs mapped on a given topology is taken as a basis for the study.

The operating system of MIMD-DM computers is usually multi-tasking, that is, many tasks can be executed on a single PE. In the framework of this work, the efficiency of parallel programs is measured experimentally by executing them on parallel computers. Running several tasks on one PE is thus not consistent with such a study. Consequently, for the sake of clarity, in the remainder of this chapter it is always assumed that each PE runs a single task.

4.1.2 Levels of parallelization

The granularity of a parallel algorithm is usually only labeled as coarse or fine (cf. page 25). The *level of parallelization* introduced here formalizes this notion and makes it possible to quantify it more precisely in the case of evolutionary algorithms.

Definition: A *level of parallelization* describes the granularity of a parallel EA with an EA element: the size of the largest indivisible (i.e., not partitioned) element that is handled on PEs determines the *level of parallelization* of a parallel EA (e.g., an individual or a population).

This concept makes it possible, among other things, to compare the granularity of different parallel EAs.

Level	Element handled by each PE	Size x of each element	Notation
encoding element	1 encoding part	$x \in [1, \lvert\text{individual}\rvert - 1]$	$L_{-1}(x)$
individual	1 sub-population	$x \in [1, \lvert\text{population}\rvert - 1]$	$L_0(x)$
population	1 subset of archipelago	$x \in [1, \lvert\text{archipelago}\rvert - 1]$	$L_1(x)$
archipelago	1 set of archipelagi	...	$L_2(x)$
...	...	...	$L_i(x)$

Table 4.1: *Different levels of parallelization in EAs.*

Notation

Table 4.1 gives the list of different levels of parallelization that can be obtained depending on the choice of the indivisible elements that are distributed on PEs. The table also proposes a notation to name and order these levels: $L_\ell(x)$.

When only the level of parallelization is concerned, it is possible to mention level L_ℓ without any precision on the size of each element. It permits to give an idea of the parallelization without giving the exact distribution. For example, if an algorithm is parallelized at level L_0, each PE handles a sub-population whose size is in the range $[1, |\text{population}| - 1]$. Such a sub-population is a subset of the whole population.

An order can be defined by assigning low levels to fine-grain parallelism and high levels to coarse-grain parallelism:

$$\forall i, j \in [-1, \infty[, \forall x, y \in \{0\} \cup [1, \infty[,\ L_i(x) < L_j(y) \iff \begin{cases} i < j \text{ and } y \neq 0 \\ \text{or} \\ i = j \text{ and } x < y \\ \text{or} \\ x = 0 \end{cases} \tag{4.1}$$

$$\forall i, j \in [-1, \infty[, \forall x, y \in \{0\} \cup [1, \infty[,\ L_i(x) = L_j(y) \iff \begin{cases} i = j \text{ and } x = y \\ \text{or} \\ x = y = 0 \end{cases} \tag{4.2}$$

The notation $L_\ell(0)$ is necessary for the consistency of the formulae that will follow. It represents the granularity of the void algorithm. This level is noted $L_\emptyset = L_\ell(0), \forall \ell$.

Level constraints

The number of rows (i.e., description levels) in the TEA of an EA gives some information on the level range in which the parallelization can be done.

The level of parallelization of a sequential EA has no meaning in terms of parallelism. However the notation can be used to refer to the main EA entity (i.e., most global set S) that evolves in a sequential EA: a population, an archipelago, etc. By definition, the number of main entities is trivially 1 so the level of parallelization of a sequential EA must be $L_\ell(1)$ where $\ell \geq 1$ is the number of "algorithmic" description levels of the algorithm (i.e., the number of rows of its TEA description generally[1]). The smallest element (according to the definition given in Section 2.3) that is handled by an EA is a population. Consequently, the lowest level of parallelization that a sequential EA can have is $L_1(1)$.

The notation can be extended to define a level of parallelization range. Such a range can be useful to give a succinct yet precise description of the possible parallelization levels in which the parallelization of an EA can be investigated.

For example, if it is set that an EA $\mathcal{A}$ of level $L_1(1)$ must be parallelized by partitioning its main population of size |population| into sub-populations with at least two individuals, this information can simply be noted $\mathcal{A}_{/\!/} \in \,]L_0(1), L_1(1)[\, = [L_0(2), L_0(|\text{population}| - 1)]$, where $\mathcal{A}_{/\!/}$ is the parallel version of $\mathcal{A}$.

If an algorithm describes the evolution of a single population (as the classical GA does), it cannot handle several populations once it has been parallelized because its behavior must be left unchanged by the parallelization. More generally, a parallel algorithm always has a strictly lower level of parallelization (in the sense of Equation 4.1) than its original sequential version. For example, the sequential version of the classical GA has a parallelization level of $L_1(1)$: thus it cannot be parallelized at a level higher than $L_0(|\text{population}| - 1)$. In fact if we assume that the population is fairly partitioned into parts of similar size (without partitioning any individual), then each sub-population has $\left\lceil \frac{|\text{population}|}{p} \right\rceil$ or $\left\lfloor \frac{|\text{population}|}{p} \right\rfloor$ individuals. The highest level of parallelization is thus $L_0\left(\left\lceil \frac{|\text{population}|}{p} \right\rceil\right)$ with $p \geq 2$.

Influence of the levels

Parallelization level L_{-1} implies a fine-grain parallelism approach that is often too fine to be efficiently implemented on MIMD computers. For example, if each PE handles only a single part of an individual, then the number of necessary PEs is high. Moreover, the expected communication load necessary to maintain the consistency of partitioned individuals is likely to be huge. Another possible parallelization whose level is L_{-1} consists in assigning the same part of every individual to a given PE. For example, if individuals' genotypes are bit-strings, then a given PE could be responsible for a given bit of every individual (i.e., PE #7 would be in charge of the 7^{th} bit of each bit-string). Once again the number of necessary PEs and the communication load are high. Such fine-grain parallelizations should only be investigated for SIMD computer implementations. Moreover, the encoding of individuals' genotypes must be known in advance in order to distribute parts of individuals. The latter requirement does not depend on the algorithm, and thus a parallelization level L_{-1} will not be considered when parallelizing an EA in a general framework (i.e., that only depends on the algorithm).

Parallelization level L_0 is the most intuitive since it corresponds to the partitioning

[1] Since a given level might be described by several rows in the TEA, this is not always true (cf. page 40).

of a population (that is, the main entity of any original EA) into sub-populations. Such a partitioning requires that the information about the whole population be kept consistent, hence a likely high communication load. Such a parallelization is called a *global parallelization* in [19].

Parallelization level L_1 is the most popular in the literature because it is the easiest to implement. The idea is to run an original sequential EA on independent populations (or islands). A simple implementation of a parallel EA of level L_1 is straightforward: each PE handles a remote island and runs the sequential EA locally (on the condition that the number of individuals is at least twice greater than the number of PEs. Otherwise islands with one individual are considered as individuals, hence a level L_0). L_1 also includes more complex designs that permit to have more islands than PEs. *Sub-populations* that are part of a single population must not be mixed up with independent islands since they do not evolve according to the same algorithm.

A parallelization level $L_\ell, \ell \geq 2$ was not found in the literature since no EA with several archipelagi are currently being used.

Distribution of heterogeneous EA entities over several PEs: operator //

A level of parallelization $L_i(x)$ cannot describe parallel approaches in which the distribution of EA entities is heterogeneous (e.g., a population on a PE, and an individual on every other PE). Let us introduce an operator that can be used to describe such parallelization approaches.

Definition: If an EA is parallelized with a parallelization level $L_i(x)$ on some PEs and with a parallelization level $L_j(y)$ on some others then the resulting parallelization level is noted with the *operator* //: $L_i(x)//L_j(y)$.

The order in which the different levels are declared has no importance (i.e., operator // is commutative):

$$L_i(x)//L_j(y) = L_j(y)//L_i(x) \tag{4.3}$$

For example, if a farmer[2] PE handles a population while worker PEs process individuals, the parallelization level of the algorithm is $L_1(1)$ from the farmer PE point of view and $L_0(|\text{population}|/(p-1))$ with $p \geq 2$ from the worker PE point of view. The parallelization level is then noted: $L_1(1)//L_0(|\text{population}|/(p-1)), p \geq 2$. Such a level is lower than L_1 since it deals with individuals. It is however higher than L_0 since the whole population is handled on one PE. By generalizing this remark, the following property can be set:

$$\min(L_i(x), L_j(y)) \;\leq\; L_i(x)//L_j(y) \;\leq\; \max(L_i(x), L_j(y)) \tag{4.4}$$

When the operator // is used between two identical levels, the following simplification rule is set (this rule results from the definition of a level of parallelization, and it is

[2]Cf. Section 2.4.4

coherent with Equation 4.4):

$$L_i(x)//L_i(x) = L_i(x) \tag{4.5}$$

$L_\emptyset$ is a neutral element for operator //:

$$\forall i \in [-1, \infty[, \forall x \in [0, \infty[,\ \ L_i(x)//L_\emptyset = L_i(x) \tag{4.6}$$

Handling of heterogeneous EA entities by a PE: operator +

A level of parallelization $L_i(x)$ or $L_i(x)//L_j(y)$ cannot describe parallel approaches in which a PE handles heterogeneous EA entities (e.g., one population and some individuals). Let us introduce an operator that can be used to describe such parallelization approaches.

Definition: If a PE handles heterogeneous EA entities then each of these entities defines a different level of parallelization: $L_i(x)$ and $L_j(y)$ for example. *Operator* + aggregates these levels in order to describe the resulting level of parallelization: $L_i(x) + L_j(y)$.

For example, let us suppose that 3 islands of 10 individuals need to be partitioned on 2 PEs so that each PE handles 1.5 islands. The parallelization has two different levels simultaneously:

- a population level $L_1(1)$ (one island is distributed on each PE), and
- an individual level $L_0(10/2) = L_0(5)$ (5 individuals are distributed on each PE).

The level of parallelization of this parallel EA is noted: $L_0(5) + L_1(1)$. The level of parallelization hence obtained must not be higher than $L_1(2)$ (i.e., the lowest level obtained if the smallest entity considered here is replaced by the largest). Moreover, this level of parallelization must be higher than that of the smallest entity (i.e., $L_0(5)$). By generalizing this remark, the following property can be set:

$$\begin{gathered}\forall i, j \in [-1, \infty[, \forall x, y \in [1, \infty[,\ \min(L_i(x), L_j(y))\ \leq L_i(x) + L_j(y)\ \leq \max(L, L') \\ \text{where} \left\{ \begin{array}{l} L = \left\{ \begin{array}{l} L_{i+1}(1) \text{ if } x \text{ is the maximal size of level } L_i \\ L_i(x+1) \text{ otherwise} \end{array} \right. \\ L' = \left\{ \begin{array}{l} L_{j+1}(1) \text{ if } y \text{ is the maximal size of level } L_j \\ L_j(y+1) \text{ otherwise} \end{array} \right. \end{array} \right.\end{gathered} \tag{4.7}$$

The order in which the different levels are declared has no importance (i.e., operator + is commutative). $L_\emptyset$ is a neutral element for operator +:

$$\forall i \in [-1, \infty[, \forall x \in [0, \infty[,\ \ L_i(x) + L_\emptyset = L_i(x) \tag{4.8}$$

Operator + is artificially given a higher precedence than // in order to avoid multiple interpretations of a complex level written with several levels and both + and //:

$$\forall i, j, k \in [-1, \infty[, \forall x, y, z \in [0, \infty[,\ \ L_i(x)//L_j(y) + L_k(z) = L_i(x)//(L_j(y) + L_k(z)) \tag{4.9}$$

It is however advised to put brackets in order to improve the readability of the level.

Simplification of the notation

It often happens that the size x of a level $L_\ell(x)$ is the result of a ratio $\frac{y}{z}$ where y and z are some parameters of the algorithm (e.g., number of islands, number of PEs, etc.). In that case, the level of parallelization should be written $L_\ell(\lceil\frac{y}{z}\rceil)\,//L_\ell(\lfloor\frac{y}{z}\rfloor)$ because no hypothesis can be done on the divisibility of y by z. In order to keep the notation of levels readable, the following notation is introduced:

$$\left[\frac{y}{z}\right] = \left\{ \begin{array}{l} \lceil\frac{y}{z}\rceil \\ \text{and} \\ \lfloor\frac{y}{z}\rfloor \end{array} \right. \tag{4.10}$$

and it can simply be written $L_\ell(\lceil\frac{y}{z}\rceil)\,//L_\ell(\lfloor\frac{y}{z}\rfloor) = L_\ell([\frac{y}{z}])$.

4.1.3 Influence of the main ingredients

This section discusses the influence that the ingredients enumerated in Section 3.2.3 have on the parallelization of an EA. It must not be forgotten that these ingredients have a meaning for each row of the Table of Evolutionary Algorithms (TEA) introduced in Section 3.2. That is the reason why the following rules deal with a set S of elements e (like in Section 3.2.5) instead of a population of individuals, which would restrict the discussion to the first row of the TEA. General rules are deduced for each ingredient. They permit to study the TEA description of a given EA to find the most suitable parallelization. For example, if an ingredient value implies that the partitioning of a set S is too costly, then the possible level of parallelization range is reduced accordingly.

(1) Size of S If $|S|$ is big, then a first straightforward way to parallelize the algorithm is to distribute the elements in p subsets of size $\left[\frac{|S|}{p}\right]$. If the size of the set is not constant, an adaptive task management might be appropriate. However, if the variation is not too important it can be controlled by just resizing the subsets at run-time.

(2) Structured space (topology) If it is not possible to have a property preserving mapping of the topology of the space of elements (cf. page 33) on that of the parallel computer architecture, then there are neighbor elements that cannot be placed on neighbor PEs. Therefore the exchange of information between two such neighbor elements requires that messages be routed through several PEs, thus increasing the communication load. The worst case occurs when the space of elements is completely connected (i.e., any element can exchange information with any other). In that case, contention problems are to be feared and sophisticated routing algorithms are needed[3] because parallel computers are usually not fully connected. Favorable cases occur when the topology of the space can be mapped on the parallel computer architecture topology (e.g., a ring of elements

[3]MIMD-DM supercomputers usually have such efficient routing mechanisms.

can be mapped on a torus or a hypercube architecture). In the ideal case, elements do not exchange information (i.e., the space is unstructured).

When an algorithm is designed to run on a COW or a NOW, the topology of the physical links is not always known, and the routing is transparent (i.e., the topology is perceived as a complete graph). Moreover, if the algorithm needs to be portable on many different machines a strong hypothesis on the architecture topology should be avoided. Communications can then hardly be optimized by assuming a given topology. It is however more advisable to parallelize an EA at a level whose space is as unstructured as possible since it at least reduces the risk of contention, and since it simplifies the communication control.

(3) Information sources (history of S, number of parents and exchange rate) If the set S is partitioned on several PEs and if its history is used as an information source, then this history has to be kept consistent and each PE must be able to access it when necessary. The cost of this requirement is high in terms of communication, whatever technique is used to satisfy it (*gathering* of the information on a farmer PE, or *gossiping* of the information by PEs). Moreover, the history changes at each generation (or even more frequently), potentially increasing the communication load. Consequently, if the history of the set S is used as an information source, the set should be kept as a single entity as much as possible.

The potential amount of communication necessary to retrieve the information required by offsprings is proportional to the number of their parents, the amount of information exchanged by these parents, and the frequency of these exchanges (e.g., the number of offsprings created per generation). This potential communication load is thus represented by the number of parents and the exchange rate[4]. If these values are low, a partitioning of the set S is recommended, it is not advised otherwise.

(4) Infeasible solution The way infeasible candidates are dealt with is only of algorithmic concern. It does not influence the choices to be made when parallelizing an EA.

(5) Element history If a history is associated with each element, the information that models an element is larger. The communication load of a parallel algorithm that needs to exchange such elements is then increased in the same proportion.

(6) Improving algorithm An improving algorithm is usually applied on all elements at the same time. It is thus a good source of parallelism. A farmer/worker approach is well suited to quicken the improvement phase of the algorithm. However, it must be checked if the communication load is not too important compared to the amount of computation needed by the improving algorithm to process it. In other words, the size

[4]Cf. page 36 for the definition of these parameters and page 45 for an example of their interpretation when the set S is an island.

of the elements must not be too important compared to the complexity of the improving algorithm, otherwise it is better not to parallelize this part of the EA.

(7) Noise The use of noise in an EA is an algorithmic choice that does not influence significantly the efficiency of a parallelization, because it usually requires only a very small amount of computation. It could eventually increase the computation load of the PEs while the communication load is kept unchanged, but the difference should not be perceptible in most cases in terms of the efficiency of the parallel program.

(8) Evolution A generational replacement evolution[5] (gr) needs synchronization between consecutive generations. It is thus very sensitive to a fair task allocation. An implementation on an SIMD computer thus seems to be suitable if the topology of the structured space of the EA can be mapped on that of the architecture of the computer. On MIMD machines, a gr evolution needs extra synchronization. Yet, it is usually not necessary to actually synchronize the PEs between consecutive generations because this synchronization is obtained as a side effect of the exchange of messages (e.g., migration, update of global information, etc.) between PEs.

A steady state evolution (ss) is synchronous, but it changes only a few elements at each generation. The set S is used as a pool in which elements are selected or replaced. The information of the whole set is thus required for the consistency of the selections. Since most of the computational load is due to the processing of the individuals – selection, creation and/or replacement step – the control of individuals can be distributed on remote PEs. A steady state evolution is thus well suited to a farmer/worker approach: a farmer PE manages the set S and controls the worker PEs that handle the elements e. Only the few elements that must be changed are exchanged between the farmer and the workers, so the communication load stays low.

An asynchronous evolution (as) is non-deterministic because of parallel asynchronous instructions. Even if this behavior can be simulated on a sequential computer with a random generator that models irregular execution times, such an evolution is intrinsically parallel and is well suited to any task independent implementation on MIMD computers. For example, here is a possible simple implementation of an asynchronous EA: worker PEs compute individuals and send them to a farmer PE that updates the population without any control of synchronization.

Conclusion The only ingredients that do not provide any interesting information for the parallelization of an EA are the "noise" and the "infeasible solution" ingredients. This confirms the useful purpose of the TEA for a parallelization study. The "useless" ingredients are however kept in the TEA because it is meant as a general classification-tool based on the description of the algorithmic characteristics of EAs.

[5]Cf. page 34 for the description, and page 37 for the notation, of the different types of evolution.

4.1.4 Other important criteria for parallelization

At the beginning of the execution of an EA, data must usually be distributed on each PE. This distribution can be more or less time consuming depending on the characteristics of the parallel computer (mainly its architecture): whether every PE can access the same file system to get its information, or only one PE can load the information and must broadcast[6] it to the $p-1$ other PEs. Intermediate situations are also possible and broadcast algorithms are sometimes dedicated to specific machines. In any case, contention problems can occur and must be avoided as much as possible. This problem is not specific to parallel EA: it is common to all parallel algorithms. Its study is therefore too general to be discussed here. It is thus assumed in the remainder that the information is locally available on each PE at the beginning of the program execution.

The choice of the best parallelization technique depends on the algorithm itself, but also on some properties of the problem when they are known in advance (size range of its instances, time range to evaluate a candidate, etc.). The evaluation of the fitness value of an individual, for example, is usually viewed as a black box that attributes a value to each individual. This black box can require highly time consuming computation. Indeed, the computation of a fitness value sometimes requires complex simulations and the part of time dedicated to this work can be larger than that of the evolutionary process itself. For example, it happens when EAs are used to optimize technical systems [2] (nuclear reactor core reload, optical multi-layers, heat exchanger networks, etc.). This criterion does not appear in the TEA because it is strongly problem-dependent. It must however be taken into account when planning the parallelization of an EA if the information is available. Unfortunately, such information is not always available since a single algorithm might be used to solve very different problems.

At the end of the execution of an EA, the best individual found (i.e., the output result) must be known on at least one PE chosen in advance. The knowledge of the best individual must thus either be gathered on this specific PE at the end of the execution, or be kept up-to-date during all the execution on – at least – one specific PE (this is sometimes already done by some internal mechanisms of the original algorithm). However, the possible communication overhead is usually negligible compared to the total execution time of an EA.

4.1.5 Hybrid algorithms

The TEA was not designed to describe interactions between several EAs, and it cannot describe a traditional single-solution heuristic. It can thus only give rather limited information about a hybrid EA: it informs about the use of an improving algorithm but

[6]The complexity of such a broadcast depends on the architecture topology of the computer. It is at best $O(\log p)$ on a hypercube and $O(p)$ on a ring for example.

without giving its function, it informs about the use of islands but non-EAs cannot run on these islands, and pipelines (or relays) of different algorithms cannot be described.

The design issues of Talbi's taxonomy (cf. Section 2.3.7) are precisely made for describing interactions between EAs and are thus complementary to the TEA for the parallelization of hybrid meta-heuristics[7]: the description of the EAs being hybridized can be given by the TEA while their interactions can be described with Talbi's taxonomy.

The HCH (High-level Co-evolutionary Hybrid) class of Talbi's taxonomy corresponds to the island model of level L_1, and it can be parallelized with a level of parallelization $L_\ell \in [L_{-1}, L_0]$. The choice of the best level L_ℓ depends on the algorithms run on each island (cf. Section 4.1.3). If heterogeneous meta-heuristics are hybridized then a different level of parallelization is likely to be applied to each of them.

The HRH (High-level Relay Hybrid) class describes self-contained meta-heuristics that are executed in sequence. For example, an EA can be used to generate a solution that will then be improved by a local search algorithm, or an EA can take as input the results of a local search algorithm. Both choices can even be applied one after the other. The parallelization of such a hybrid algorithm can be made by parallelizing each meta-heuristic of the sequence independently. The meta-heuristics can then execute on the same PEs one after the other since each of them requires the "final" result of the previous one. A parallel implementation with a pipeline approach would also be possible if a lot of successive runs were planned and if the different phases had approximately the same execution time. It would however have a very bad efficiency for a single run of the algorithm because only the PEs responsible for one meta-heuristic would be used at once.

The LCH (Low-level Co-evolutionary Hybrid) class represents algorithms in which a given meta-heuristic is embedded into another meta-heuristic: typically, an operator (e.g., mutation or crossover) is replaced by a local search or a greedy algorithm. Since the embedded meta-heuristic co-evolves independently from the other(s) it can be easily applied on several individuals simultaneously (on remote PEs).

The LRH (Low-level Relay Hybrid) class represents algorithms in which a given meta-heuristic is embedded into a single-solution meta-heuristic. Typically, it can be a local search algorithm using an EA to profit from the advantages of diversification and exploration. In this case the EA can be parallelized as if it was alone and independent. Since it needs to be run many times (to provide its final result to the embedding algorithm) a system memorizing the global data of the problem instance between two runs can avoid to waste of time by often rereading them.

[7]The implementation issues of this taxonomy are not considered here because they only inform that the algorithm is "sequential" which is not a pertinent information for the parallelization, or they inform that the algorithm is already "parallel" and the parallelization is not necessary anymore!

If the hybrid EA is heterogeneous (het), then several potentially different parallelizations can be needed. If each meta-heuristic treats a different problem (spe) or a sub-problem of the problem instance (par) then the distribution of the EAs on different PEs permit to distribute the data accordingly, hence a profitable memory gain. Otherwise all meta-heuristics search in the same search space (glo,gen) and the problem instance must be duplicated: no memory space can be gained.

4.2 Case study

The parallelization rules enumerated in the previous section are now applied in order to parallelize three different EAs. The first one is a classical island-based GA, the second one is an island-based AS, and the third one is the island-based genetic ant algorithm introduced in Section 3.4. The latter was chosen in order to show the limits of the rules when they are applied to an atypical hybrid EA.

4.2.1 Parallel island-based genetic algorithms

Let us consider the island-based genetic algorithm (IGA) described by Algorithm 7 (page 38), and let us set the migration rate m to 1 (i.e., one individual migrates from each island every generation). Let us suppose that this IGA controls I islands of n individuals, and that it must be parallelized on p PEs. The compact form of the TEA description of this hybrid algorithm **HCH(GA)(hom,glo,gen)** is:

$$\left| \begin{array}{l} \text{Island(Individual) [Ycompl]2}(p_c)\text{nvrN[NYgr]}, \\ \text{Archipelago(Island) [Yring]2(1/|Island|)/N[NNgr]} \end{array} \right.$$

where p_c is the probability of applying a crossover to a pair of individuals.
It can be deduced from this description that the level of the sequential algorithm is $L_2(1)$. The parallelization level is thus in $[L_0(1), L_2(1)[$. The information that can be deduced from the first line of the TEA is enumerated below (the numbering (i) corresponds to the ith column[8] of the TEA):

(1) The size of an island is constant. Hence, no task mapping is required if islands or individuals are distributed.

(2) The topology is a complete graph. An exchange of messages on this topology should thus be avoided.

(3) Two parents are necessary to provide the information needed by an offspring. Since the topology is a complete graph, it is better to keep the individuals of an island on a same PE.

[8] Cf. Tables 3.1 and 3.3

(5) Individuals have no history. The cost to communicate a potential message containing an individual is thus minimal (only its encoding needs to be sent).

(6) There is no improving algorithm. The computation of a new individual is restricted to the computation of its fitness value.

(8) The evolution is generational. Synchronization is thus needed between consecutive generations.

Such an algorithm is difficult to parallelize at level L_0 because of (2) and (3). A level of parallelization L_0 should thus be avoided. The following information can be deduced from the second line of the TEA:

(1) The number of islands is constant. Hence, no task mapping is required if islands are distributed.

(2) The topology is a ring. An exchange of messages on this topology is not too costly.

(3) Two parents are necessary to provide the information needed by an offspring.

(5) Islands have no history. The cost to communicate a potential message containing an island is thus minimal (only its encoding needs to be sent).

(6) There is no improving algorithm.

(8) The evolution is generational. Synchronization is thus needed between consecutive generations.

The parallelization that best fits these criteria is a distribution of the islands on the PEs. Islands are not partitioned because of the communication overhead that this would produce, hence a parallelization level $L_1\left(\left[\frac{I}{p}\right]\right)$ with the following property:

- If $p \geq I$ then $L_1\left(\left[\frac{I}{p}\right]\right) = L_1\left(\left\lceil\frac{I}{p}\right\rceil\right) // L_1\left(\left\lfloor\frac{I}{p}\right\rfloor\right) = L_1(1)//L_1(0) = L_1(1)$. It can be noted that if $I = 1$ then the algorithm is not parallelized and if $p > I$ then $(p - I)$ PEs are not used.

- If $p < I$ then $(I \bmod p)$ PEs handle $\left\lceil\frac{I}{p}\right\rceil$ islands and the other PEs handle $\left\lfloor\frac{I}{p}\right\rfloor$ islands. If the number of islands is a multiple of the number of PEs (see Figure 4.1(b)), each PE handles the same number of islands, hence a fair load balancing. In the other case, some PEs have one more island than the others (see Figure 4.1(a)).

More formally, it can be stated that the minimum execution time of an island-based EA with homogeneous *indivisible* islands is bounded by the execution time of the PEs with the most islands (i.e., $\left\lceil\frac{I}{p}\right\rceil$). Assuming that the cost of communication is null and that all islands have exactly the same computational load[9], the maximum theoretical

[9] It cannot be the case exactly because EAs are highly randomized algorithms but it is a realistic hypothesis in average.

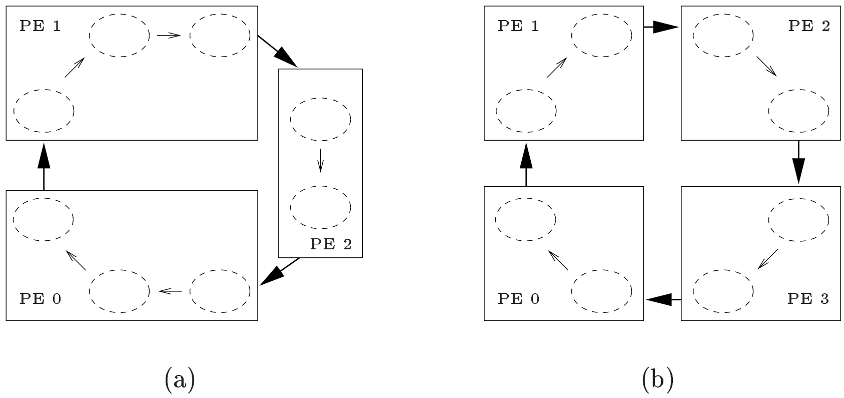


Figure 4.1: *Distribution of 8 islands positioned on an oriented ring of 3, resp. 4 PEs. Arrows model links allowing migration. Fat arrows model migrations requiring communications between PEs.*

speed-up that can be achieved is:

$$\mathrm{S}_{I\text{ islands}}^{\text{th.},\, p\, \leq\, I}(p) = \frac{I}{\left\lceil \frac{I}{p} \right\rceil} \tag{4.11}$$

It can be deduced from the definition of Efficiency (Equation 2.5) and from Equation 4.11 that:

$$\forall p \in [1, I],\ \frac{1}{2} < \mathrm{E}_{I\text{ islands}}^{\text{th.},\, p\, \leq\, I}(p) \leq 1 \tag{4.12}$$

The demonstration is given in Appendix B.1

4.2.2 Parallel island-based ant system

The island-based ant system (IAS) works like the IGA, except that an AS (cf. Section 2.3.4) runs on each island, instead of a GA. Algorithm 9 gives the scheme of this hybrid AS, that is classified as **HCH(AS)(hom,glo,gen)** in Talbi's taxonomy.

Let us suppose that this IAS runs I islands of n individuals and must be parallelized on p PEs. Its TEA is:

| Colony(Ant) [YN]0$h_S$nvrN[NNgr],
| Archipelago(Colony) [Yring]2(1/|Colony|)/N[NNgr].

The level of parallelization of the sequential ISA is the same as that of the sequential IGA ($L_2(I)$). Moreover, the second line of the TEA is also the same as that of the IGA. The analysis of the second line of the TEA made in Section 4.2.1 is thus valid here: a parallelization level $L_1\left(\left\lceil\frac{I}{p}\right\rceil\right)$ is then proposed. This parallelization uses all p PEs if $p \leq I$ exclusively.

Algorithm 9
(* ISLAND-BASED ANT SYSTEM (IAS) *)
1. determine k initial islands $[P^0, \ldots, P^{k-1}]$ and initialize the trails on each
2. **repeat**
3. **for** each island i
4. **for** each ant
5. construct a solution s_a using trails and visibility
6. evaluate the objective function at s_a
7. the best ant migrates to $P^{(i+1) \bmod k}$
8. update the trails
9. **until** termination condition is met

The information that can be deduced from the first line of the TEA is enumerated below (the numbering (i) corresponds to the ith column of the TEA):

(1) The size of the ant colony is constant and there is no parent. A parallelization of level $L_0\left(\left[\frac{n}{p'}\right]\right)$ is possible, where p' is the number of PEs on which an island can be partitioned.

(2) The space of the ant colony is not structured. A partitioning of the island is thus possible.

(3) The history of the colony is an information source. This information must thus be available on at least one PE. It is advised not to partition colonies when possible.

(5) Ants have no history. The cost to communicate a potential message containing an ant is thus minimal (only its encoding needs to be sent).

(6) There is no improving algorithm.

(8) The evolution is generational. Synchronization is thus needed between consecutive generations.

At this stage, a partitioning of the islands is possible and the information of the colony must be available on at least one PE. The parallelization that best corresponds to these criteria is a farmer/worker approach of level $L_0\left(\left[\frac{n}{p'}\right]\right)$ where p' is the number of PEs on which a colony is partitioned (p' must be greater than 1 otherwise the colony is not partitioned and thus not parallelized).

According to the second line of the TEA, it is better to distribute islands on PEs without partitioning them. Hence, as long as islands can remain unpartitioned (i.e., as long as $p \leq I$) they will not be partitioned. If $p > I$ then colonies are partitioned on $p' = \left[\frac{p}{I}\right]$ PEs.

The parallelism hence obtained has three levels, depending on the values of p and I:

- If $p \leq I$ then the level of parallelization is $L_1\left(\left[\frac{I}{p}\right]\right)$.
- If $I < p < 2I$ then the level of parallelization is $L_1(1)//L_0\left(\left[\frac{n}{2}\right]\right)$.
- If $2I \leq p$ then the level of parallelization is $L_0\left(\left[\frac{n}{\left[\frac{p}{I}\right]}\right]\right)$.

The distribution of I islands on p PEs is done as follows:

- If $p \leq I$ then (I mod p) PEs handle $\left\lceil \frac{I}{p} \right\rceil$ islands and the other PEs handle $\left\lfloor \frac{I}{p} \right\rfloor$ islands.
- If $I < p < 2I$ then each PE handles one island of size n or one sub-island of size $\left\lceil \frac{n}{2} \right\rceil$ or $\left\lfloor \frac{n}{2} \right\rfloor$.
- If $2I \leq p$ then each PE handles one sub-island of size $\left\lceil \frac{n}{\left\lceil \frac{p}{I} \right\rceil} \right\rceil$ or $\left\lceil \frac{n}{\left\lfloor \frac{p}{I} \right\rfloor} \right\rceil$ or $\left\lfloor \frac{n}{\left\lceil \frac{p}{I} \right\rceil} \right\rfloor$ or $\left\lfloor \frac{n}{\left\lfloor \frac{p}{I} \right\rfloor} \right\rfloor$.

Figure 4.2 shows the role of each PE on an example. The farmer sub-islands are responsible for the communications (migration) between islands, while worker sub-islands only communicate with their farmer sub-island in order to update the traces.

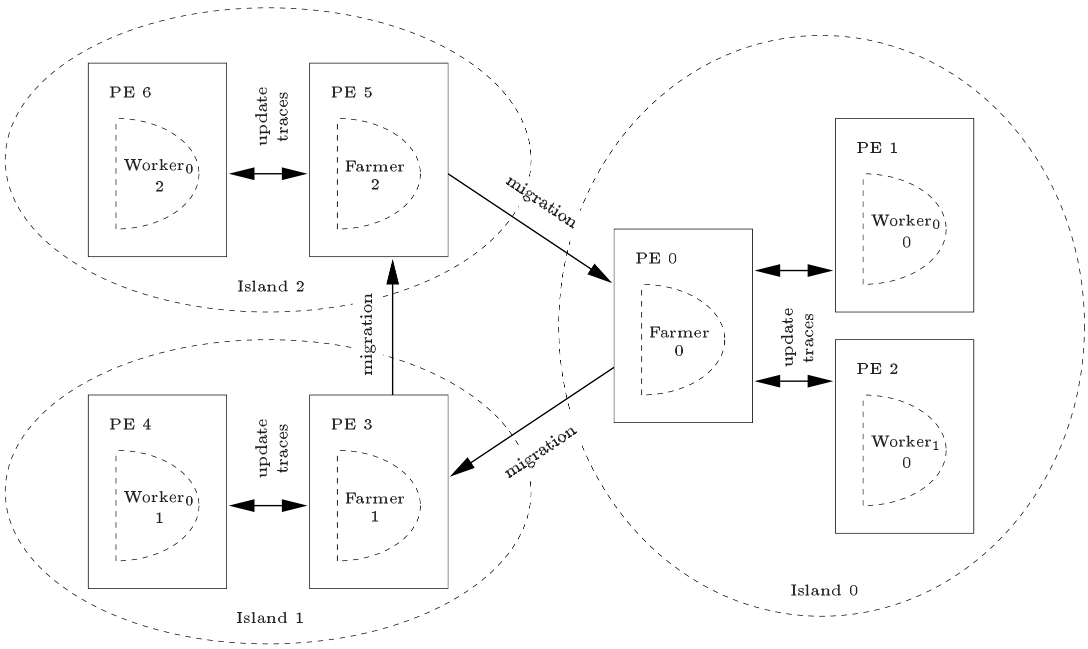


Figure 4.2: *Distribution of 3 islands on 7 PEs. Each PE handles a sub-island (farmer or worker).*

If $p \leq I$, the islands are distributed on the PEs without being partitioned, hence the same theoretical speed-up (with a null communication cost) as that of the IGA (cf. Equation 4.11). Otherwise, if $p > I$ it can be stated that the minimum execution time of an island-based EA with homogeneous *divisible* islands is bounded by the execution time of the PE(s) with the most individuals (ants here). Such PE(s) must handle a part of the less partitioned island, that is, an island with $\lfloor \frac{p}{I} \rfloor$ partitions. The size of the largest sub-population is thus $\left\lceil \frac{n}{\lfloor \frac{p}{I} \rfloor} \right\rceil$ hence the theoretical speed-up (with a null communication cost):

$$\mathrm{S}_{I\text{ size }n\text{ islands}}^{\text{th., }p\,\geq\,I}(p) = \frac{I \times n}{\left\lceil \frac{n}{\lfloor \frac{p}{I} \rfloor} \right\rceil} \tag{4.13}$$

Figure 4.3 shows an example of theoretical speed-up that is computed with Equations 4.11 and 4.13. This graph represents the maximum speed-up that can be achieved by the parallel IAS described above. The number of ants is not necessarily the same on each PE, and the PEs with the most islands bound the speed-up that can be achieved. Each time $\left\lceil \frac{I}{p} \right\rceil \neq \left\lfloor \frac{I}{p} \right\rfloor$ for $p \leq I$, and each time $\lceil \frac{p}{I} \rceil \neq \lfloor \frac{p}{I} \rfloor$ for $p > I$, there is a step on the speed-up graph. A 100% efficiency can only be achieved when the number of PEs is a divisor, or a multiple, of the number of islands (i.e., not for every step).

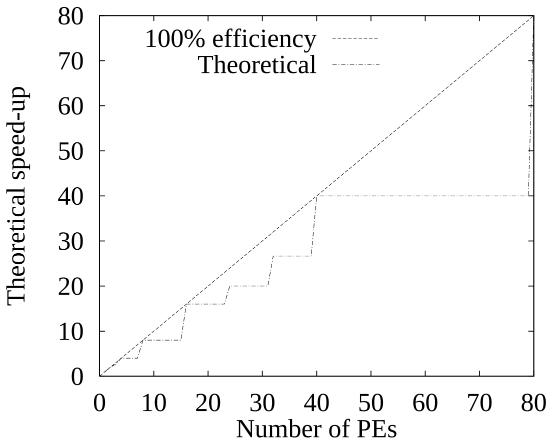


Figure 4.3: *Theoretical speed-up for 8 islands of 10 ants. It is computed by Equations 4.11 up to 40 PEs, and by Equation 4.13 from 40 to 80 PEs.*

As previously demonstrated for $p \leq I$, it can be deduced from Equation 4.13 and Equation 2.5 that the theoretical efficiency is greater than $\frac{1}{2}$ when $p \geq I$ (cf. the demonstration in Appendix B.2):

$$\forall p \in [I, I \times n],\ \frac{1}{2} < \mathrm{E}_{I\text{ size }n\text{ islands}}^{\text{th., }p\,\geq\,I}(p) \leq 1 \tag{4.14}$$

A fair partitioning of the islands could be done by distributing the ants on the PEs instead of distributing and partitioning islands. This would result in a level of parallelization:

$$L_1\left(\left\lfloor\frac{I}{p}\right\rfloor\right) + L_0\left(\left\lceil\frac{\left(I - \left\lfloor\frac{I}{p}\right\rfloor\right)n}{p}\right\rceil\right).$$

With such a parallelization the difference of computational load between two PEs is at most the computation required by one ant. However, this parallelization was not chosen in order to minimize the communication required to control the colony history (cf. the third parallelization rule deduced from the first line of the TEA). It can be noted that when $I \bmod p = 0$ this level of parallelization is the same as that of the parallelization deduced from the rules: $L_1\left(\frac{I}{p}\right)$.

The parallel IAS (and AS) described here was implemented and tested, and the results are presented in Chapter 5 and Chapter 6. Another study on parallelization strategies for the AS (without islands) can be found in [10]. A farmer/worker model was also chosen in this article. However, the speed-ups it presents were not based on real parallel executions but on simulation.

4.2.3 Parallel island-based genetic ant algorithm

Let us now parallelize the island-based genetic ant algorithm (IGAA) described by Algorithm 8 in Section 3.4. Let us suppose that it runs $I \geq 2$ islands of n individuals on p PEs. The compact TEA associated to this hybrid EA is:

| Population(Individual) [Ycompl]2(p_c)nvrN[NYgr],
| Colony(Ant) [YN]0$h_S$nvrN[NNgr],
| Archipelago(Population,Colony) [Yring]2(1/|Population|)/N[NNgr].

The first two lines of the TEA are respectively the same as the first line of the TEA of the IGA, and the first line of the TEA of the IAS. Moreover, the third line of the TEA is the same as their second line except that islands now evolve according to heterogeneous EAs. GA and AS have different computational loads and the difference between these algorithms is *a priori* not known: it depends on their implementation, on the complexity of their objective functions, etc. At this stage, islands are simply considered to be computationally homogeneous. Since the levels of parallelization proposed for the IGA and the IAS were identical when $p \leq I$, this level is proposed for the IGAA under the same condition:

- If $p \leq I$ then the level of parallelization is $L_1\left(\left\lceil\frac{I}{p}\right\rceil\right)$.

For $p > I$, the level of parallelization suggested by the first line of the TEA is still $L_1\left(\left\lceil\frac{I}{p}\right\rceil\right)$ (it is equal to $L_1(1)$ in this case) whereas the second line suggests the levels proposed for the IAS (cf. page 63). Islands are assumed to be homogeneous by lack of information. It is yet possible to guess that the constructive techniques applied by the AS are more time consuming than the operators of a GA. It is thus assumed that the island with ants needs a little bit more computation than the others in order to iterate one generation. This hypothesis is approximative, but it permits a first application of the parallelization rules in this atypical case: the level of parallelization applied to the ant island is tried to be kept lower than that of the other islands. The parallelization level that is proposed is then $L_0\left(\left[\frac{n}{\left[\frac{p'}{I'}\right]}\right]\right)//L_1\left(\left[\frac{I''}{p''}\right]\right)$ where p' is the number of PEs on which $I' = 1$ ant island is partitioned and p'' is the number of PEs on which the $I'' = I - 1$ other islands are distributed (with $p = p' + p''$). The resulting level of parallelization is then $L_0\left(\left[\frac{n}{p'}\right]\right)//L_1\left(\left[\frac{I-1}{p-p'}\right]\right)$ with $p' \in [p-1, p-I+1]$. For the parallel prototype used in the next chapters p' is chosen in order to minimize the highest component of the level (i.e., $L_1\left(\left[\frac{I-1}{p-p'}\right]\right)$): $p' = p - I + 1$. The consequences of this choice and of the hypothesis that the AS is a little bit more time consuming than the GA are shown and discussed in Sections 5.6 and 6.6. It results that:

- If $p > I$ then the level of parallelization is $L_0\left(\left[\frac{n}{p-I+1}\right]\right)//L_1(1)$.

Figure 4.4 shows an example of the distribution of islands for Algorithm 8 with the level of parallelization defined above. Islands are first distributed uniformly on the PEs, and when $p > I$ the ant island (numbered 0) is partitioned on the $(p - I + 1)$ remaining PEs.

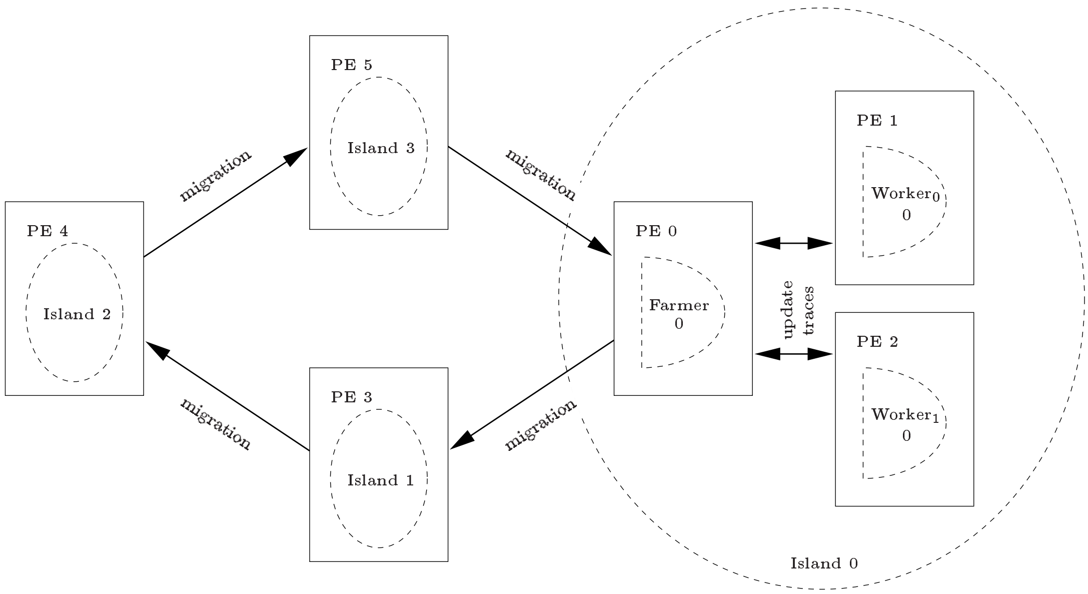


Figure 4.4: *Distribution of 4 heterogeneous islands on 6 PEs. Island 0 is an ant colony while the others are populations that evolve with a genetic algorithm.*

4.3 A library for evolutionary algorithms

4.3.1 Requirements

A software environment is necessary in order to test the parallel EAs described in the previous section. A library must be chosen to test them in a general framework, that is, to treat different problems encoded in different manners with different EAs. The same code should be reused as much as possible in each test case.

This library must be used to test the parallelization rules discussed in this chapter. Since the aim of these rules is to provide the best way to parallelize a given EA, the parallelization technique, that is chosen in each case, should be transparently applied (from the user point of view).

The necessity of a reusable and modular library that permits the use of complex data structures and that can encapsulate the parallel code leads to choose an object-oriented library (object-oriented concepts are described in [76]). This library must:

- make the implementation of EAs easier (including genetic algorithms, ant systems, etc.) by providing their data structures (e.g., population, individual, etc.), and their basic functions (selection, migration, mutation, etc.),

- make the conception of hybrid algorithms easier,

- encapsulate parallel computing functions in order to allow parallel executions as transparently as possible, without requiring any parallel computing knowledge,

- identify and isolate the parts of code related to:

    1 the algorithm,

    2 the encoding of the genotypes,

    3 the problem.

Some general comments can be made from these minimal requirements. First, the design of the library should be thought directly for distributed *sub-populations*, remote *islands*, etc. If parallel features are added afterward, the result of their integration in an existing library cannot be as elegant, and as efficient, as built-in functions. Second, the library must contain general EA classes, classes dedicated to specific EAs, problem specific classes and classes to encode the genotypes.

The choices of the mutation operator, the population size, and the stop criterion, are not algorithmic choices but only algorithmic parameters. It must thus be possible to make these choices at run-time through configuration files, and not when writing or compiling a program.

Name	Type	OS	Overview *(author of the software)*
DGenesis	GE, ED	Unix	A distributed implementation in which each island is handled by a Unix process. The topology between the islands can be set. *(E. Cantu-Paz)*
GALOPPS	GE	Unix, Dos	A general-purpose parallel GA system with a lots of options, and an optional graphical interface. *(E. Goodman)*
PARAGenesis	GE	CM	Implements a classical GA on a CM-200 in C*. *(M. van Lent)*
PGA	SS, GE	Unix	A simple testbed for basic explorations in GAs. Command line arguments control a range of parameters. Provides a lots of GA options. *(P. Ross)*
PGAPack	GA	Unix, Dos	A general-purpose, data-structure-neutral parallel GA library. Provides most of capabilities in an integrated, seamless, and portable manner. *(D. Levine)*

Table 4.2: *Description of C libraries dedicated to parallel GAs. The following acronyms are used: GA (Genetic Algorithm), GE (GEnerational GA), SS (Steady-State GA), ED (Educational Demo), CM (Connection Machine).*

4.3.2 Existing libraries

To date, 54 system packages[10] related to ES and GA can be found in the literature [58]. Among them, 5 libraries were designed to run parallel GAs. In fact, they run parallel *island-based* GAs (except PARAGenesis). These 5 libraries are written in C. They were developed by academic institutions, and all of them are freely available. They are listed in Table 4.2.

Among the 54 system packages, 11 are object-oriented libraries. None of these object-oriented libraries were designed for parallel computing. They are listed in Table 4.3. Further information, contacts and descriptions of EA libraries are available in [58].

None of the existing libraries fits, or can be extended to, the requirements introduced in 4.3.1. The main difficulty is that none of the object-oriented libraries were designed directly for parallel computing. A library that would be enhanced with parallel functions cannot be as consistent as a library that would be designed for parallel computing. Moreover, these libraries were not thought to integrate GA and constructive algorithms in a same framework. A new EA library that is not based on an existing one was thus developed. This new library is named APPEAL (for *Advanced Parallel Population-based Evolutionary Algorithm Library*).

[10] 14 of these packages are commercial products.

Name	Type	OS	Lang.	Free	Overview *(author of the software)*
EvoFrame/ REALizer	ES	Mac, Dos	C++/ OPas		A programming tool with a prototyping tool. It permits pseudo-parallel optimization of many problems at once. *(Optimum soft.)*
GAGS	GA	Unix, Dos	C++, perl	×	Features a class library for GA programming and is also a GA application generator (taking the function to be optimized as sole input data). *(J. J. Merelo)*
GAlib	GA	Unix, Mac, Dos	C++	×	Provides usual genetic operators and data representation classes, and permits to customize them. *(M. Wall)*
GAME	GA	WIN	C++		Aims to demonstrate GA applications and build a suitable programming environment. *(J. R. Filho)*
GA Workbench	GE, ED	Dos	C++	×	A mouse-driven interactive GA demonstration program. The source code is not provided. *(M. Hugues)*
Generator	GA, ES, ED	Win, Excel	C++	×	Solves problems using Excel formulae, tables and functions. Progress can be monitored and saved. *(S. McGrew)*
GPEIST	GP	Win, OS/2	Smalltalk	×	Provides a framework for the investigation of GP within a ParcPlace VisualWorks development system. *(T. White)*
Imogene	GP	Win	C++	×	Generates images by combining and mutating formulae applied to each pixel. The result is a simulation of natural selection in which images evolve. *(H. Davis)*
MicroGA/ Galapagos	SS	Mac, Win	C++		A tool which allows programmers to integrate GAs into their software. It comes with source, documentations, and an application generator. *(Emergent Behavior, Inc.)*
OOGA	GE	Mac, Dos	Lisp		Designed for industrial use. Each GA technique is represented by an object that may be modified. *(L. Davis)*
TOLKIEN	GE	Unix, Dos	C++	×	Designed to reduce effort in developing genetic-based applications by providing common classes. *(A. Y-C. Tang)*

Table 4.3: *Description of object-oriented libraries dedicated to EAs. The following acronyms are used: GA (Genetic Algorithm), GE (GEnerational GA), GP (Genetic Programming), SS (Steady-State GA), ES (Evolution Strategy), ED (Educational Demo), OPas (Object Pascal).*

4.3.3 Object-oriented model of APPEAL

The object-oriented model of the *Advanced Parallel Population-based Evolutionary Algorithm Library* is presented below. It corresponds to the requirements enumerated in 4.3.1.

The overall model of the library is given here using the notations of the Fusion method [24]. This method divides the process for software development into several phases. Since the complete description of the application of these phases would be lengthy, only the analysis phase is overviewed here. The implementation choices are explained in 4.3.4. The notation used in the following figures is intuitive and relies on simple object-oriented software design concepts. It is described in Table 4.4.

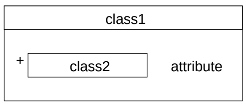
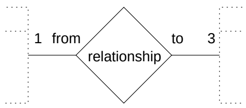
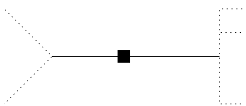
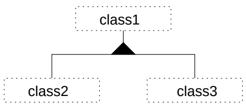
Representation	Definition
	Classes are represented by rectangular boxes that can contain other boxes (i.e., aggregate other classes) and attributes. The cardinality of each aggregated class appears in its upper-left corner (a '+' means "at least one").
	Relationships between classes are written with diamonds. The cardinality and the role of each class involved in the relationship appears on the line that links the diamond to the classes.
	Small black squares indicate that a relationship is mandatory.
	Super-classes are written above sub-classes, and a small triangle drawn on the line that links super-classes and sub-classes models their inheritance relationship.

Table 4.4: *Notations used to describe the object-oriented model. It is a subset of the notations used in the analysis phase of the Fusion method [24].*

Figures 4.5 and 4.6, whose descriptions follow, are taken as examples to illustrate the use of this notation.

Figure 4.5 shows the main classes needed to design an evolutionary algorithm. The largest rectangular box represents the Evolution class that controls the evolution of an algorithm. It aggregates one Transcoder and at least one Population. The Transcoder is used to encode (and decode) the information in the Genotype of every Individual contained in a Population. A more detailed description of these classes is given in the next pages.

Figure 4.6 shows the use of sub-classes inherited from the class Genotype. This class models the genotype of an individual. It has two attributes (size and maxValue), and one aggregate class (RandomGenerator). The attribute size represents the number of

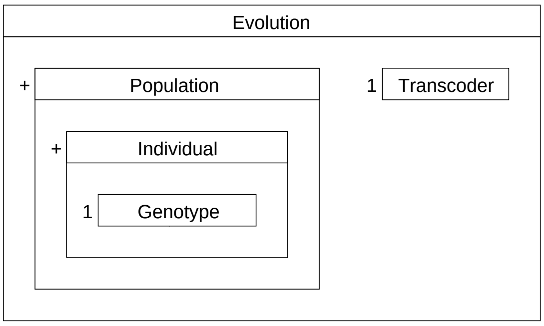


Figure 4.5: *Example of class aggregation: the class* Evolution *aggregates one* Transcoder *and at least one* Population*, that aggregates at least one* Individual*, etc.*

components that constitutes the genotype (e.g., the length of a genotype array), and maxValue is the maximum value that a component can take. Class RandomGenerator is necessary to construct random genotypes.

The relationships between Genotype and Transcoder (i.e., construct and evaluate) are shown in diamonds in Figure 4.6. These relationships are mandatory, that is, a Transcoder must have a method to construct a Genotype and a method to evaluate this Genotype. The class Transcoder translates combinatorial optimization problem information into Genotype encoding. It is the only class that is "aware" of the problem and of the way its candidates are encoded.

In Figure 4.6, the first level of inheritance determines the way class Genotype is encoded[11] (e.g., BoolGT represent a boolean vector encoding that consists in at least one boolean component of type Bool). The second level of inheritance is optional. It permits the enhancement of the interface of Genotype by adding specific attributes and operators. For example, in a GA an individual must be encoded by a genotype that inherits from GeneticOperator (e.g., GeneticBoolGT), in order to have an attribute isMated and a crossover operator (that produces two GeneticOperator's from two GeneticOperator's). The implementation of these operators is different for each encoding of Genotype and is not written in GeneticOperator. This second inheritance is thus only possible if the implementation of the operators is made in the class that defines the encoding of Genotype (e.g., BoolGT, IntegerGT, etc.).

Figure 4.7 shows how a Transcoder can construct a Genotype. First, the Transcoder determines the choice that must be made, that is, the set of solution elements that could be added to the partial solution encoded in a Genotype. Second, the Transcoder sets the probability value of each solution element. Problem-specific knowledge (known as

[11] This first level of inheritance can in fact have several intermediate levels of inheritance (not shown on Figure 4.6). For example, sub-classes such as VectorGT or MatrixGT can inherit from Genotype and they can then be specialized in BoolVectorGT, IntegerMatrixGT, BoolMatrixGT, etc.

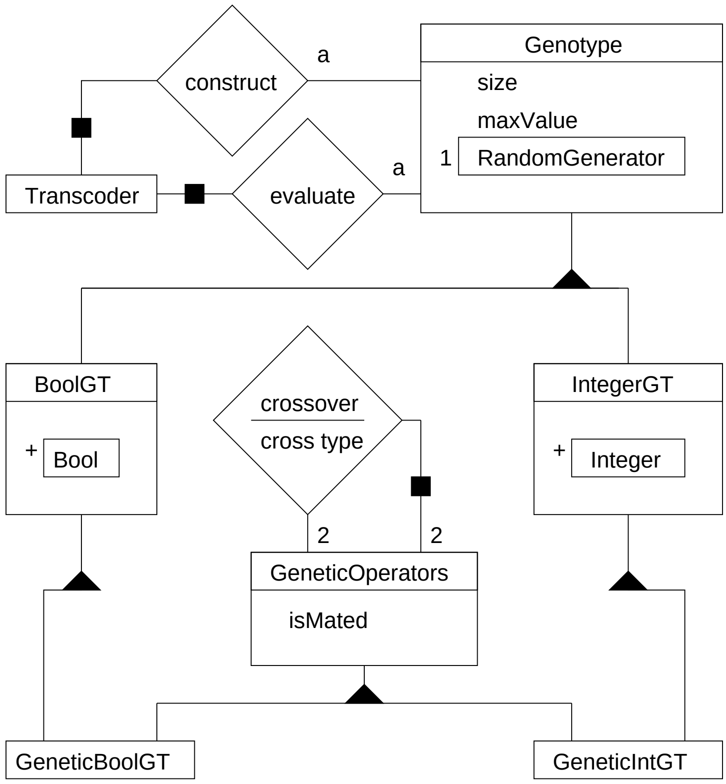


Figure 4.6: *The class* Genotype*, its sub-classes, and its relationships with the* Transcoder*.*

visibility[12]) and external evolutionary information (called Trail[12]) can be used to compute this probability. It is then possible to choose a SolutionElement from a Choice according to different rules:

1 the choice depends on the probability for each SolutionElement to be chosen, or

2 the choice is made totally at random, or

3 the SolutionElement with the highest probability is always chosen.

Finally, the Transcoder adds the chosen SolutionElement to the Genotype that represents a partial solution. These steps are repeatedly applied until the Genotype represents a complete solution.

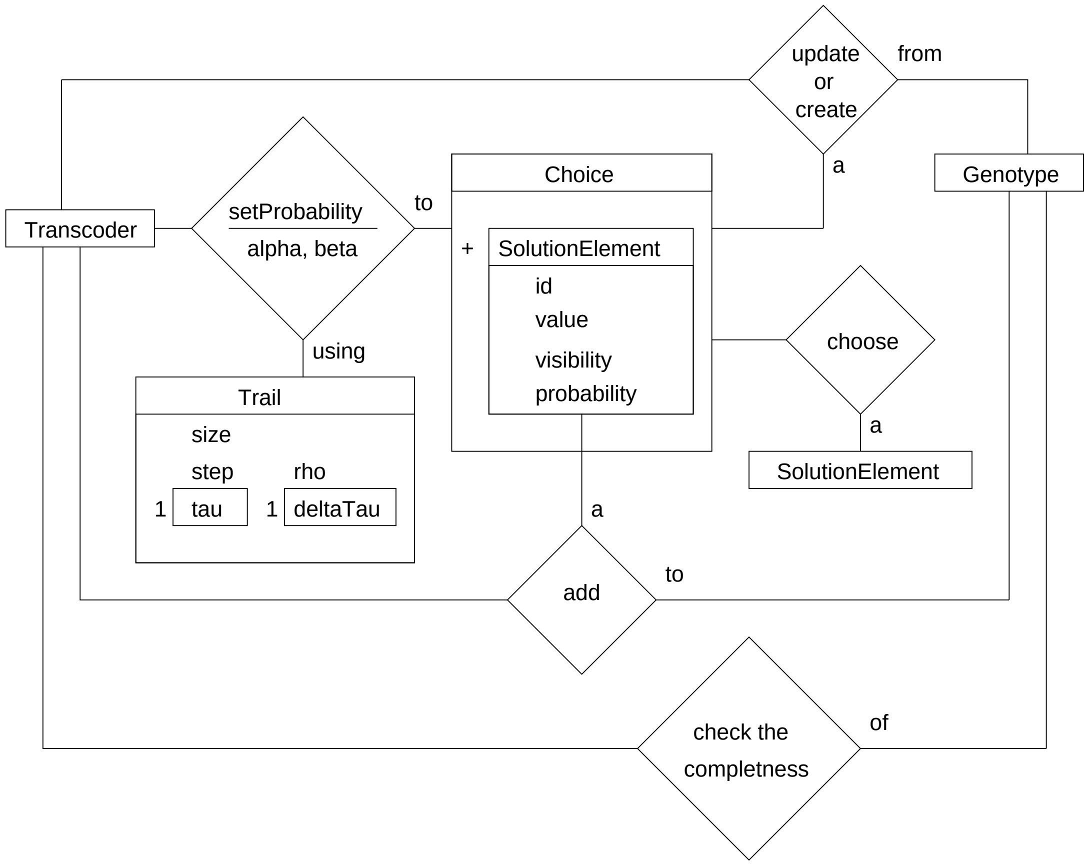


Figure 4.7: *Classes used to construct a* Genotype.

Figure 4.8 shows the main characteristics of the class Population. It contains many attributes and a set of Individual's. It is possible to replace any of its Individual's and to

[12]Cf. Section 2.3.4

select one of them according to different rules (at random, according to its fitness value, etc.). It is also possible to scale an Individual in order to normalize its fitness value within a Population. A Population "remembers" the best Individual (that with the highest fitness value) that has ever been part of it. A Transcoder can update an Individual, that is, it can check and (re)compute each of its attributes in order to make them consistent.

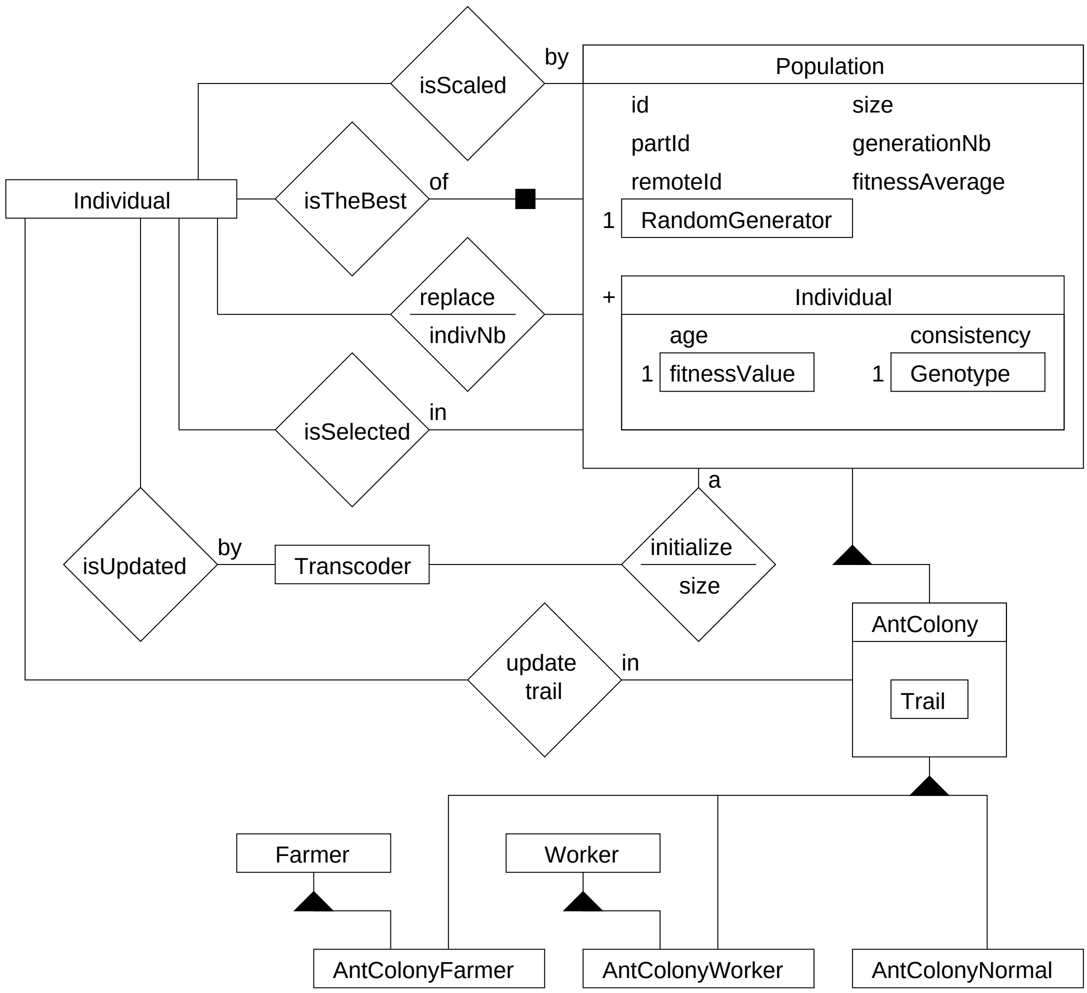


Figure 4.8: *Class* Population, *its sub-classes, and its relationships with* Individual *and* Transcoder.

An AntColony is a Population that contains and evolves Trail's. No specific class is designed for ants since they are similar to Individual's. Trail's are thus updated by Individual's.

If a population needs to be partitioned on remote PEs, it can be implemented either as a "farmer" or as a "worker" Population. A "farmer" Population is a part of a population that is responsible of the consistency of the information for the whole Population. A "worker" population is a part of the population that exchange information with its "farmer" Population. If the population does not need to be partitioned, it can simply be implemented as a "normal" population. An example is given with AntColony in Figure 4.8. The classes AntColonyNormal, AntColonyFarmer, and AntColonyWorker are only used for the internal mechanisms of the library. Therefore, this part of the model can be ignored by a programmer who only wants to use the library.

Class Evolution, shown in Figure 4.9, is the heart of evolutionary algorithms. It contains all the information and specific data structures an EA needs, that is, at least one Population, a Transcoder whose specialized sub-classes permit to treat specific combinatorial optimization problems, general evolutionary parameters, a ComBox to control parallel executions, a RandomGenerator and a Timer that can be used by a StopCriterion or for statistical observations.

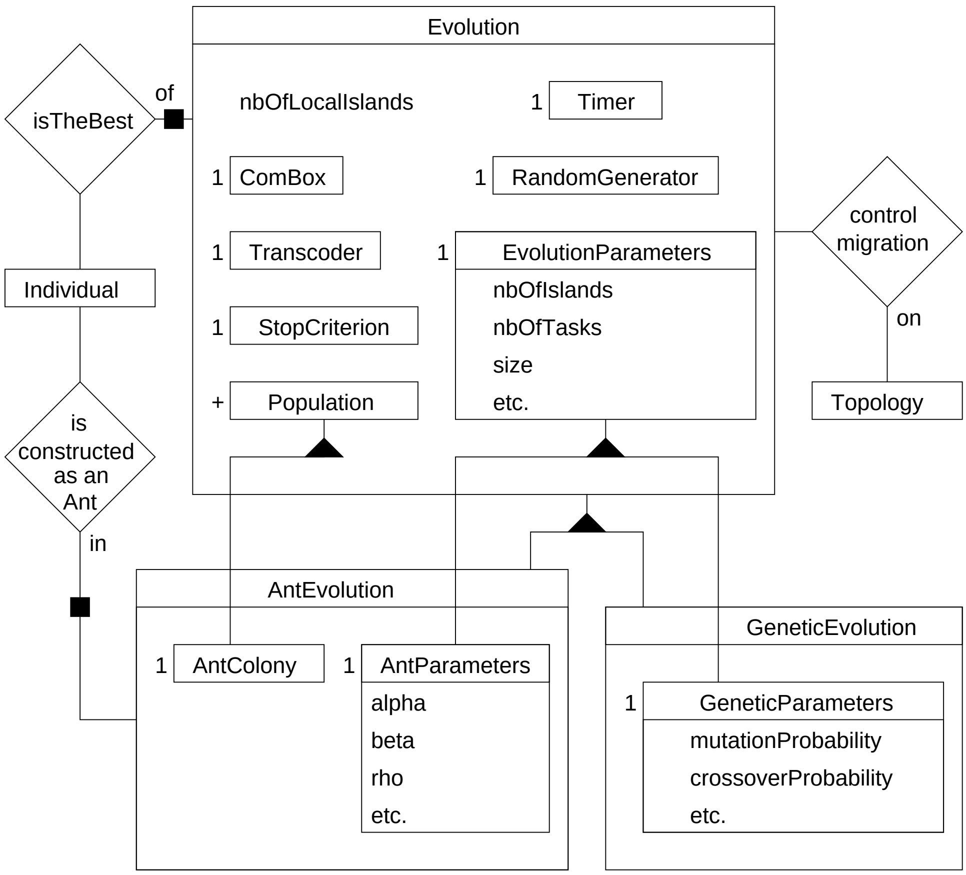


Figure 4.9: *Class* Evolution *and two examples of its sub-classes (for an AS and a GA).*

ComBox provides standard message passing methods such as *sendTo, receiveFrom,* etc., to send and receive messages from one PE to another. The purpose of this class is to hide the system specific-calls to message passing functions. The class Transmissible uses ComBox to send and receive contiguous memory segment that encode "transmissible" objects. The Serialisable class is similar to Transmissible but it provides packing and unpacking methods that automatically encode any "serialisable" object in a contiguous memory segment. Any object whose class is inherited from Serialisable can therefore be sent to (resp. received from) a PE by simply calling the *sendTo(targetPE)* (resp. *receiveFrom(sourcePE)*) method. Figure 4.10 shows typical classes that are "serialisable" (e.g., Individual).

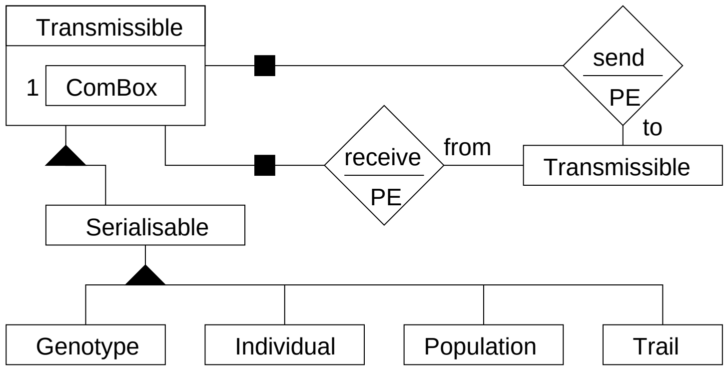


Figure 4.10: *The parallel computing class hierarchy.*

Evolution has potentially as many sub-classes as there exist different EAs. It controls the migration of Individual's on a given Topology, and it can return the best Individual that was ever found since its construction.

An AntEvolution aggregates specialized sub-classes of Population and EvolutionParameters in order to execute ant systems. It is for example capable of constructing an Individual as an "ant" (cf. Algorithm 3).

The originality of this model is the consideration of evolutionary and constructive approaches in the same framework. This permits, among other things, to implement ASs that need to construct ants during the evolution of the population they control. The information needed to construct the Genotype of an Individual that represents an ant is in the following classes: Transcoder, Trace, RandomGenerator and AntParameters. The construction of such an Individual in an AS is thus possible by using the relationships shown in Figure 4.7, because all these classes are available in AntEvolution class.

In this object-oriented model, classes are divided in three distinct categories:

1 The classes related to EAs: Evolution, GeneticEvolution, AntEvolution, Population, AntColony, Individual, Genotype, GeneticOperators, Choice, SolutionElement, EvolutionParameters, GeneticParameters, AntParameters, etc.

2 The classes related to the encoding of the candidates into genotypes: IntegerGT, BoolGT, etc.

3 The classes related to the problem to be solved: Graph, ColoringParameters, etc.

Figure 4.11 gives a graphical representation of this partition into three categories. The only class that cannot be classified in one of these categories is the Transcoder class that serves as an interface between the classes related to the problem and those related to the encoding of its candidates. Such a class is required since the encoding/decoding of a candidate (that is problem-dependent) into a given genotype (that is encoding-dependent) is necessary in any EA.

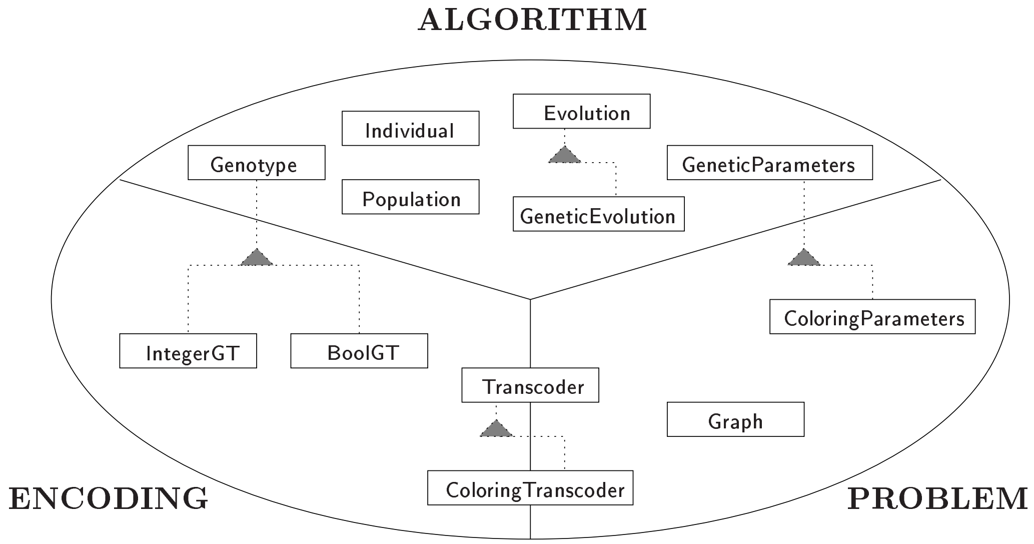


Figure 4.11: *Partition of the classes into three categories. The* Transcoder *class is the link between the ENCODING and the PROBLEM categories. It is the only one that can encode a given problem modeling into a genotype encoding, and decode a given genotype encoding into the proper problem modeling.*

Classes that are only necessary for the internal mechanisms of the library are not considered here because they are not explicitly used in a program. Most of these classes are standard classes (e.g., Array, String, RandomGenerator, Timer, etc.). The others are used for the transparent parallelization of the EAs (e.g., AntColonyFarmer, Serialisable, ComBox, etc.).

Let us suppose that the graph coloring problem must be solved by the hybrid genetic ant algorithm of Algorithm 8. Figure 4.12 shows how classes dedicated to this specific problem are integrated in the framework of the library: specific parameters (e.g., maximum number of colors to use) are attributes of ColoringParameters, the graph to be colored is modeled by the Graph class, and the fitness value of GeneticIntGT that encode candidates is computed by the ColoringTranscoder with the FitnessFunction class.

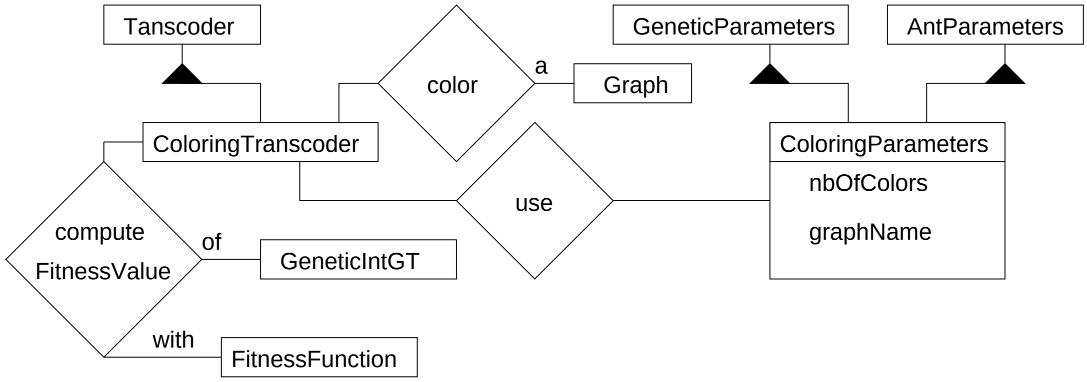


Figure 4.12: *An example of instantiation of APPEAL classes for the graph coloring problem.*

4.3.4 Implementation of APPEAL

The implementation of the *Advanced Parallel Population-based Evolutionary Algorithm Library (APPEAL)* was done in C++ [95]. Even if this language has many gaps [64] and is sometimes limited with respect to some object-oriented concepts (like genericity). C++ is widely used and permits to benefit from existing libraries and programs to interface with.

The kernel of a GA C++ program available at EPFL was used as a basis to write APPEAL. The code is written according to the object-oriented recommendations of [59] and [29]. It tries to reuse existing standard classes as much as possible (according to object-oriented programming style). For example, the C++ library LEDA (Library of Efficient Data types and Algorithms) [80] is used to avoid rewriting basic classes such as String, List, and Set. Nevertheless, it would be easy to replace LEDA by any other library providing the same basic classes.

The implementation of message passing functions is currently based on the PVM library [41]. Function calls to the PVM library are well integrated to the rest of the library, but are however not spread all over the classes. They are encapsulated in only one class – ComBox – whose interface exhibits basic message passing methods. ComBox could easily be rewritten with any other message passing library (like MPI [91] or any parallel computer specific library), hence a good portability of APPEAL on any MIMD computer.

The portability of the library was not checked on every possible platform, since the aim of this work was primarily to use APPEAL within different projects (LEOPARD, STORMS, PERFO) in order to apply and test the parallelization rules. However, recent tests (e.g., with egc) showed that the use of APPEAL with up-to-date compilers is straightforward.

4.3.5 Current state and future evolution of APPEAL

A complete description of the C++ library APPEAL (release 2.2, August 1999) is given in a reference manual [11]. The current version of APPEAL does not include all the classes required for every EA and every possible genotype encoding, but only those necessary for the EAs presented in the last two chapters of this thesis. It currently includes:

- all the general EA classes: Evolution, Individual, Genotype, etc.,
- the classes dedicated to island-based generational replacement GAs, ant systems, and one of their hybridization (cf. Section 3.4): GeneticEvolution, AntEvolution, GeneticAntEvolution, AntColony, Trace, etc.,
- the classes to encode the Genotype into a bit-string (BoolGT) and into a real-valued array (IntegerGT),
- the Transcoder class, that consists of the minimum interface that is necessary to encode a given problem candidate into a Genotype and to decode the latter,
- a simple example of classes necessary to treat a given combinatorial optimization problem: the graph coloring problem that is described in Chapter 6 (ColoringParameters, ColoringTranscoder, FitnessFunction)[13],
- the parallel processing package (ComBox, Serialisable, Transmissible) that is necessary to execute parallel programs.

APPEAL can easily be enhanced by adding new classes at the condition that they satisfy the primary specifications stated in 4.3.1. For example, the only topology currently implemented is the ring. It would be interesting to implement (or use an existing) software library that provides topology classes in order to change them as easily as any other parameter of a program written with APPEAL.

Let us suppose that a problem is solved by a program written with the current classes of APPEAL. If an evolution strategy (cf. Section 2.3.2) must be tested on this problem, then only two classes must be written (EvolutionStrategyEvolution and EvolutionStrategyParameters) since the other classes already exist (Population, etc.). If a new encoding of the candidates must be tested, then:

- the class of the new encoding – a sub-class of Genotype – is possibly written (if it is not available in APPEAL),
- the declaration of the encoding class is changed in the main program (only one word must be changed),
- the encoding and decoding methods corresponding to the new encoding must be written in the sub-class of Transcoder that is associated to the problem (e.g., NewProblemTranscoder),

and the new program can be compiled.

[13]These classes are used to implement the programs of Chapter 6.

4.4 Alternative approaches to the parallelization of EAs

The choice made in this thesis is to parallelize EAs without changing their original behavior. This section presents two alternative approaches that do not satisfy this choice.

4.4.1 Parallelization based on autonomous agents

Instead of parallelizing EAs by managing the distribution and the partitioning of elements (individuals, populations, etc.), it might be possible to consider autonomous *agents* that represent these elements. For example, each individual could be an autonomous agent that would evolve according to its own evolutionary rules. It could:

- mutate,
- select other "individual agents" to mate with,
- migrate from an island to another on its own initiative (even if these islands are on different PEs),
- evaluate its own fitness by submitting its genotype to a transcoder-agent[14].

With such an approach it would probably be difficult to ensure that the behavior of the algorithm is similar in sequential as in parallel. Moreover, if each individual-agent needed to be aware of any other, the communication load would be very high (each agent sending its request, the same message would be sent several times, hence a likely inefficiency in this case).

This approach could however be used in some particular cases (e.g., when the individuals do not share a global information and do not need to be aware of each other). The TEA description can help to identify such situations. Here are the minimal rules that should be satisfied in order to be able to parallelize an EA with autonomous agents (the numbering (i) corresponds to the i^{th} column of the TEA):

(2) The set of elements should be unstructured. If it is structured, the topology used to map it should have a small diameter.

(3) The exchange of information should involve a small amount of elements and must not use the history of the set.

(8) The synchronization of agents is very costly. An asynchronous evolution is thus necessary.

[14]The notion of transcoder is explained page 72.

4.4.2 Asynchronous parallelization

Let us suppose that a parallel EA is executed on heterogeneous PEs. In this case, the PEs have different speed, and some of them may end their computation much later than the others. Let us now suppose that the load balancing of a parallel EA is unfair (on homogeneous or heterogeneous PEs). In this second case, the same problem occurs. The PEs that are in charge of the greatest amount of computation may end their computation much later than the others. In this two cases, if the parallel EA is synchronous, then an important part of time is wasted by waiting for the end of the computation on the slowest (resp. most charged) PE. It is possible to avoid this waste of time by taking advantage of asynchronous communications. However, this implies that the original algorithm be transformed into an asynchronous one.

An asynchronous evolution can be inspired by a generational or a steady-state evolution with a slight modification: the removal of any synchronization constraint. This modification has no positive algorithmic effect since it can lead to a partial loss of information that might decrease the quality of the solution. When executed on a heterogeneous network of PEs (a NOW, for example), the resulting asynchronous evolution should however be faster than the synchronous evolution it is inspired by. The main question that should be considered when designing an asynchronous evolution this way is: "Is the time gained worth loosing a given ratio of the solution quality?[15]"

For example, in the parallel algorithms introduced in Section 4.2, each PE must wait to receive the best individual from its neighbor before starting the evolution of the next generation. The probability of having a slow PE in a heterogeneous network of workstations increases with the number of PEs used. Thus, when the number of PEs is high, an important part of time is wasted. The following modifications can be made to the original parallel island-based algorithms described in Section 4.2: any synchronous communication due to individual migration is replaced by an asynchronous one, and the algorithm stops when at least one island has reached the stop criterion (e.g., it has evolved during a given number of generations). Algorithm 10 gives the scheme[16] of this AIEA (Asynchronous Island-based Evolutionary Algorithm).

This approach is not consistent with the aim of studying the parallelization of EAs without changing their behaviors. This approach is thus not considered in the remainder. It is however presented here since the study of such asynchronous parallel EAs (in terms of speed-up and performance) would be an interesting complement to the present work.

[15] A loss of performance is probable since in an asynchronous EA the amount of information exchanged is altered by the system.

[16] The ring topology is kept in order to allow comparison with the other algorithms. This scheme can however be generalized to any topology.

Algorithm 10

(∗ Asynchronous island-based evolutionary algorithm (AIEA) ∗)

determine k initial islands $[P^0, \ldots, P^{k-1}]$
generation $\leftarrow 0$
repeat on each island P^i simultaneously (without synchronizations)
    generation $\leftarrow$ generation $+1$
    apply evolutionary operators in P^i
    the best individual of P^i is sent to $P^{(i+1) \bmod k}$
    if at least one individual was received from $P^{(i-1) \bmod k}$ since last time
        then the individual the most recently received is put in P^i
until at least one island satisfies the termination condition

The wireless telegraph is not difficult to understand. The ordinary telegraph is like a very long cat. You pull the tail in New York, and it meows in Los Angeles. The wireless is the same, only without the cat.
Albert Einstein, physicist (1879–1955)

Chapter 5

Transceiver siting application

One of the key issues telecommunication companies must face when deploying a mobile phone network is the selection of a good set of sites among those possible for installing transceivers or *Base Transceiver Stations (BTSs)*. The problem comes down to serving a maximum surface of a geographical area with a minimum number of BTSs. The set of sites where BTSs may be installed is taken as an input, and the goal is to find a minimum subset of sites that allows a 'good' service in the geographical area. This *transceiver siting problem* is tackled in the European project STORMS[1] which aims at the definition, implementation, and validation of a software tool to be used for design and planning of the UMTS[2] network project.

In this chapter, a model of the transceiver siting problem as well as different programs that were developed to solve it are described. The programs are based on the parallel EAs described in Section 4.2 and on other algorithms used for comparison. The following sections elaborate on speed-ups achieved experimentally, and the last one overviews the quality of the results returned by each algorithm.

5.1 Problem modeling

5.1.1 Urban radio wave propagation simulation software

This section briefly presents a urban radio wave propagation simulation software, called ParFlow++. It is not thoroughly described in this manuscript for homogeneity reasons. However, its development was necessary to create realistic input data for the transceiver siting applications introduced in this chapter, and it is thus presented here.

[1]STORMS stands for Software Tools for the Optimization of Resources in Mobile Systems.
[2]UMTS stands for Universal Mobile Telecommunication System

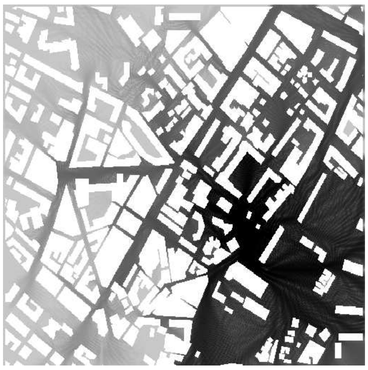


Figure 5.1: *Results of a radio wave propagation simulation achieved for a 1 km^2 district of the city of Geneva. The darker the grey, the better the signal reception.*

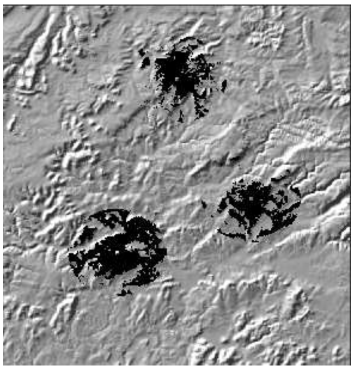
In 1995, a new approach to modeling radio wave propagation in urban environments based on a Transmission Line Matrix (TLM [61]) was designed at the University of Geneva [75]. The ParFlow method compares with the so-called Lattice Boltzman Model, that describes a physical system in terms of motion of fictitious microscopic particles over a lattice [8]. The ParFlow method permits fast bidimensional radio wave propagation simulation, using a digitized city map, assuming infinite building height (see Figure 5.1). It is thus appropriate for simulating radio wave propagation when transmitting antennas are placed below rooftops. This is the case in urban networks composed of micro-cells.

ParFlow++ denotes an object-oriented, irregular implementation of the ParFlow method, targeted at MIMD-DM platforms. Its purpose is to compute cells covered by BTSs in a urban environment. To date the use of object-oriented programming is not very common in parallel super-computing. For this reason implementing the ParFlow method using object-oriented techniques appeared to be an appealing challenge. ParFlow++ runs on networks of workstations, on a Cray T3D, and on a SGI Origin 2000. This work is described in details in [53, 49, 48, 54, 50, 51, 52].

5.1.2 Cells

A geographical location is said to be *served* when it can receive the signal broadcast by a BTS with a given quality of service[3]. The area served by a BTS is called a *cell*. A cell is usually not connex. It must be noticed that, since each BTS is associated to a cell, the distinction between a BTS, its site and its cell will not be done in the remainder of this chapter. The computation of cells may be based either on sophisticated wave propagation models, on measurements, or on draft estimations. In any case, we assume

[3] The notion of service is sometimes compared to the notion of coverage. The latter is however only related to the physical notion of receiving a radio wave independently from the notion of quality of service (e.g., restriction on time delay and delay spread, that are the average and the standard deviation of the time needed by a message to propagate between the transceiver and the receiver).



(a)

(b)

Figure 5.2: *Three cells computed on the French region "Les Vosges" (a), and in a district of the city of Geneva (b). The black zones represent the served areas.*

that cells can be computed and returned by an *ad hoc* function. In the present case, geographical locations are discretized on a *regular grid*, and the cells are computed by thresholding the output data of radio wave propagation prediction tools[4] such as that presented in previous section. Figures 5.2 shows the shape of cells, computed in the hilly French region "Les Vosges" and in the Swiss urban district of Geneva (in this example, indoor radio wave propagation is not considered).

5.1.3 Examples of instances.

One artificial and three real-life cases are chosen as instances of the transceiver siting problem. They all include a geographical region and a set of potential transceiver sites.

The artificial instance, called *Squares149*, includes a set of 149 dummy potential BTS sites. It is generated as follows: on a 287×287 point grid representing an open-air flat area, 49 square cells are distributed regularly in order to form a 7×7 grid structure. Each of the 41×41 point cells is associated to a BTS. A hundred complementary BTS locations are then randomly selected, associated to new 41×41 point cells (fewer when clipped by the border of the area), and shuffled with the 49 primary ones. By construction, the best solution for this instance is that with the 49 primary BTSs that serves 100% of the area.

The first real-case instance, referenced as *Vosges150*, includes a set of 150 user-provided potential sites. These sites are located on a 5,493 km^2 digital terrain model of the French region "Les Vosges" that is discretized on a 291×302 point grid[5].

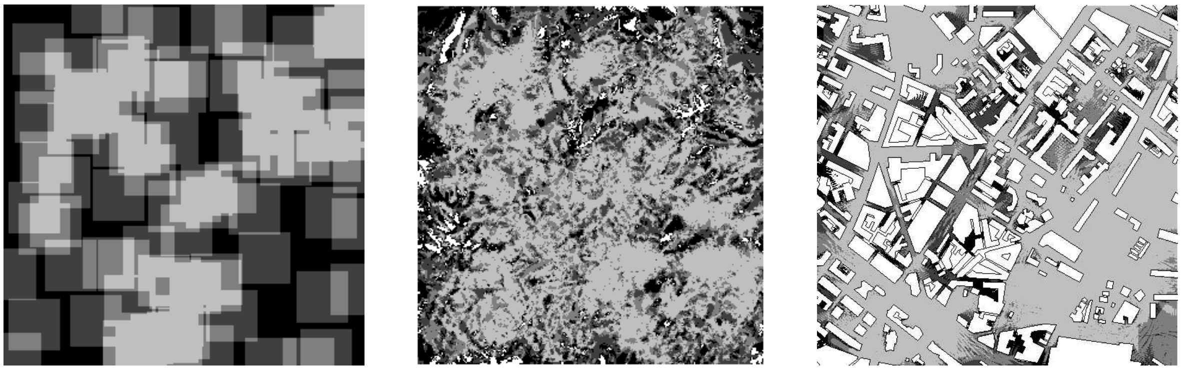
The second real-case instance, referenced as *Geneva99*, includes a set of 99 user-provided potential sites. These sites are located on a 500×500 point zone modeling a

[4]The radio wave propagation prediction software used in rural environment is provided by Télédiffusion de France (France Telecom group).

[5]One pixel represents a 250×250 m square.

1 km^2 district of the Swiss city of Geneva[6].

Figure 5.3 shows the service area that would be obtained if transceivers were installed at every potential site for *Squares149*, *Vosges150*, and *Geneva99*.



(a) (b) (c)

Figure 5.3: *Representation of the service that would be obtained if all the BTSs were installed at every potential site: 149 BTS on the artificial area of the* Squares149 *data set (a), 150 BTSs on the French region "Les Vosges" (b), and 99 in a district of the city of Geneva (c). White locations are not served, black locations are served once, and gray locations are served several times.*

The last real-case instance is used for experimenting on big instance of problems. It is referenced as *Vosges600* and includes 600 potential sites located on a 585×651 point zone modeling a 23,802 km^2 area of the Vosges region.

5.1.4 Modeling of the service.

The relationship between each pixelized location served and the BTSs is naturally modeled as a bipartite graph whose nodes represent either BTSs or geographical locations (pixels) [16]. When many geographical locations must be allowed for, such a graph tends to be huge (see Figure 5.4(a)). A smart way to reduce the graph size without loosing any useful information is to build a bipartite graph whose nodes represent either BTSs or *intercells* [14]. An intercell is defined as the set of geographical locations that are potentially served by exactly the same set of BTSs. For each intercell node, one only needs to encode the *cost* of this intercell, that is the number of locations it contains (see Figure 5.4(b)). The bipartite graph hence obtained can be smaller than the former one by more than one order of magnitude.

The bipartite graph (simple or with intercells) is used to compute the service ratio

[6]One pixel represents a 2×2 m square.

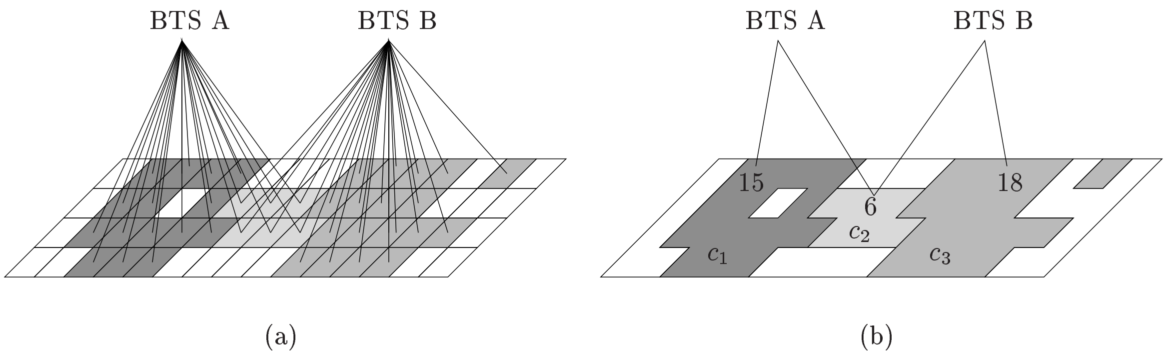


Figure 5.4: *A bipartite graph that models 2 BTSs and their associated cells $cell_A$ and $cell_B$. The pixelized locations can be all represented (a) or they can be gathered in 3 intercells $c_1 = cell_A - cell_B$, $c_2 = cell_A \cap cell_B$, $c_3 = cell_B - cell_A$ (b).*

produced by any subset s of BTSs:

$$\text{service ratio} = \frac{\text{surface of area served by } s}{\text{surface of the maximum served area}} \tag{5.1}$$

where the "maximum served area" is the area that is served when every BTS of the initial set is taken. The algorithm is simple: the adjacent nodes of the BTS nodes in s are visited and their values are summed up, hence the surface of the served area. Such a computation is likely to be done very often in an EA since it must evaluate the quality of all the numerous individuals considered. Moreover, it is clear that the size of the graph influences directly the time needed to compute the service ratio, hence the interest to reduce it.

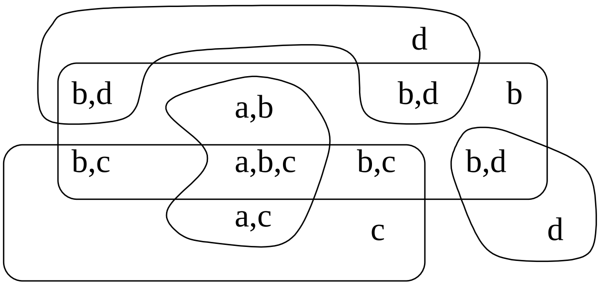
5.1.5 Problem representation using set systems

A *set system* $(X, \Re)$ is a set X of n *elements*, with a collection $\Re$ of m subsets of X called *ranges*. Let us consider a set system $(X, \Re)$ where X is the set of cells and $\Re$ is the set of all *intercells*. Figure 5.5 depicts such a set system. For example, the notation $\{a, b\}$ represents the intercell "covered" exclusively by cell a and cell b (i.e., $(a \cap b) - (c \cup d)$).

A *weighted set system* is a set system in which each range $R \in \Re$ is given a cost c_R (e.g., c_R is the number of pixels contained by the intercell R). For each element $x \in X$, let $x_\Re = \{R \in \Re | x \in R\}$ denote the set of ranges in which x is included. Denote $c(x_\Re) = \sum_{R \in x_\Re} c_R$ and $c(X) = \sum_{x \in X} c(x)$.

5.1.6 Hitting set and set cover problems

A *hitting set* of the set system $(X, \Re)$ is a subset $H \subseteq X$ of cells such that H has a non-empty intersection with every intercell R in $\Re$. Roughly speaking, this means that each pixelized location of the whole area is served by at least one BTS. The transceiver



$$X = \{a, b, c, d\}$$

$$\begin{aligned} \Re = \{&\{b\}, \{c\}, \{d\},\\ &\{a, b\}, \{a, c\}, \{b, c\},\\ &\{b, d\}, \{a, b, c\}\} \end{aligned}$$

Figure 5.5: *A set system of $n = 4$ cells inducing $m = 8$ intercells.*

siting problem recalls the minimum *Hitting Set Problem* (HSP) whose NP-completeness (shown by Karp [65]) dates back the early seventies. However, it slightly differs from minimum HSP because the goal is to select a satisfactory subset of BTSs that ensures a service in *almost* all the area. This means that non-combinatorial parameters such as the target service ratio ($tsr \in [0, 1]$) are also to be taken into account in practice. tsr expresses the ratio of the area that is targeted to be served over the maximum area that can be served. For each cell $x \in X$, let $x_\Re = \{R \in \Re | x \in R\}$ denote the set of intercells in which x is included. Let $X_\Re = \{x_\Re | x \in X\}$ be the set of groups of all intercells. A *covering set* of a set system $(\Re, X_\Re)$ is a subset of $X'_\Re \subseteq X_\Re$ such that $\cup_{x_\Re \in X'_\Re} = \Re$. A *minimum set cover problem (SCP)* asks for a minimum-size covering set $X_{\Re min}$ such that $|X_{\Re min}| = \min(|X'_\Re|, X'_\Re \subseteq X_\Re$ and $\cup_{x_\Re \in X'_\Re} = \Re)$. The set system $(\Re, X_\Re)$ is said to be the *dual* of $(X, \Re)$ since a solution to HSP implies a solution to SCP and vice-versa. Denote by k-HSP the HSP where each range has at most k BTSs and by k-SCP the SCP where each cell covers at most k intercells.

The dimension of the search space is 2^ν, where ν is the number of potential BTS sites. In order to have an idea of the order of magnitude it implies, let us take an initial set that contains 150 potential sites, the size of the search space is then $2^{150} \simeq 10^{45}$. It means that, if checking a candidate takes 1 millisecond then 10^{34} years are necessary to check all of them. Even if, for example, only the solutions with 60 BTSs are considered, 10^{32} years[7] are still necessary to check all of them. In comparison, the Universe is "only" 12.10^9 years old according to current theory! This illustrates the impossibility of enumerating the search space of an instance of the transceiver siting problem, even with only 150 potential BTS sites. Different heuristics are considered in the following sections in order to treat this problem.

5.2 Greedy-like algorithms

A simple *greedy-like algorithm* was used as a basis for the comparison with the different parallel EAs that were experimented. Introduced by Chvátal [21] and thoroughly ana-

[7]The binomial coefficient $\binom{150}{60} = 5.10^{42}$ gives the number of solutions with 60 BTSs taken among 150 potential ones, hence $= 10^{32}$ years to check every solution if the evaluation of each of them takes 1 millisecond.

lyzed by Slavík [90], the natural greedy-like heuristic achieves a *performance ratio*[8] of $\log k - \log\log k + \Theta(1)$, where k is the maximum number of intercells lying in a BTS. Note that if $k \leq 2$ then SCP is equivalent to the *Edge Cover* problem and therefore can be solved in $O(n\sqrt{m})$-time using a *maximum matching* in a bipartite graph [77]. This approach was used to improve the greedy-like heuristic and k-HSP [35]. Recently, Feige [36] proved that unless $NP \subseteq DTIME[n^{\log\log n}]$ there is no polynomial-time algorithm that guarantees a $(1-\epsilon)\log k$ performance ratio. This plainly explains the intractability of HSP, and dually of SCP, from both the theoretical and practical point of view.

The *partial hitting set cover problem (PHSP)* consists in hitting, with as few cells as possible, at least $|\Re|r$ intercells for a given hitting ratio $r \in [0,1]$. PHSP has also been proven NP-complete by Kearns [66] as soon as $0 < r \leq 1$. Kearns used a variant of the greedy-like heuristic with performance ratio $2H(m) + 3$, where $H(c) = \sum_{i=1}^{c} \frac{1}{i} \leq \log c + 1$ is the c^{th} *harmonic number*. Slavík [89] lowered the performance ratio to $\min\{H(\lceil rm \rceil), H(k)\}$.

The greedy-like heuristic can be naturally extended to weighted set systems and runs in $O(nm)$ time and space for dense (resp. $O(n\log n)$ for sparse, i.e., $m = O(n)$) set systems as shown below (See Algorithm 11). There exists various extensions of HSP that lead to different heuristics and hardness results. Refer to [83] for an up-to-date survey.

Algorithm 11
(∗ Greedy ∗)
(∗ Implements Kearn's greedy-like heuristic ∗)
(∗ *tsr* is the target service ratio ∗)
(∗ *rest* is the surface yet to be served in order to obtain a *tsr*-service ∗)
1. $X' \leftarrow \emptyset$; $rest \leftarrow tsr$
(∗ Initialize the current solution X' with 0 BTS ∗)
2. **while** less than tsr of the surface is served by X' **do**
3. Add x_i (the i^{th} BTS) to X' such that x_i maximizes $\min(rest, c(x_{i\Re} \backslash X'_{\Re}))$
4. Update $rest \leftarrow tsr - \sum_{x\in X'} c(x)$

5.3 Experimental conditions

5.3.1 Network configuration for speed-up measurements

The network used for the speed-up measurements contains 10 groups of 9 Sparc-4 workstations (with 64 MB of RAM). They are connected by one Ethernet HUB within each group. The groups are then connected to a Sparc-1000E server (with 4 processors and 256 MB of RAM) by FDDI via a HUB-FDDI. The use of any other computer for experiments will be clearly mentioned. The experiments were made at night and during

[8]The performance ratio is the ratio between the value of a given solution and any optimal solution.

week-ends in order to minimize disruptions of the network (e.g., the side effects of its use by many other users and system management routines). The programs were executed several times (from two to ten times depending on the experiment[9]) but only the minimum time was eventually considered for the speed-up computation.

5.3.2 Influence of islands on execution time

As explained in [16], when a step of a GA has a complexity of $\Theta(n^2)$ (e.g., when the selection step is based on a roulette wheel), the number of operations achieved during this step is proportional to the square of the number of individuals per island. Actually, when the number of islands is doubled, the number of individuals per island is divided by 2 and the computation load of the selection step is divided by $2^2 = 4$. Consequently, the execution of an IGA with 2 islands of n individuals is quicker than that of a GA with a population of $2 \times n$ individuals. The time factor gained is at most 2.

Figure 5.6 shows the time factor that can be achieved (without any parallelization) on a single PE by simply distributing a fixed number of individuals on islands.

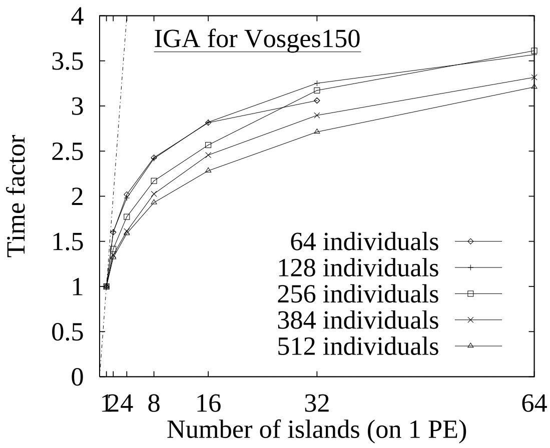


Figure 5.6: *Factor gained (in terms of time) when a given number of individuals is distributed on islands (the program is sequential and runs on a single PE).*

Let us suppose that the number of islands increases with the number of PEs during the speed-up measurement of a parallel EA. The speed-up hence achieved is not only due to the parallelization, but also to the time factor gained by increasing the number of islands. The speed-up achieved is thus artificially super-linear or at least excellent (i.e., near-linear).

The notion of island is purely algorithmic and the use of islands should not influence speed-up measurements on several PEs. Consequently, the number of islands is kept constant during each speed-up measurement presented in this work (whatever the number

[9] Each measurement was done ten times for the first experiments, but since the disruptions of the network were low it turned out that punctual checks were sufficient.

of PEs is). Besides, the use of islands is always clearly distinguished from that of parallel EAs. This choice is coherent with the parallelization rules – introduced the previous chapter – that suppose that the parallelization does not change the behavior of the original sequential algorithm (cf. page 52).

The former precision is necessary because the way to compute the speed-up of parallel EAs has raised significant controversy in the GA community [20]. Since the standard GA is often taken as a basis in many studies, the following cases can be found in the GA literature: in [99] the speed-up compares the sequential time of a standard GA to the parallel time of an IGA with p islands distributed on p PEs (by keeping the total number of individuals constant), and in [20] the number of islands together with the total number of individuals changes with the number of PEs. In the latter case the aim is to have the same expected quality of solution with both the sequential GA and the parallel IGA. These situations are only possible because of the lack of rigor in the definitions of the speed-up enumerated in 2.4.3. In this work, the speed-up is computed by measuring the sequential and the parallel time of a same program parameterized by the number of PEs (as stated page 23). It is therefore possible to estimate the speed-up gained to obtain the same results in sequential and in parallel.

It can be observed in Figure 5.6 that, as expected, the factor gained when doubling the number of islands is within $[1, 2]$. The time factor is actually at most 1.6. This can be explained by two phenomena:

- Only one part of the algorithm has a complexity of $\Theta(n^2)$, while the remainder has a lower complexity. Thus, only one part of the algorithm contributes to gain time.

- The amount of time required by the migration step increases with the number of islands, and this computational overhead minimizes the time factor gained.

The second phenomenon can also explain why the time factor decreases when the number of islands increases.

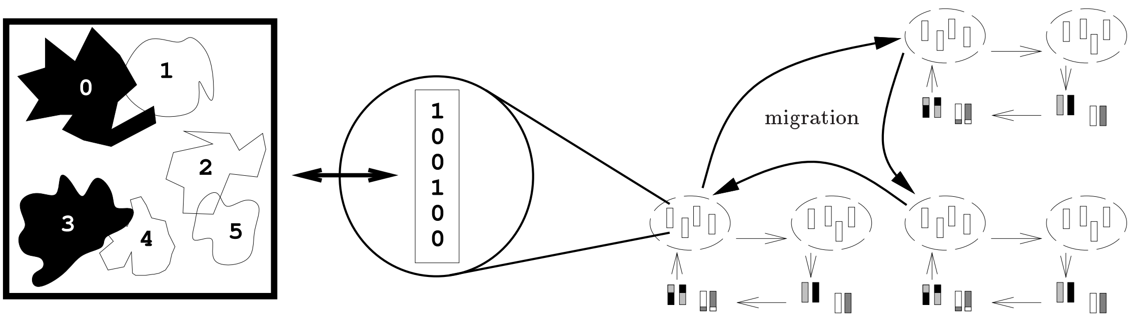
All this discussion only deals with GAs. It can however apply to any EA that has at least a step with a complexity higher or equal to $\Theta(n^2)$.

5.3.3 The choice of the number of generations

It was tried to compute speed-ups by measuring sequential time that do not exceed 10 hours. This could be done by reducing the number of generations when the execution time of an EA was too long. Consequently, the number of generations remains constant during each speed-up measurement, but it can vary from an experiment to another. It was however checked that the number of generations be large enough to provide accurate measurements. Since this parameter has no influence on the speed-up measurements, it is not mentioned in the remainder (except in the sections dealt with the quality of the solutions obtained).

5.4 Parallel island-based genetic algorithms

The IGA used here does not rely directly on the notion of set system but it still uses the bipartite graph introduced in 5.1.4 to compute the service ratio.



(a) (b) (c)

Figure 5.7: *(a) A problem instance with 6 potential BTSs is proposed here. An example of a candidate with two BTSs (associated to black cells) is represented on a dummy region. (b) The individual that models this candidate is encoded by a bit-string. (c) Three populations of such individuals evolve according to the four phases of a genetic algorithm (cf Section 2.3.1).*

A generational replacement GA was developed to solve the radio coverage problem introduced in Section 5.1. In the implementation, the chromosome-like bit string represents the whole set of possible BTS locations. Whether a location is actually selected in a potential solution depends on the value of the corresponding entry in the bit string (see Figure 5.7(a) and 5.7(b)).

The individuals that form the initial population are generated randomly. The selection operator is implemented as a roulette wheel selection: the population is mapped onto a biased roulette wheel where each individual is represented by a slot sized in proportion to its fitness. The wheel is then repeatedly spun to select the individuals to be put in the intermediate population, achieving in this way a stochastic sampling with replacement. The crossover and mutation operators are the one-point crossover and the mutation operator described in 2.3.1 with a 60% probability to be applied at an individual at each generation[10] (i.e., $p_c = p_m = 0.6$). The execution terminates after a predefined number of generations without any consideration of the convergence speed of the algorithm. This permits to change some parameters (e.g., the number of islands) without changing the number of generations computed and thus to fairly compare the execution time of the GA with different parameter tunings.

The following objective function is taken as the fitness function in order to assign a

[10]The role of this parameter is discussed in the last section of this chapter (page 102).

fitness value to each individual:

$$f = \frac{(\text{service ratio})^{\gamma}}{\text{number of BTSs used}} \tag{5.2}$$

where γ is a parameter that can be tuned to favor the service ratio with respect to the number of BTSs used. For example, Figure 5.8 shows the influence of this parameter on the characteristics of the results returned by the IGA for the *Vosges150* data set[11].

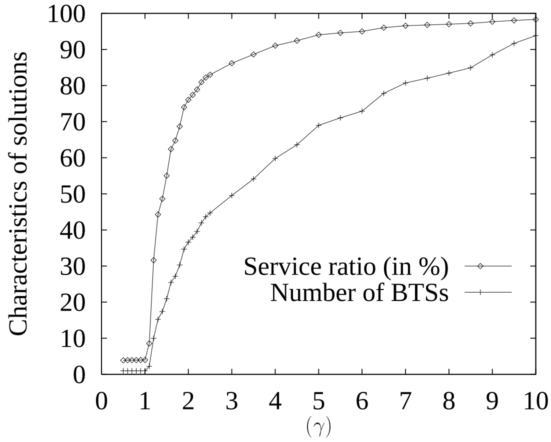


Figure 5.8: *Influence of the value of γ on the characteristics of the solution found (number of BTSs used and service ratio obtained). The test was done with the* Vosges150 *data (cf. page 87).*

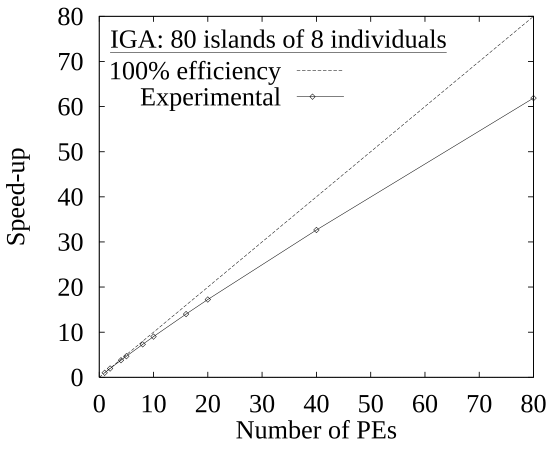
It shows that:

- for $0.5 < \gamma \leq 1$ only one BTS is chosen (that with the largest cell),
- for $1 < \gamma \leq 1.5$ the coverage is not acceptable ($< 55\%$),
- for $1.5 < \gamma \leq 4$ the coverage grows quickly while the number of BTSs grows slowly,
- for $4 < \gamma \leq 10$ the coverage remains almost constant while the number of BTSs keeps growing.

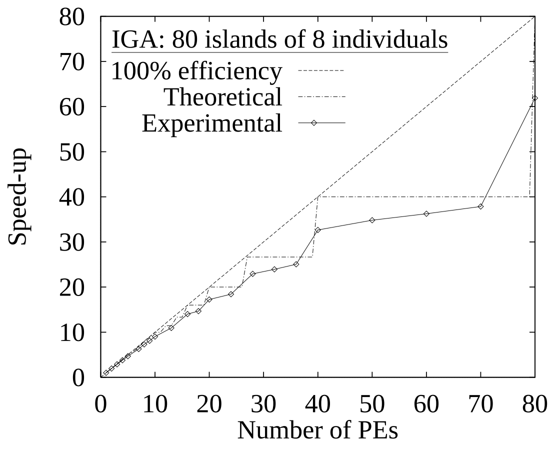
A good value for γ should thus be chosen in $]1.5, 4]$. When $\gamma = 4$, solutions returned by the IGA give around 90% of service ratio, which is considered as a satisfactory result by telecommunication specialists. This value will thus be used in the remainder of the tests.

The speed-up achieved by the IGA with 640 individuals distributed on 80 islands for the *Vosges600* data set is shown in Figure 5.9. A very good 74.4% efficiency is observed on 80 PEs. Figures 5.9(a) and 5.9(b) show the speed-up for the same experiment: in the first figure, only the numbers of PEs that are divisors of the number of islands are considered (i.e., $2, 4, 5, 8, 10, 16, 20, 40, 80$ PEs for 80 islands), whereas in the second figure any number of PEs is considered. The number of islands is not necessarily the same on

[11] A similar graph (not shown here) was obtained with a greedy-like algorithm.



(a)



(b)

Figure 5.9: *Speed-up achieved with 640 individuals distributed on 80 islands for the* Vosges600 *data set. (a) Only the numbers of PEs p that result in a fair load balancing are considered (i.e.,* $80 \bmod p = 0$*), hence a near-linear graph. (b) The number of PEs is arbitrary, hence a graph with steps. The theoretical graph is based on Equation 4.11 with* $I = 80$*.*

each PE (as explained in 4.2.1), and the PEs with the most islands bound the speed-up that can be achieved, hence a speed-up graph with steps.

Figure 5.9 shows that a graph is not necessarily an objective representation of the reality, and that its interpretation must be done carefully with respect to the experimental framework.

Figure 5.9(b) compares the experimental speed-up to the theoretical one based on Equation 4.11 (assuming a null communication cost). Each time $\left\lceil \frac{80}{p} \right\rceil \neq \left\lceil \frac{80}{p+1} \right\rceil$ there is a step on the theoretical graph. Three phenomena explain why the steps of the theoretical graph are less pronounced in the experimental one:

- When the number of PEs increases, the computation time of each PE decreases while the communication time remains constant (one individual migrates from each PE)[12]. Thus, the ratio $\frac{t_{\text{communication}}}{t_{\text{computation}}}$ increases and the speed-up is lowered accordingly.
- The speed-up points could not be measured for every number of PEs[13], hence a gentle transition from one step to another (this explains why the experimental graph sometimes crosses the theoretical one, even if experimental points are always under the theoretical graph).

[12]The total amount of communications increases but since these communications occur simultaneously, the communication time remains constant.

[13]Because of the unavailability of the network.

- When the number of islands I is not a multiple of the number of PEs p, then the PEs that handle $\left\lfloor \frac{I}{p} \right\rfloor$ islands can communicate while the others (with $\left\lceil \frac{I}{p} \right\rceil$ islands) end their computation. The apparent communication load is thus limited to that of the PEs with the most islands. Thus, when p increases the speed-up also increases, hence the non-flat steps observed in Figure 5.9(b).

Steps can however be identified on the experimental graph, and they fit well the theoretical steps (i.e., their position in terms of number of PEs is coherent).

Figure 5.10 shows that the speed-up is a little bit higher when a large instance of problem is treated. When running the IGA in parallel, a speed-up of up to 39.4 (resp. 45.4) was observed on 80 workstations with 320 individuals distributed on 80 islands for the *Vosges150* (resp. *Vosges600*) data set. Figure 5.10 also shows that the speed-up is on average the same for any number of islands. However, for a given number of islands, the highest speed-up is always achieved with one island per PE.

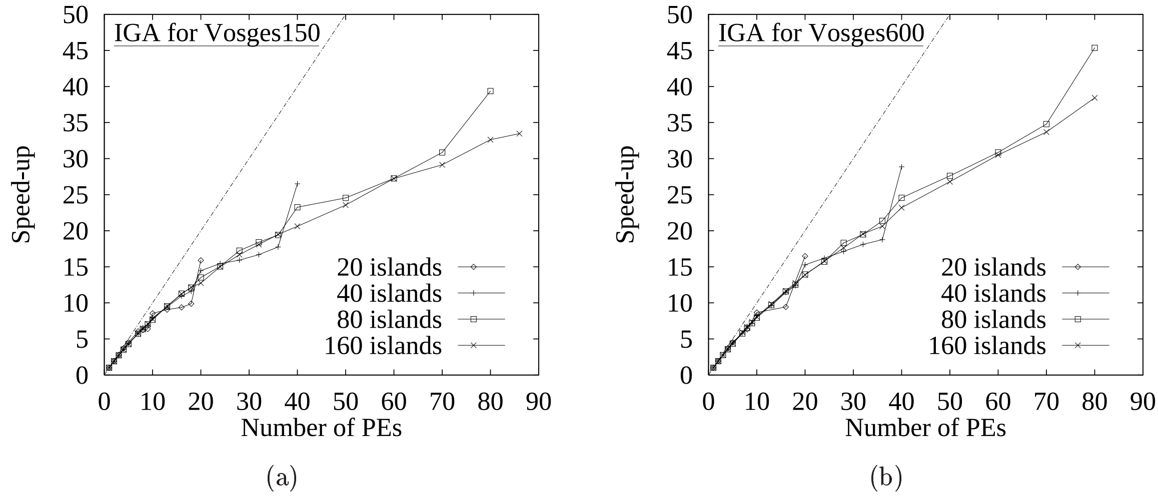


Figure 5.10: *Speed-ups achieved with 320 individuals distributed on 20 to 160 islands for* Vosges150 *(a) and* Vosges600 *(b).*

This experiment shows that, when solving the transceiver siting problem with the IGA, it is always profitable to use the maximum number of usable PEs, that is, the same number of PEs as that of islands.

5.5 Parallel island-based ant systems

The parallel island-based ant system used here was described in 4.2.2. It is based on Algorithm 3. In the context of the *transceiver siting problem*, the notion of transition (as defined in 2.3.4) is not considered since the probability of choosing a BTS does not depend on the choice of the last BTS. However, this probability still depends on the partial solution represented by the ant under construction: only *available* BTSs (i.e., that are not part of the candidate modeled by an ant under construction) have a non-null probability to be chosen. Hence, the probability p_j of adding a BTS j to an ant that is being constructed replaces the transition probability p_{ij} as defined in Equations 2.2 and 2.1 (page 15):

$$p_j = \frac{\tau_j^\alpha \times \eta_j^\beta}{\sum_{l(\text{available BTS})}(\tau_l^\alpha \times \eta_l^\beta)} \text{ if BTS } j \text{ is available, } p_j = 0 \text{ otherwise} \tag{5.3}$$

$$\text{with } \tau_j(t+n) = \rho \cdot \tau_j(t) + \Delta\tau_j(t, t+n) \tag{5.4}$$

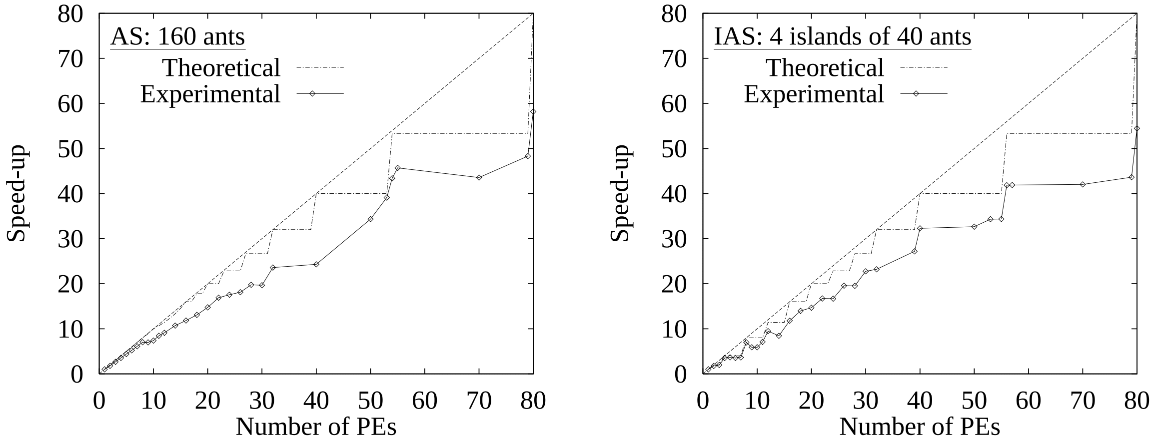
$$\text{and } \Delta\tau_j = \frac{\Delta\tau_j^*}{\sum_l \Delta\tau_l^*} \text{ with } \Delta\tau_j^* = \sum_k \Delta\tau_j^k \tag{5.5}$$

The speed-ups achieved experimentally with one colony of 160 ants, with 4 islands of 40 ants, and with 40 islands of 4 ants are respectively shown in Figures 5.11(a), 5.11(b) and 5.12(a) for the *Vosges150* data set. The theoretical speed-ups that would be achieved if the cost of the communications was null and if every island had exactly the same computational load is also shown in each figure. Figure 5.12(b) compares these theoretical speed-ups. It shows the different width of the speed-up steps with respect to the number of islands used. It thus illustrates the different shapes of the experimental speed-ups. It can be noted that the experimental graphs fit the theoretical ones. The efficiency on 80 workstations is 73% with one colony and 68% with 4 or 40 islands. This efficiency can be considered as very good, considering that two models of parallelization coexist[14], and that a large number of workstations (not dedicated to high performance computing) are used.

The speed-ups achieved with one colony of 160 ants and with 4 islands of 40 ants were measured again for every number of PEs up to 32 (see Figure 5.13). Since the measurements required to compute these speed-ups for any number of workstations are very time consuming, a network of Sparc-20 workstations was used. Moreover this permits to check the behavior of the parallel AS and IAS on a network with different characteristics (Sparc-20 processors are about 5 times faster than Sparc-4 processors, while the network links are the same as in the Sparc-4 network[15]).

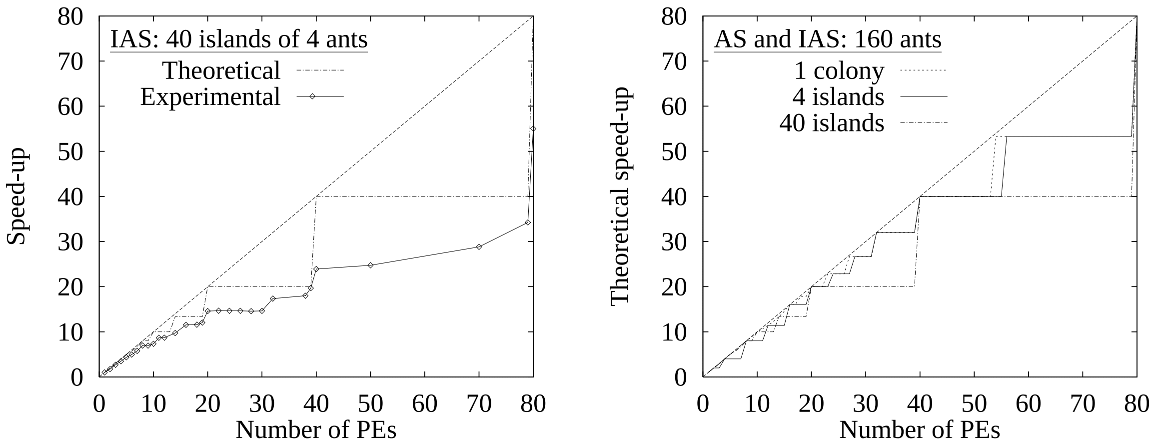
[14]The farmer/worker model used to update trails coexists with the migration of ants between remote islands.

[15]Cf. Section 5.3.1.



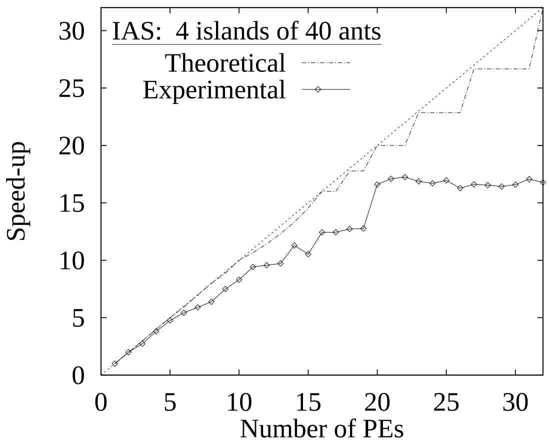
(a) (b)

Figure 5.11: *Speed-ups achieved with 160 ants in one colony (a) and distributed on 4 islands (b) for* Vosges150. *They are compared to the theoretical speed-ups assuming a null communication cost (cf. Equations 4.11 and 4.13).*

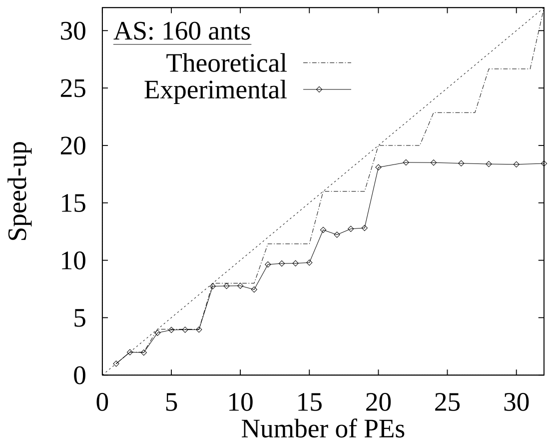


(a) (b)

Figure 5.12: *(a) Speed-up achieved with 160 ants distributed on 40 islands for the* Vosges150 *data set. (b) Comparison of the theoretical speed-ups assuming a null communication cost (cf. Equations 4.11 and 4.13) for different number of islands.*



(a)



(b)

Figure 5.13: *Speed-ups achieved with 160 ants in one colony (a) and distributed on 4 islands (b) for* Vosges150.

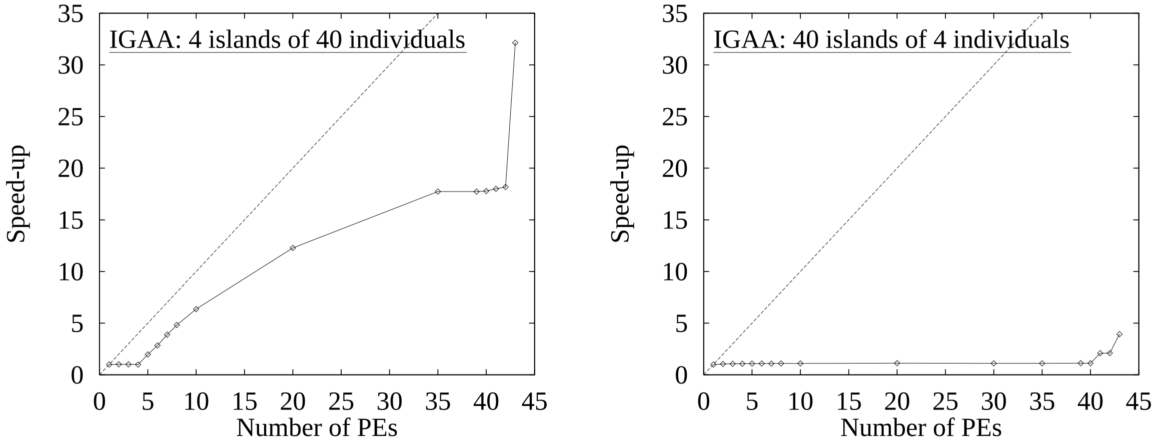
In Figure 5.13(a), the speed-up remains almost constant[16] between 20 and 32 PEs, whereas in Figure 5.11(a) it increases of 60% in the same range. The same observation can be made when comparing Figure 5.13(b) and Figure 5.11(b). The "flattening" of the speed-up graph can be explained by the use of faster workstations in the second experiment than in the first one. Indeed, if the computation part of the algorithm is performed quicker while the communication load is the same, then the ratio $\frac{t_{\text{communication}}}{t_{\text{computation}}}$ increases and the speed-up reaches its maximum sooner.

5.6 Parallel island-based genetic ant algorithm

Figure 5.14 shows the speed-ups achieved by the parallel IGAA based on Algorithm 8 (cf. Section 4.2.3) with 160 individuals distributed on islands. With 4 islands of 40 individuals each, the efficiency is good (75% on 43 workstations), but with 40 islands of 4 individuals each the efficiency is rather poor compared to those of the tests made with IGA and IAS (only 9% on 43 workstations and 3% on 40 workstations). These results can be explained by a first observation: an AS generation is 400 times slower to be processed than a GA generation with the *Vosges150* data set[17]. Between I and $I+n-1$ PEs, the speed-up is thus mainly that of an AS. Since the PE with the ant island bounds the execution time, the speed-up is negligible when the number of PEs is less than the number of islands.

[16] The speed-up varies within $[16.4, 17.2]$.

[17] This estimation was made after comparing the execution time of the AS and the GA run in sequential during 50 generations with the *Vosges150* data set.



(a) (b)

Figure 5.14: *Speed-ups achieved with 160 individuals distributed on 4 islands (a) and 40 islands (b) for* Vosges150.

When the 160 ants are distributed on 40 islands, the computation of the only ant island represents 91% of the total computation, hence a maximum theoretical speed-up of 1.09 on 40 PEs[18].

From this experiment, it can be deduced that the IGAA should be parallelized by partitioning the ant island without distributing the others, hence a parallelization level $L_1(I-1)//L_0\left(\left[\frac{n}{p-I+1}\right]\right)$ instead of $L_1(1)//L_0\left(\left[\frac{n}{p-I+1}\right]\right)$. The speed-up would then be that of the AS with one colony of n ants. The speed-up shown in Figure 5.14(a) (resp. Figure 5.14(b)) would then start to increase when using 2 PEs instead of 5 (resp. 41). The maximum number of PEs would then be bounded by $n+1$.

This experiment shows the limit of the parallelization rules based exclusively on EA ingredients.

5.7 Quality of the results

This section discusses the quality of the results obtained with the algorithms discussed in this chapter. Although the aim is neither to find the best algorithm, nor to find the best tuning for the present problem, it is interesting to have an idea of the efficiency of these algorithms in terms of quality of solution and in terms of execution time. Indeed, it would be useless to parallelize an algorithm that requires a high computational load in order to find solutions of the same quality as that of a simple greedy-like algorithm that runs 100 times faster.

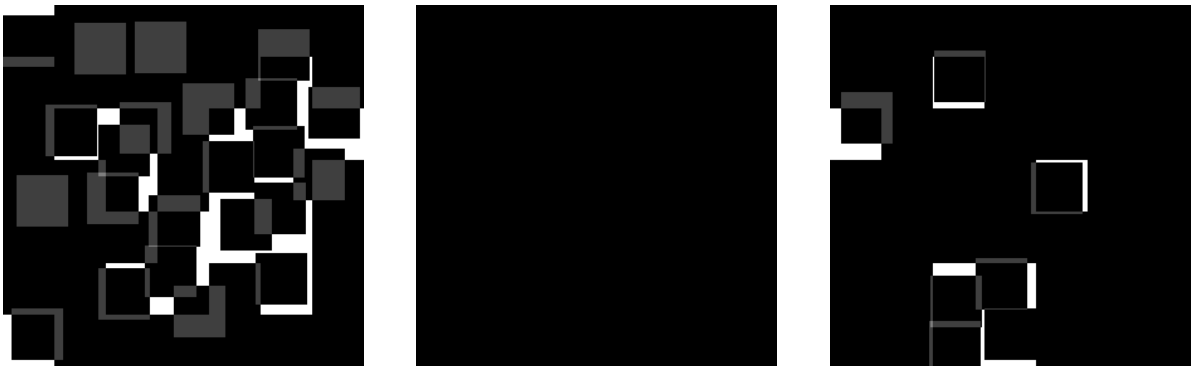
[18]When 40 PEs are used, the ant island is handled by a single PE.

5.7.1 Results

Greedy gives quite good results in the two real cases and does not need to be tuned. However, with the *Squares149* data set it returns a rather bad solution with 55 BTSs for 92.7% of service (see Figure 5.15(a)). The optimal solution has only 49 BTSs for 100% of service.

The IGA runs with 160 individuals that evolve during 320 generations. Experience shows that it gives better results when the probabilities of mutation and crossover are high (typically $p_m, p_c \in [0.6, 0.9]$), but that a finer tuning has insignificant effects.

The results obtained with the IGA for the *Squares149* data set are very good [16]. With 40 islands, the IGA returned sometimes the optimal solution shown in Figure 5.15(b). The rest of the time, solutions were always very close to the optimal one. Figure 5.15(c) shows a typical solution returned by the IGA with 40 islands.

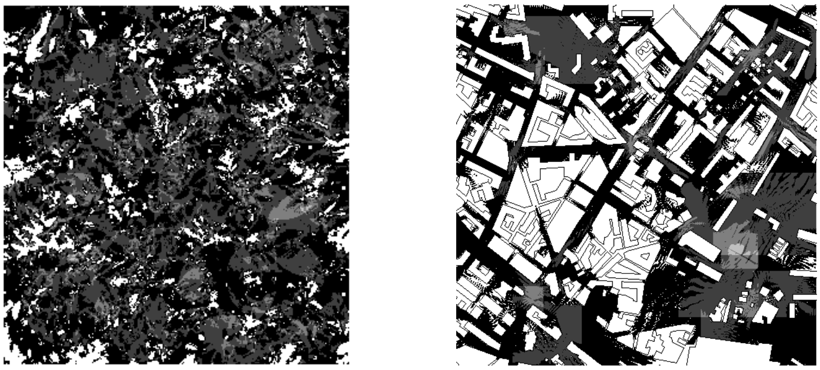


(a) (b) (c)

Figure 5.15: *(a) The solution returned by* Greedy *for the* Squares149 *data set. It has 55 BTS for 92.7% of service. (b) The optimal solution that was sometimes returned by the IGA with 40 islands. (c) Example of a typical solution found by the IGA with 40 islands. It has 49 BTS for 97% of service.*

Figure 5.16(a) and 5.16(b) show examples of the best solutions that were found by IGA configured with 40 islands of 4 individuals. They are typical of the kind of solution returned by every algorithm. It can be noticed that the number of locations that are covered more than once is very small. This side effect is due to the fact that the algorithms tend to minimize the overlaps between cells.

Figure 5.17 shows the characteristics of solutions that were obtained by the different algorithms. A set of randomized solutions was first computed in order to serve as a basis for comparison. The different solutions obtained by IGA are due to several runs with



(a) (b)

Figure 5.16: *Results returned by IGA with 40 islands of 4 individuals each. For a service ratio of 90%, the solutions have: 57 BTSs for the* Vosges150 *data set (a) and 22 BTSs for the* Geneva99 *data set (b). White locations are not served, black locations are served once, and gray locations are served several times.*

different random seeds. Since at each step of GREEDY, a partial solution exists, the evolution of the service ratio can be observed for any number of BTSs selected. It can be noticed that the quality of the solutions returned by the IGA are better than that of GREEDY. The difference is more significant with the *Geneva99* data set. It can be explained by the shape of the cells that are more polygonal, because it was observed that the *Squares149* data set shows the most significant difference between these algorithms.

The solutions returned by the AS and the IAS are in the same quality range: from 67 to 69 BTSs for 92% of service with the *Vosges150* data set and 29 BTSs for 92% of service with the *Geneva99* data set. They are on average not as good as those returned by the IGA with 40 islands, but better than those returned by the GA. Again, experiments were done to have a general idea of the behavior of these algorithms and no fine tuning was tried[19].

GREEDY is 100 times to 10,000 times quicker than EAs and returns on average solutions of the same quality. Yet slightly better solutions can be found by the IGA. This difference, even if small, is very interesting in terms of cost[20] for telecommunication operators. Moreover, the results returned by EAs have a constant quality. Experience shows that when an optimal solution is known, it can be found by the IGA, whereas GREEDY can fall in bad, yet attractive, local optima.

[19]The AS parameters settings are: $\alpha = 1$, $\beta = 1$, $\rho = 0.8$ with 160 ants that evolve during 320 generations.

[20]A BTS is very expensive, and the cost of the installation of a mobile radio network is very high.

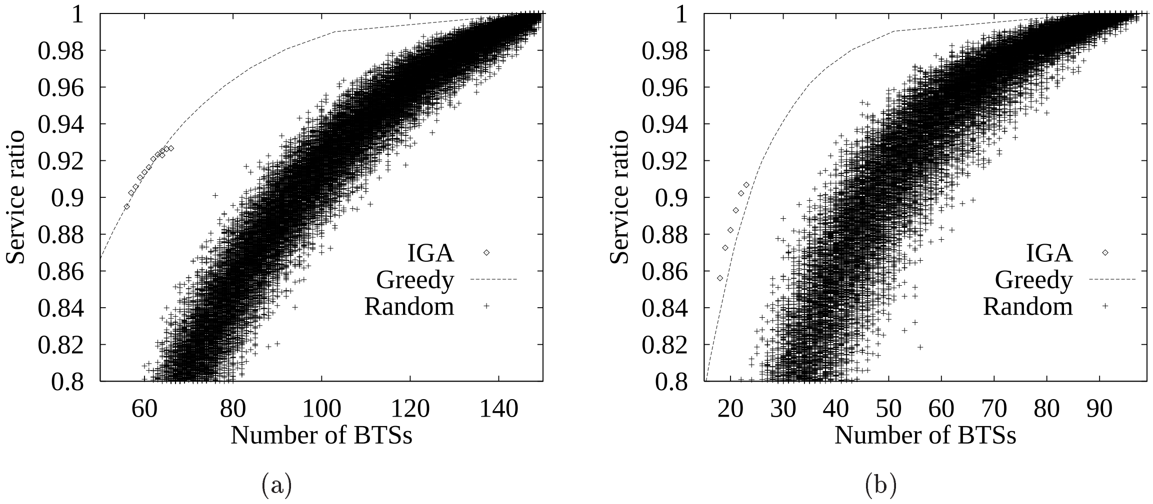


Figure 5.17: *Service ratio against the number of BTSs selected by three algorithms: the IGA,* Greedy*, and at random. (a) for the* Vosges150 *data set, and (b) for the* Geneva99 *data set.*

5.7.2 Influence of islands on the results

In order to observe the influence of the number of islands in the IGA (Algorithm 7), the program was used with only one island. It results in a standard GA (as that of Algorithm 1). An example of convergence speed observation with and without islands, is shown in Figure 5.18(a) (a more detailed study can be found in [16, 18]). It is clear that the IGA converges faster than the GA.

A similar experiment, whose result is shown in Figure 5.18(b), shows the convergence speed of the IAS (Algorithm 9) with and without islands. It can be observed that the behavior of the algorithm is not changed significantly by the use of islands. Even if no improvement of the algorithm can be noticed, as shown above with the IGA, this observation is a positive point in this analysis. Indeed, since the AS and the IAS (with 40 islands) return similar solutions and have a similar speed-up on 80 PEs (see Figures 5.11(a) and 5.12(a)), and since an IAS is more difficult to implement than an AS, it is wiser to parallelize an AS than an IAS (in the present case).

5.7.3 Results of other algorithms

While tackling the transceiver siting problem, some other algorithms, that are not analyzed in details in this thesis, were also tried. Here is a brief description of their results.

A *darwinism algorithm* [17] based on the search of ϵ-nets[21] (see [17] for a description of the algorithm) was studied because of its originality, and because of its similitude with EAs (dozens of potential solutions are generated based on probabilistic rules). However,

[21] If a subset $N \subseteq X$ intersects each set R of $\Re$ of size bigger than $\epsilon * |X|$, then N is called an *ϵ-net*.

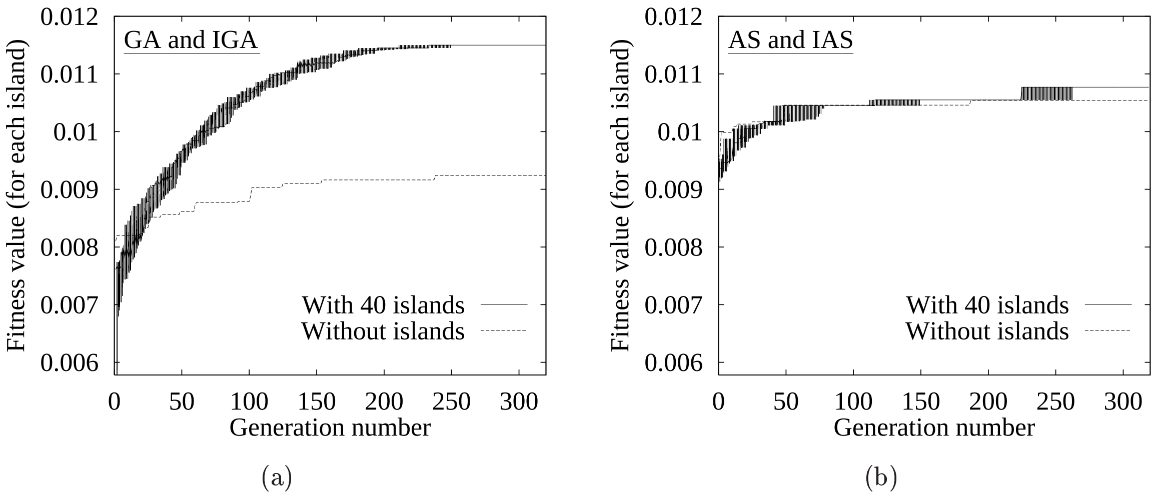


Figure 5.18: *Comparison of the convergence speed, with and without islands, of a genetic algorithm (a) and of an ant system (b) operating on 160 individuals (or ants).*

it was not further used or parallelized because it currently shows poor results.

Greedy was modified in order to be non-deterministic (i.e., in Algorithm 11, x_i was added to X' by following a biased randomized rule instead of maximizing the remaining served zone) and it was run hundreds of time. The results were not as good as those obtained with Greedy.

5.7.4 Summary

The conclusion of these observations is that EAs are not well suited to give quick approximative solutions to the transceiver siting problem. However, considering that:

- a good speed-up can be achieved with a proper parallelization,
- the quality of the solutions returned is constant,
- an increase of the performance is possible, and it can have important consequences (e.g., on the cost of the radio network).
- among the EAs, IGA returned the best results,

the parallel IGA is the most reliable algorithm for solving the transceiver siting problem in an industrial environment.

5.7.5 Cooperation with other projects

ParaGene, the program implementing this parallel IGA, is now used as the main optimization engine in the STORMS[22] radio network optimization software [13, 15]. It is part of a complex computation chain taking advantage of parallelism at several levels. [72] gives more information about this computation chain that integrates UNIX pipe commands for synchronization, farmer/worker programs for pre-computation, a client/server web interface, and ParaGene.

A cooperation was also organized with the project PERFO[23] (Performance modeling of distributed MIMD architectures [81]). ParaGene was used as a test-bench in this project that aimed at predicting performances of irregular parallel programs.

[22] Cf. page 85.

[23] PERFO was funded by the Swiss National Science Foundation (grants $2100 - 042391.94/1$, 1995-98).

To sail is to accept constraints
that we have chosen. It is a privilege.
Mémoires du large, 1997.
Eric Tabarly, sailor (1931–1998)

Chapter 6

Graph coloring application

This chapter describes the classical graph coloring problem, and different programs used to solve it. The programs are based on the parallel EAs described in Section 4.2 and on simple heuristics used for comparison. The following sections elaborate on speed-ups achieved experimentally, and the last one overviews the quality of the results returned by each algorithm.

6.1 Definition of the problem

Let us suppose a non-oriented graph $G(V, E)$ with N vertices $v_i \in V, i \in \{0, 1, \ldots, N-1\}$, and M edges $e_{ij} = (v_i, v_j) = (v_j, v_i) \in E$. $\mathcal{N}(v_i)$ denotes the set of adjacent vertices of v_i in a graph G, that is, $v_j \in \mathcal{N}(v_i) \Leftrightarrow \exists e_{ij} \in E$. Let us take a set of q colors $c \in \{0, 1, \ldots, q-1\}$. When a vertex v is assigned a color c, it is said to be colored with c. It follows that the coloring of a graph is the assignment of a color to each of its vertices. Let us define a conflict as a pair of adjacent vertices that are colored with the same color. And let us define a q-coloring of a graph G as a coloring of G with q colors and no conflict.

The *graph coloring problem* is a NP-hard problem. Here are two different manners to express it:

1. "Given a graph G and $q \in \mathbb{N}$ colors, Find a coloring of G with the minimum number of conflicts?".

2. "What is the minimum number of colors χ such that a χ-coloring of G exists?". $\chi(G)$ is called the *chromatic number* of G.

The first expression is chosen to define the problem that is treated in the remainder of this chapter.

Figure 6.1(b) gives an example of a graph coloring with two colors. In this example, one conflict can be noticed between vertex 4 and vertex 5 (which are both black).

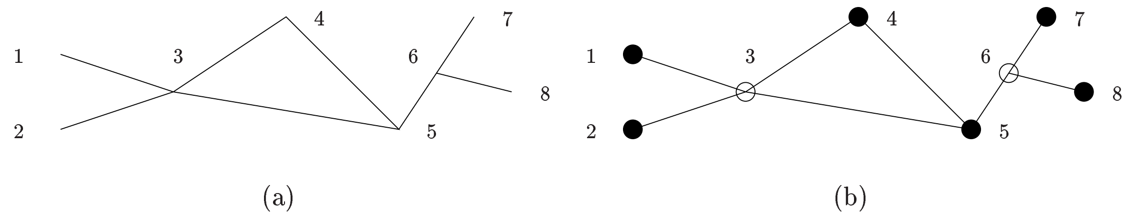


Figure 6.1: *Graph G is colored with two colors. In the solution shown on the right, a conflict remains unsolved between the adjacent vertices 4 and 5; indeed there is no 2-coloring of this graph, $\chi(G) = 3$.*

6.2 Examples of instances

A practical application of this problem is the frequency allocation problem. Let us suppose that each vertex models an antenna, and that G models a radio network such that two vertices of G are linked *if and only if* the corresponding antennas can interfere with each other. Let us suppose now that frequencies are modeled by colors. The problem of allocating a frequency to each antenna such that the amount of interferences is minimum is equivalent to the problem of coloring G such that the number of conflicts is minimum. Since the number of allowed frequencies is usually low, this problem is crucial for telecommunication operators[1]. The algorithms could thus have been tested by taking radio networks planned in the previous chapter as input data. However, they are tested on graphs taken from a test-bench library [100] that provides their minimal q-coloring known. This makes it possible to estimate the quality of the results obtained[2] and thus to have a precise idea of the performance of each algorithm. Even if the quality of the results is not a major concern in this work, such a comparison is instructive on the utility of EAs to solve a graph coloring problem. Results are all discussed in Section 6.7.

Five problem instances of the library [100] were chosen: *G1*, *G2* and *G3* were chosen for their difficulty to be colored by a constructive algorithm while *G4* and *G5* were chosen for their easiness to be colored. *G2* was used for every speed-up analysis while the other graphs were only used in order to compare the quality of the solutions obtained with the different algorithms dealt in this thesis (cf. Section 6.7). The characteristics of the problem instances are:

G1 : 450 vertices, 5714 edges, and an optimal coloring known with 5 colors. It is a Leighton graph [73] (file `le450_5a.col` from C. Morgenstern).

G2 : 450 vertices, 17425 edges, and an optimal coloring known with 25 colors. It is also a Leighton graph [73] (file `le450_25d.col` from C. Morgenstern).

[1]New technologies using a randomized dynamic allocation of frequencies, called *frequency hopping*, tends to solve this problem for mobile phones, but the problem is still actual for other radio networks (television, radio, etc.).

[2]When the chromatic number of these graphs is known, it is given.

G3 : 169 vertices, 6656 edges, and an optimal coloring known with 13 colors. It has a natural interpretation: "is it possible to place 13 sets of 13 queens on a 13×13 chess-board so that no two queens of the same set are in the same row, column or diagonal?" (file `queen13_13.col` from M. Trick).

G4 : 451 vertices, 8691 edges, and an optimal coloring known with 30 colors. It is based on a problem of register allocation for variables in real codes (file `fpsol2.i.2.col` from G. Lewandowski).

G5 : 47 vertices, 236 edges, and an optimal coloring known with 6 colors. This graph is said difficult to color because it is triangle free (file `myciel5.col` from M. Trick). It was however observed during, that this graph was easily colored with the simple greedy-like algorithm used in this work.

The experimental conditions exposed in Section 5.3 are still valid for this chapter. They include the definition of the speed-up and the description of the network of work-stations used for time measurements.

6.3 Greedy-like algorithm

As in the previous chapter, a greedy-like algorithm was used to estimate the difficulty of the instances and the quality of the solutions obtained with the different EAs.

Algorithm 12 is based on a very simple, although efficient in practice, sequential technique that is repeated N times[3]: a not yet colored vertex is chosen at random, it is then colored in such a way that the number of conflicts is minimum (at best it is assigned a color that is not yet assigned to any of its adjacent vertices). On average, such an algorithm colors a graph with a few hundreds of vertices in less than a second (cf. Section 6.7).

Algorithm 12
(∗ COLORING GRAPH GREEDY ∗)
1. **for** each vertex v of G
2.   **for** each color $c \in [0, q-1]$
3. $Set(c) \leftarrow \{v' \in \mathcal{N}(v), color(v') = c\}$
4.   assign color c to vertex v such that $|Set(c)| = \min_{c' \in [0,q-1]} |Set(c')|$

6.4 Parallel island-based genetic algorithms

A candidate to the *coloring problem* is modeled by an individual whose genotype is encoded as a vector with integer components. The size of the genotype (i.e., the vector

[3] N is the number of vertices of a graph G.

length) is the number of vertices of the graph to be colored. Each component of this vector models the color of its associated vertex and it is thus in the range $[0, q-1]$.

This modeling is rather simple compared to some optimized encoding found in the literature [31, 39] but the goal is not here to discover the best coloring graph algorithm. The goal is however twofold: first to confirm the results of the previous chapter in terms of speed-up on a totally different problem, and second to test the flexibility of the software library APPEAL described in Section 4.3.

The speed-up achieved with 40 islands of 4 individuals is shown in Figure 6.2. It is not as good as that previously achieved in the same conditions for the transceiver siting problem with the *Vosges150* data set. It is however similar with an efficiency of 55% on 40 workstations, instead of an efficiency of 66% with *Vosges150* (cf. Figure 5.10(a)).

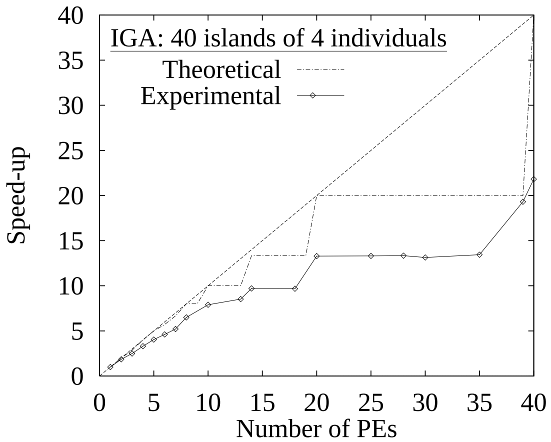


Figure 6.2: *Speed-up achieved with 40 islands of 4 individuals.*

On one hand, the size of an individual is larger for the *G2* instance of the graph coloring problem than for the *Vosges150* instance of the transceiver siting problem. When an individual migrates, about 450 integer values must be sent in the first case instead of 150 boolean values in the latter. On the other hand, the computational load is 7 times lower in the first case than in the latter[4]. The similarity of the speed-ups in these two cases can thus be explained by the increase of both communication and the computational loads (i.e., the ratio $\frac{t_{\text{communication}}}{t_{\text{computation}}}$ remains almost constant).

When p PEs are used, with $p \in [20, 39]$, there must be at least one PE that handles 2 of the 40 islands while the other PEs handle a single island. The speed-up should thus be constant within this range since the execution time is bounded by the PE(s) with the most islands. Yet, it can be noticed in Figure 6.2 that the speed-up achieved with 39 PEs is higher than that achieved with $p \in [20, 35]$ PEs (19.3 instead of 13.3). This can be explained by the ratio of communication time over computation time in this particular case. If 39 PEs are used, only one PE handles 2 islands. The other PEs have

[4] According to experimental measurements.

twice less computation to perform and since no synchronization barrier is required by the algorithm they can communicate before this PE ends its computation. The apparent communication load, limited to that of one PE, is thus negligible. The speed-up achieved is then very close to the theoretical speed-up that does not consider communications (cf. Equation 4.11):

$$\mathrm{S}^{\text{th.},\, p\,\leq\,40}_{40\text{ islands}}(39) = \frac{40}{\left\lceil \frac{40}{39} \right\rceil} = 20. \tag{6.1}$$

This phenomenon is not observed when many PEs handle 2 islands.

6.5 Parallel island-based ant systems

The parallel island-based ant system is based on the ant system of Algorithm 13. There exist several ways to encode the trails τ [31, 39]. For example, it can be stored in a $n \times n$ matrix M where $M(v, v')$ is proportional to the quality of the coloring obtained by giving the same color to the adjacent vertices v and v'. The simplest encoding of τ was chosen here, since the performance of the algorithm is not a priority: τ is simply a $q \times n$ matrix M' where $M'(v, c)$ is proportional to the quality of the coloring obtained by giving the color c to the vertex v.

The constructive method of the AS is based on the greedy-like Algorithm 12. It can be noted that if $\alpha = 0$ and $\beta = 1$ then Algorithm 13 becomes Algorithm 12. In this context, the notion of transition (as defined in 2.3.4) is not considered: the probability $p_{v,c}$ of assigning color c to vertex v depends neither on the last colored vertex, nor on the color used to color it. The probability $p_{v,c}$ replaces thus the transition probability $p_{i,j}$ as defined in Equation 2.2.

The speed-ups achieved with a colony of 80 ants, and with a colony of 160 ants, are shown in Figure 6.3(a). They are equivalent up to 20 PEs but the speed-up achieved with 160 ants continues to increase up to 32 PEs while that achieved with 80 ants does not. After reaching their maximum values, both speed-ups remain constant and finally decrease slowly. When the 160 ants are distributed on 4 islands, the speed-up increases regularly up to 28 PEs and then remains constant with a gentle increasing slope (see Figure 6.3(b)).

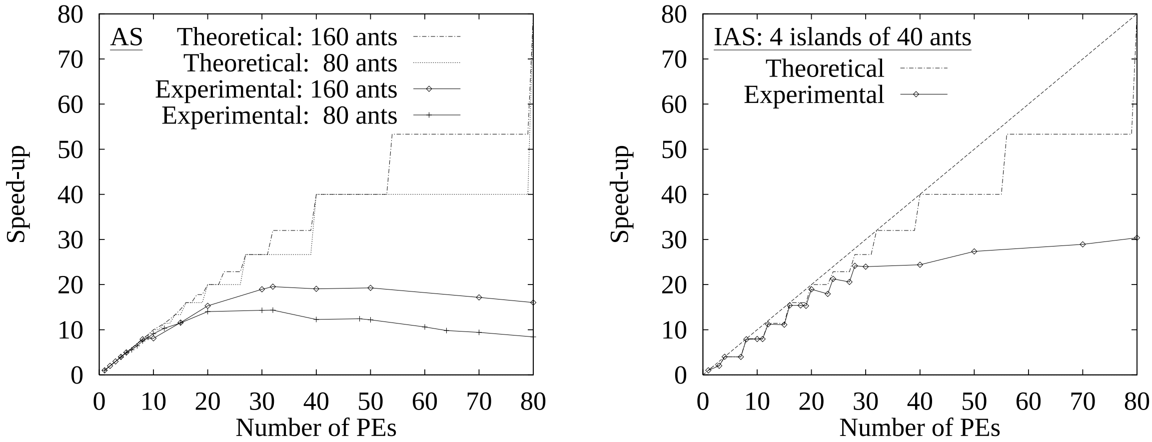
The speed-up achieved here with 160 ants on one colony, and on 4 islands, are very different from those achieved in the same conditions for the transceiver siting problem with the *Vosges150* data set (cf. Figures 5.11(a) and 5.11(b)). The difference can be explained by two phenomena. First, the size of the individuals and the size of the trails are larger than for the previous problem instance: when an individual is exchanged between two PEs, 450 integer values must be sent instead of 150 boolean values; and when a trail matrix is exchanged between a worker and a farmer PE, $25 * 450 = 11250$ real values must be sent instead of 150. Second, the computational load is 1.4 times lower for the *G2*

Algorithm 13

(∗ COLORING GRAPH ANT SYSTEM ∗)

initialize the trails (τ)
cycle $\leftarrow 0$
repeat
  cycle $\leftarrow$ cycle $+1$
  initialize $\Delta\tau$
  for each ant of the colony
    (∗ color G with at most q colors using trails τ and visibility η ∗)
    for each vertex v of G
      for each color c
        if $\exists v' \in \mathcal{N}(v)$ such that $color(v') = c$
          then $\eta_{v,c} \leftarrow 0$
          else $\eta_{v,c} \leftarrow$ (1/number of colors not assigned in $\mathcal{N}(v)$)
        $p_{v,c}(t) \leftarrow \frac{[\tau_{v,c}(t)]^{\alpha}[\eta_{v,c}(t)]^{\beta}}{\sum_{d\in[0,q-1]}[\tau_{v,d}(t)]^{\alpha}[\eta_{v,d}(t)]^{\beta}}$
      assign color c to vertex v with probability $p_{v,c}$
    evaluate $nb_{conflicts}$ the number of conflicts in G
    for each vertex v of G
      $\Delta\tau_{v,color(v)} \leftarrow \Delta\tau_{v,color(v)} + (1/nb_{conflicts})$
  (∗ update the trails ∗)
  $\tau(t+n) \leftarrow \rho \cdot \tau(t) + \Delta\tau(t, t+n)$
until cycle $\geq$ max_cycle

instance than for the *Vosges150* instance[5]. The early stop of increasing of the speed-up shown in Figure 6.3(a) is thus due to a low computational load and a high communication load (i.e., a larger ratio $\frac{t_{\text{communication}}}{t_{\text{computation}}}$).



(a) (b)

Figure 6.3: *(a) Speed-ups achieved with an ant colony of 80 and 160 ants. (b) Speed-ups achieved for 160 ants distributed on 4 islands.*

The speed-up achieved with 160 ants distributed on 40 islands is shown in Figure 6.4(a). When the number p of PEs is lower than the number I of islands, the speed-up fits its theoretical graph. When $p > I$, the speed-up remains good, and it reaches an efficiency of 73% on 80 PEs. Its shape is similar to that obtained in the same conditions for the transceiver siting problem with the *Vosges150* data set (cf. Figure 5.12(a)). The reasons of this similarity are the same than those given page 110 for the IGA. Indeed, in this case the parallelization of the IAS compares with that of the IGA: it is mainly realized by the distribution of low partitioned islands. In opposition, the parallelization of the AS and the IAS with 4 islands is rather based on the partitioning of few islands. The speed-up shown in Figure 6.4(a) is thus less sensitive to the differences between problem instances than the speed-ups shown in Figure 6.3.

It can be observed, once more, that the speed-up becomes higher when there is a low number of PEs with a high computational load (e.g., for $p \in [32, 40]$ and for $p = 79$).

The speed-up achieved with 8 islands of one ant is shown in Figure 6.4(b). It is very similar to the theoretical speed-up computed by Equation 4.11, and it shows an efficiency of 96% on 8 PEs. An equivalent efficiency can be observed on most of the other experimental speed-up graphs with only a low number of PEs (e.g., 8 or 10 PEs). Moreover, when the number of islands is equal to the number of PEs the speed-up is usually good

[5]According to experimental measurements.

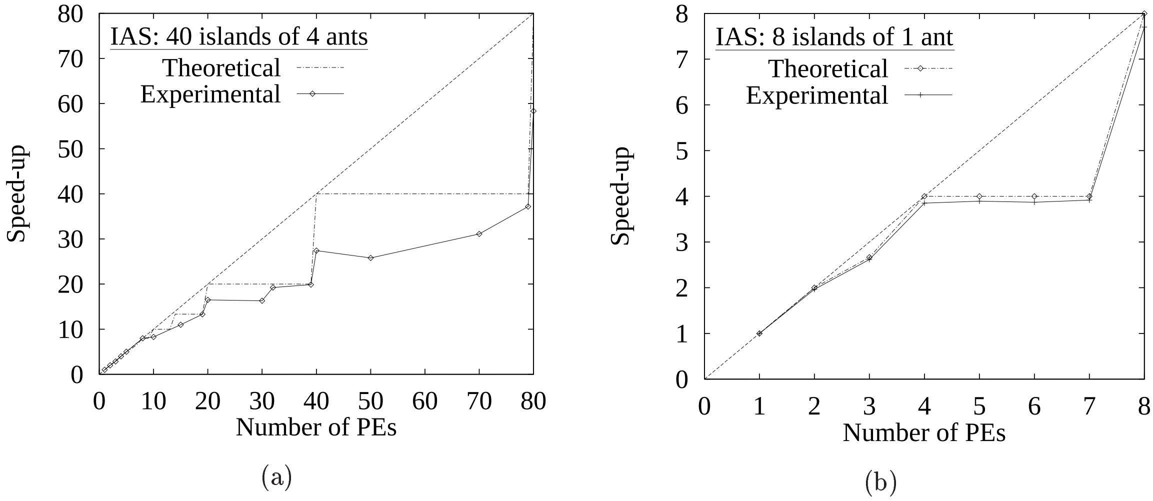


Figure 6.4: *(a) Speed-ups achieved for 160 ants distributed on 40 islands. (b) Speed-up achieved with 8 islands of one ant.*

even when the island size is small (this was previously observed in Figure 5.10). This observation was verified by another experiment with 80 islands of one ant. A very good efficiency of 81% was achieved when the 80 islands were distributed on 80 PEs.

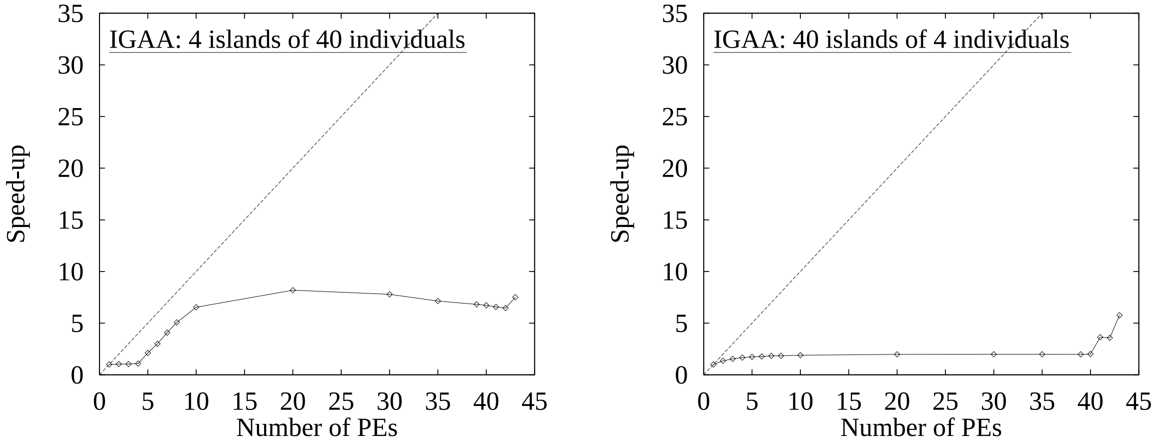
6.6 Parallel island-based genetic ant algorithm

The speed-ups achieved with 160 individuals distributed on 4 and 40 islands are shown in Figure 6.5. The efficiency is rather poor (respectively 17% and 13%) but the speed-ups shown in Figure 6.5 have the same properties than those commented about Figure 5.14 for the transceiver siting problem.

This rather bad efficiency is not surprising since the efficiency of the parallel ant system is also low. Indeed, as said in the previous chapter, the speed-up of this hybrid EA is bounded by the speed-up of its ant island. Since the latter stops to increase when few ants are distributed on each PE, the shape of this speed-up is coherent with that of Figure 6.3(a).

An AS generation is 40 times slower to be processed than a GA generation with graph *G2*. The computation of the only ant island represents thus now 51% of the total computation, hence a maximum theoretical speed-up of 2 on 40 PEs (against 1.09 with the *Vosges150* data set of the transceiver siting problem). This explains why, between 1 and 40 PEs, the speed-up seems to slowly increase in Figure 6.5(b) whereas it looks absolutely constant in Figure 5.14(b).

This confirms that the IGAA should be parallelized by partitioning the ant island



(a) (b)

Figure 6.5: *Speed-ups achieved with 160 ants distributed on 4 islands (a) and 40 islands (b).*

without distributing the others, hence a level of parallelization $L_1(I-1)//L_0\left(\left[\frac{n}{p-1}\right]\right)$ instead of $L_1(1)//L_0\left(\left[\frac{n}{p-I+1}\right]\right)$. The speed-up would then be that of the AS with one colony of n ants (as mentioned in Section 5.6).

This experiment shows the limit of the parallelization rules based exclusively on EA ingredients when heterogeneous meta-heuristics are hybridized. It shows that, in that case, the parallelization rules should take a new factor into account: the ratio of computational load between the different meta-heuristics. This ratio might depend on the characteristics of the problem instances that must be solved (e.g., the time to evaluate the fitness of an individual). It is thus very difficult to estimate it without experimental measurements.

6.7 Quality of the results

6.7.1 Results

Table 6.1(b) compares the quality of the results obtained with the different algorithms discussed in this thesis. The quality of the results is estimated by the number of conflicts that remains in the returned solutions: the less conflicts, the better the solution.

The graphs are those described in Section 6.2. Table 6.1(a) summarizes their characteristics. For each of these graphs noted Gx, an optimal coloring with q_x colors is known. q_x is given to the algorithm as the maximum number of colors that can be used in order to color a graph Gx.

	G1	*G2*	*G3*	*G4*	*G5*
Number of colors given	5	25	13	30	6
Number of vertices	450	450	169	451	47
Number of edges	5714	17425	6656	8691	236

(a)

Number of conflicts		*G1*	*G2*	*G3*	*G4*	*G5*
Random		1126	665	240	273	36
Greedy		1721	783	275	0	0
IGA	1 population	1033	611	178	198	1
	4 islands	878	499	132	137	1
	40 islands	855	486	130	116	2
IAS	1 population	719	211	51	1	1
	4 islands	667	197	50	1	1
	40 islands	653	155	39	1	1
IGAA	4 islands	756	203	53	1	1
	40 islands	672	162	45	1	1

(b)

Table 6.1: *(a) Characteristics of the five graphs that were solved by 10 different algorithms during the experiments. Their complete description is given in Section 6.2. (b) Comparison of the quality (estimated by the number of conflicts) of the results found.*

Random is an algorithm that creates 160 random solutions and returns the best, and *Greedy* is Algorithm 12. The other algorithms are the EAs previously tackled in this thesis. The random and greedy-like algorithms are used to provide a basis of discussion since they give a good idea of the difficulty of each instance.

Each result shown in Table 6.1 is the best that was obtained after 4 runs with different random seeds (except for *Greedy*, that is a deterministic algorithm).

6.7.2 Summary

It can be noted that when *Random* works better than *Greedy* (i.e., with the three first graphs) the EAs work even better. Inversely, when *Greedy* finds the optimal solution – without conflicts – then *Random* shows poor performance. In this latter case, the EAs are deceiving. They are usually close to the optimal but they do not find it. It can thus be stated that EAs are not well suited for solving such instances. Indeed, their execution time is in the order of magnitude of a few hours while that of *Greedy* is in the order of magnitude of the second.

The quality of the solutions increases, or remains constant, when the number of islands increases (except in one case: when *G5* is treated by the IGA with 40 islands). The use of island-based EAs seems thus to be a good choice in general. Among the island-based EAs tried, the IAS always performs better than the IGA. That is not surprising since the AS performs already much better than the GA on the graph coloring problem. The IGAA performs also better than the IGA, but not always as well as the IAS. No benefit of the hybridization can thus be noticed here. In conclusion, the best algorithm that was tested to solve the graph coloring problem is the IAS with 40 islands, but only when *Random* returns better solutions than *Greedy*.

Begin at the beginning and go on
till you come to the end; then stop.
Alice's adventures in Wonderland, 1864.
Lewis Carrol (1832–1898)

Chapter 7

Conclusion

7.1 Summary

First, this thesis gives an overview of NP-hard combinatorial optimization problems, and of heuristic approaches usually applied to solve them. It then focuses on the evolutionary approach that is based on the evolution of a set of candidates, in opposition to traditional heuristics that construct or modify a single candidate. The aim of evolutionary algorithms (EAs) is to avoid the attraction of local optima and to explore a wide range of the search space. Classical evolutionary algorithms (EAs) are enumerated and described in the second chapter. From the observation of the miscellaneous vocabulary used to describe EAs in the literature, a unifying taxonomy is then proposed: the main ingredients of EAs are identified and used to build a classification tool named TEA[1]. The way existing EAs can be characterized using the TEA is illustrated at the end of the third chapter.

Since EAs are highly time and memory consuming, and since they can treat huge instances of problems, the question of their parallelization is then tackled in the remainder. The study focuses on MIMD-DM[2] architectures, mainly because of their increasing availability and their flexibility. Parallel computing concepts are introduced in the second chapter. They are used in the fourth chapter to set parallelization rules based on the TEA description of EAs. An original notation of the granularity of parallel EAs (by level) is proposed. It makes it possible to compare the granularity of parallel EAs.

The main features of EAs and their influence on the parallelization of EAs are now better identified. Taking advantage of this study, an object-oriented software library was designed in order to test the parallelization rules and to facilitate the implementation of parallel EAs. This library, named APPEAL, fulfilled its original aims and also proved the correctness of its object-oriented model. The parallelization rules were experimentally validated for an island-based genetic algorithm (IGA), an ant system (AS), and an island-based ant system (IAS). An island-based hybrid genetic ant algorithm (named IGAA)

[1] TEA stands for Table of Evolutionary Algorithms.
[2] MIMD-DM stands for Multiple Instruction stream, Multiple Data stream, Distributed Memory.

was used to show the limits of these rules. Two test problems were chosen for the study: a realistic problem related to the choice of transceiver sites in mobile phone networks, and a classical graph coloring problem.

The benefit of speeding up the execution of EAs by parallelizing them is highly effective. The speed-ups achieved with the programs written with APPEAL meet the demand, and one of them is now used as the main optimization engine in the STORMS[3] project. This program, named ParaGene, is a parallel island-based genetic algorithm that makes it possible to optimize transceiver siting in interactive mode, as required by telecommunication operators.

Even if the goal is not primarily to study the efficiency of EAs, but rather to study the best way to parallelize them, some observations were made on the quality of their results. Island-based EAs with many islands gave better solutions than standard EAs. This phenomenon was experimentally observed on the two test problems with a GA, an AS, and an IGAA. In some cases, the quality of the solutions was not significantly improved but it was never degraded by the use of islands.

By comparing the results obtained using EAs with the results obtained using simple greedy-like algorithms, three cases were observed.

- Results obtained with EAs are much better in two cases: when solving particular instances of the transceiver siting problem, and when solving some difficult instances of the graph coloring problem.
- Results obtained with greedy-like algorithms are much worse for solving some "easy" instances of the graph coloring problem.
- Results are of the same order of magnitude for the transceivers siting problem, when solving realistic cases.

It was observed that the difficulty of a problem instance is not directly dependent on its size. Instances of problems should thus always be solved first by a traditional heuristic that is simple to implement and fast to execute (such as a greedy-like algorithm). It should then only conditionally be solved by an EA, provided the result is unsatisfactory.

Every time an EA is used advisedly to solve huge instances, its inherent need for important computation power sums up with the large data to be processed. The requirement of efficient parallel versions of EAs in this case justifies the effort put in the present work to parallelize EAs.

In the case of heterogeneous hybrid high-level co-evolution (e.g., an island-based EA with different algorithm on different islands), the parallelization critically depends on the ratio of computational load between the different algorithms involved in the hybrid algorithm. If this ratio is not known in advance, poor results are likely to be obtained (this was illustrated by the poor efficiency achieved by the parallel IGAA in the last chapters).

[3]STORMS stands for Software Tools for the Optimization of Resources in Mobile Systems.

7.2 Major contributions of this work

My major contributions can be summarized in six points. First, I proposed an original taxonomy for evolutionary algorithms associated to classification tool with potential for additional facilities (the TEA). Second, based on this taxonomy, I also proposed a new way to tackle parallel EAs by dissociating their parallel design from algorithmic choices, that is, to distinguish between the search for efficiency in terms of speed-up and in terms of quality of solutions. Third, I developed APPEAL, an object-oriented library dedicated to parallel evolutionary algorithms with a clear design that differentiates three categories of classes: algorithm, problem and candidate encoding classes. Fourth, I measured speed-ups on a large network of workstations (up to 80 Sparc-4) with different programs applied on two different combinatorial optimization problems. Fifth, I studied these speed-ups when the number of islands (or of individuals) is not a multiple, or a divisor, of the number of PEs used concurrently. Finally, I successfully applied a part of this work to the European project STORMS.

7.3 Perspectives

The number of experiments presented in this thesis had to be chosen carefully for the sake of clarity and coherence. It is however very easy to wonder about the results that would have been obtained if this or that hybrid algorithm had been tested, if a given parameter had been tuned differently, if several heterogeneous archipelagi had been handled, or if different computers had been used. Moreover, the number of combinations of algorithms, hybridization and problems that could be experimented is enormous, and it is not realistic to experiment with all these combinations. However, this section proposes several hints about studies that could be complementary to the present work.

Speed-up measurements are very sensitive to the load of computer networks. The results of the present work only apply for a network almost free of any other user. Two interesting complements to this work could be: first, a study of asynchronous EAs; and second, a different approach considering individuals as autonomous agents that control their evolution with their own rules. The efficiency of these parallel EAs could be considered from two orthogonal viewpoints: the speed-up achieved on busy computer networks, and the possible loss in terms of quality of solution.

The extension of APPEAL and the checking of its compatibility with other C++ compilers is planned. The results of the experiments exposed in the two last chapters will serve as a basis for future tests on other parallel platforms. It could also be possible to apply its object-oriented model to another language (e.g., Eiffel [76]).

The conclusions about the two test problems could be generalized to other applications, and thus this work could be reused in a totally different framework (theoretical or industrial).

Some complementary work could also be done by improving the parallelization rules. They are currently rather qualitative and could be enhanced with quantitative conditional rules based on the type of problems treated, on the time to evaluate the fitness of a candidate (with respect to the size of an instance), and the technological characteristics of the parallel computer(s) (e.g., processor speed, link bandwidth, etc.).

Finally, even if the content of the TEA is still somewhat questionable, this new taxonomy can still be refined in the future. It could serve as a sound basis for reasoning about EAs, hence generating a new perception of EAs in the research community.

A good notation has a subtlety and
suggestiveness which at times make it
almost seem like a live teacher.
The World of Mathematics, 1956.
Bertrand Russell (1872-1970)

Appendix A

Glossary and acronyms

A.1 Glossary of usual evolutionary terms

Evolution The *evolution* is the iterative process that is performed by an evolutionary algorithm on a *population*.

Fitness The *fitness value* is a real number that evaluates the quality of the candidate modeled by an *individual*. It is computed by a *fitness function* that is unique for a *population*.

Generation A *generation* is equivalent to one iteration step on an evolutionary algorithm on a *population*.

Genotype A *genotype* is the encoding part of an *individual*. For example, it can be a bit string, a vector of real numbers, a matrix of characters, etc. It encodes the candidate (or the part of the candidate) modeled by an *individual*.

Individual An *individual* is composed of a *genotype* and a *phenotype*. It models a candidate (or a part of a candidate) for a problem. It is always associated to a *fitness value* that represents the quality of the potential solution it models.

Phenotype A *phenotype* is the external vision that one has of an *individual*. For example, it can be a color, a list of valued properties, etc. It is in fact the understandable transcription of the *genotype* of the *individual*.

Population A *population* is a set of *individuals*.

A.2 Frequently used acronyms

Acronyms related to evolutionary computation

ACS	Ant Colony System
AIGA	Asynchronous Island-based Genetic Algorithm
APPEAL	Advanced Parallel Population-based Evolutionary Algorithm Library
AS	Ant System
EA	Evolutionary Algorithm
EC	Evolutionary Computation
ES	Evolution Strategy
GA	Genetic Algorithm
IAS	Island-based Ant System
IGA	Island-based Genetic Algorithm
IGAA	Island-based Genetic Ant Algorithm
PBIL	Population-Based Incremental Learning
PGA	Parallel Genetic Algorithm
PIGA	Parallel Island-based Genetic Algorithm
SC	SCatter search
TEA	Table of Evolutionary Algorithms

Acronyms related to parallel computing

BSP	Bulk-Synchronous Parallel computers
COW	Cluster Of Workstations
DM	Distributed Memory
MIMD	Multiple Instruction stream, Multiple Data stream
MPI	Message Passing Interface
NOW	Network Of Workstations
PE	Processing Element
PRAM	Parallel Random Access Memory
PVM	Parallel Virtual Machine
SIMD	Single Instruction stream, Multiple Data stream
SISD	Single Instruction stream, Single Data stream
SPMD	Single Program, Multiple Data stream

Acronyms related to telecommunication

BTS	Base Transceiver Station
ParaGene	Parallel Genetic software for radio network optimization
ParFlow++	Parallel Flow simulation software
STORMS	Software Tools for the Optimization of Resources in Mobile Systems
UMTS	Universal Mobile Telecommunication System

“Imagine...”
Imagine, 1971.
John Lennon (1940–1980)

Appendix B

Demonstrations

Notations

p is the number of PEs
I is the number of islands
n is the number of individuals per islands

B.1 Theoretical efficiency with indivisible islands

Definitions

Def 1: $\mathrm{E}(p) = \frac{\mathrm{S}(p)}{p}$ (Equation 2.5)
Def 2: $\mathrm{S}^{\text{th.},\, p \leq I}_{I \text{ islands}}(p) = \frac{I}{\left\lceil \frac{I}{p} \right\rceil}$ (Equation 4.11)
Def 3: $\forall x,\ x \leq \lceil x \rceil < x + 1$

Hypothesis

Hyp 1: $p, I \in \mathbb{N}^*,\ p \leq I$ (There are more islands than PEs)

Demonstration

$$\text{Def 1 and Def 2} \Rightarrow \mathrm{E}^{\text{th.},\, p \leq I}_{I \text{ islands}}(p) = \frac{I}{p\left\lceil \frac{I}{p} \right\rceil} \tag{B.1}$$

$$\text{Def 3} \Rightarrow \frac{I}{p} \leq \left\lceil \frac{I}{p} \right\rceil \Rightarrow I \leq p\left\lceil \frac{I}{p} \right\rceil \Rightarrow \frac{I}{p\left\lceil \frac{I}{p} \right\rceil} \leq 1 \overset{(B.1)}{\Rightarrow} \mathrm{E}^{\text{th.},\, p \leq I}_{I \text{ islands}}(p) \leq 1 \tag{B.2}$$

$$\begin{aligned} \text{Def 3} \Rightarrow \left\lceil \frac{I}{p} \right\rceil < \frac{I}{p} + 1 \Rightarrow p\left\lceil \frac{I}{p} \right\rceil < I + p \overset{\text{Hyp 1}}{\Rightarrow} p\left\lceil \frac{I}{p} \right\rceil < 2I \Rightarrow \frac{1}{2} < \frac{I}{p\left\lceil \frac{I}{p} \right\rceil} \\ \overset{(B.1)}{\Rightarrow} \frac{1}{2} < \mathrm{E}^{\text{th.},\, p \leq I}_{I \text{ islands}}(p) \end{aligned} \tag{B.3}$$

Conclusion

(B.2) and (B.3) $\Rightarrow \forall p \in [1, I],\ \frac{1}{2} < \mathrm{E}^{\text{th.},\, p \leq I}_{I \text{ islands}}(p) \leq 1$ (Equation 4.12)

B.2 Theoretical efficiency with partitioned islands

Definitions

Def 1': $\mathrm{E}(p) = \frac{\mathrm{S}(p)}{p}$ (Equation 2.5)
Def 2': $\mathrm{S}^{\text{th.}, p \geq I}_{I \text{ size } n \text{ islands}}(p) = \frac{I \times n}{\left\lceil \frac{n}{\lfloor \frac{p}{I} \rfloor} \right\rceil}$ (Equation 4.13)
Def 3': $\forall x,\ x \leq \lceil x \rceil$
Def 4': $\forall x,\ \lfloor x \rfloor \leq x$

Hypothesis

Hyp 1': $p, I, n \in \mathbb{N}^*,\ I \times n \geq p \geq I$ (There are more PEs than islands)

Demonstration[1]

$$\text{Def 1' and Def 2'} \Rightarrow \mathrm{E}^{\text{th.}, p \geq I}_{I \text{ size } n \text{ islands}}(p) = \frac{I \times n}{p \left\lceil \frac{n}{\lfloor \frac{p}{I} \rfloor} \right\rceil} \tag{B.4}$$

$$\text{Def 4'} \Rightarrow \left\lfloor \frac{p}{I} \right\rfloor \leq \frac{p}{I} \Rightarrow \frac{n}{\frac{p}{I}} \leq \frac{n}{\lfloor \frac{p}{I} \rfloor} \overset{\text{Def 3'}}{\Rightarrow} \frac{n \times I}{p} \leq \left\lceil \frac{n}{\lfloor \frac{p}{I} \rfloor} \right\rceil \Rightarrow \frac{I \times n}{p \left\lceil \frac{n}{\lfloor \frac{p}{I} \rfloor} \right\rceil} \leq 1$$

$$\overset{(B.4)}{\Rightarrow} \mathrm{E}^{\text{th.}, p \geq I}_{I \text{ size } n \text{ islands}}(p) \leq 1 \tag{B.5}$$

Let us take $\alpha = \left\lfloor \frac{p}{I} \right\rfloor$
We have $\alpha \leq \frac{p}{I} < \alpha + 1$

$$\left.\begin{array}{l} \text{Hyp 1'} \Rightarrow p \geq I \Rightarrow \alpha \geq 1 \\ \text{Hyp 1'} \Rightarrow p \leq I \times n \Rightarrow \alpha \leq n \end{array}\right\} \Rightarrow 1 \leq \alpha \leq n$$

$\boxed{\text{If } \alpha = n}$

$$\frac{I \times n}{p \left\lceil \frac{n}{\lfloor \frac{p}{I} \rfloor} \right\rceil} = \frac{I \times n}{p} = 1 > \frac{1}{2} \tag{B.6}$$

$\boxed{\text{If } 1 \leq \alpha \leq n-1}$

$$\frac{1}{2} \overset{?}{<} \frac{I \times n}{p \left\lceil \frac{n}{\lfloor \frac{p}{I} \rfloor} \right\rceil} \Leftrightarrow \underbrace{\frac{p}{I} \left\lceil \frac{n}{\lfloor \frac{p}{I} \rfloor} \right\rceil}_{T} \overset{?}{<} 2n$$

It can be written that $n = \alpha \left\lfloor \frac{n}{\alpha} \right\rfloor + r$ with $r \in \mathbb{N}, r \in [0, \alpha - 1]$
It follows that
- If $r = 0$
  then $\frac{n}{\alpha} = \left\lceil \frac{n}{\alpha} \right\rceil \Rightarrow \frac{n}{\alpha} + \frac{\alpha - 1}{\alpha} > \left\lceil \frac{n}{\alpha} \right\rceil$

[1] Thanks to D. Kobler for his contribution.

- If $r \in [1, \alpha - 1]$
  then $\frac{n}{\alpha} = \lfloor \frac{n}{\alpha} \rfloor + \frac{r}{\alpha} = \lceil \frac{n}{\alpha} \rceil - 1 + \frac{r}{\alpha} \Rightarrow \frac{n}{\alpha} \geq \lceil \frac{n}{\alpha} \rceil - 1 + \frac{1}{\alpha} \Rightarrow \frac{n}{\alpha} + \frac{\alpha-1}{\alpha} \geq \lceil \frac{n}{\alpha} \rceil$
and thus we have always $\frac{n}{\alpha} + \frac{\alpha-1}{\alpha} \geq \lceil \frac{n}{\alpha} \rceil$.

We have thus $T < (\alpha+1) \lceil \frac{n}{\alpha} \rceil < (\alpha+1) \left(\frac{n}{\alpha} + \frac{\alpha-1}{\alpha} \right) = n + \alpha + \frac{n-1}{\alpha}$.
Let us show that $n + \alpha + \frac{n-1}{\alpha} \leq 2n$ in order to prove that $T < 2n$.
It comes to show that $\alpha + \frac{n-1}{\alpha} \leq n$.
Let us take $F(\alpha) = \alpha + \frac{n-1}{\alpha}$
We have $F'(\alpha) = 1 - \frac{n-1}{\alpha^2} \Rightarrow \begin{cases} \text{if } \alpha \leq \sqrt{n-1}, F'(\alpha) \leq 0 \Rightarrow F(\alpha) \text{ decreases} \\ \text{if } \alpha \geq \sqrt{n-1}, F'(\alpha) \geq 0 \Rightarrow F(\alpha) \text{ increases} \end{cases}$
So, for $\alpha \in \{1, \ldots, n-1\}$ the maximum values of $F(\alpha)$ are reached at the bounds.
Since $F(1) = n$ and $F(n-1) = n$, we conclude that $\forall \alpha \in [1, n-1], F(\alpha) \leq n$.
It follows that $T < n + \alpha + \frac{n-1}{\alpha} \leq 2n$.
and thus $\frac{1}{2} < \mathrm{E}^{\text{th.}, \, p \, \geq \, I}_{I \text{ size } n \text{ islands}}(p)$ (B.7)

Conclusion

$$\left.\begin{array}{l} \text{(B.5)} \\ \text{(B.6)} \\ \text{(B.7)} \end{array}\right\} \Rightarrow \forall p \in [I, I \times n], \ \frac{1}{2} < \mathrm{E}^{\text{th.}, \, p \, \geq \, I}_{I \text{ size } n \text{ islands}}(p) \leq 1 \text{ (Equation 4.14)}$$

List of Algorithms

1 Standard genetic algorithm (GA) ... 12
2 Evolution strategy (ES) ... 13
3 Ant system(AS) ... 15
4 PBIL algorithm ... 16
5 Scatter search (SC) ... 17
6 Adaptative memory ... 17
7 Island-based genetic algorithm (IGA) ... 38
8 Island-based genetic ant algorithm (IGAA) ... 47
9 Island-based ant system (IAS) ... 63
10 Asynchronous island-based evolutionary algorithm (AIEA) ... 83
11 Greedy-like algorithm for the transceiver siting problem (Greedy) ... 91
12 Coloring graph greedy-like algorithm ... 109
13 Coloring graph ant system ... 112

Bibliography

[1] G. M. Amdahl. Validity of the single-processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, number 30, pages 483–485. AFIPS Press, 1967.

[2] Th. Bäck, U. Hammel, M. Schütz, H.-P. Schwefel, and J. Sprave. Applications of evolutionary algorithms at the center for applied systems analysis. In J.-A. Désidéri et al., editor, *Computational Methods in Applied Sciences'96*, pages 243–250. Wiley, Chichester, 1996.

[3] Th. Bäck and H.-P. Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, 1:1–23, 1993.

[4] L. Baker and J. S. Bradley. *Parallel Programming.* McGraw-Hill, 1996. ISBN 0-07-912259-0.

[5] S. Baluja and R. Caruana. Removing the genetics from the standard genetic algorithm. In *Proc. 12th International Conference on Machine Learning*, pages 38–46. Morgan Kaufmann, 1995.

[6] A. Beck. Greed is (sometimes) not enough. *American Mathematics Monthly*, 97(4):289–294, 1990.

[7] T. H. Belding. The Distributed Genetic Algorithm Revisited. In L. Eshelman, editor, *Proceeding of the Sixth International Conference on Genetic Algorithms (ICGA)*, pages 114–121. Morgan Kaufmann, 1995.

[8] R. Benzi, S. Succi, and M. Vergassola. The Lattice Boltzmann Equation: Theory and Applications. *Physics Reports*, 222(3):145–197, 1992.

[9] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation.* Prentice-Hall, 1989.

[10] B. Bullnheimer, G. Kotsis, and C. Strauss. Parallelization Strategies for the Ant System. In R. De Leone, A. Murli, P. Pardalos, and G. Toraldo, editors, *High Performance Algorithms and Software in Nonlinear Optimization*, volume 24 of *Applied Optimization*, pages 87–100. Kluwer:Dordrecht, 1998.

[11] P. Calégari. APPEAL manual: Advanced Parallel Population-based Evolutionary Algorithm Library. Technical Report 124, LITH (Computer Science Theory Laboratory), EPFL, CH-1015 Lausanne, Switzerland, August 1999. Available at: http://grip.epfl.ch/APPEAL.

[12] P. Calégari, G. Coray, A. Hertz, D. Kobler, and P. Kuonen. A Taxonomy of Evolutionary Algorithms in Combinatorial Optimization. *Journal of Heuristics*, 5(2):145–158, July 1999. A first version was published in a technical report ORWP97/02 at Swiss Federal Institute of Technology, Lausanne, 1997.

[13] P. Calégari, F. Guidec, and P. Kuonen. A Parallel Genetic Approach to Transceiver Placement Optimisation. In C.-A. Héritier and B. Chopard, editors, *Proceedings of the SIPAR Workshop'96: Parallel and Distributed Systems*, pages 21–24, October 1996.

[14] P. Calégari, F. Guidec, and P. Kuonen. Urban Radio Network Planning for Mobile Phones. *EPFL Supercomputing Review*, 9:4–10, November 1997.

[15] P. Calégari, F. Guidec, P. Kuonen, B. Chamaret, S. Josselin, D. Wagner, and M. Pizarosso. Radio Network Planning with Combinatorial Optimization Algorithms. In Chr. Christensen, editor, *Proceedings of the 1st ACTS Mobile Telecommunications Summit 96*, volume 2, pages 707–713, November 1996.

[16] P. Calégari, F. Guidec, P. Kuonen, and D. Kobler. Parallel Island-Based Genetic Algorithm for Radio Network Design. *Journal of Parallel and Distributed Computing (JPDC): special issue on Parallel Evolutionary Computing, Academic Press*, 47(1):86–90, November 1997.

[17] P. Calégari, F. Guidec, P. Kuonen, and F. Nielsen. Combinatorial optimization algorithms for radio network planning. *Theoretical Computer Science (TCS)*, 1999. (in print).

[18] P. Calégari, P. Kuonen, F. Guidec, and D. Wagner. A Genetic Approach to Radio Network Optimization for Mobile Systems. In IEEE, editor, *Proceedings of the IEEE 47th Vehicular Technology Conference (VTC)*, volume 2 of *Technology in Motion*, pages 755–759, May 1997.

[19] E. Cantù-Paz. A Summary of Research on Parallel Genetic Algorithms. Technical report, Illinois Genetic Algorithms Laboratory, 1995.

[20] E. Cantú-Paz and D. E. Goldberg. Predicting speedups of idealized bounding cases of parallel genetic algorithms. In Th. Bäck, editor, *Proceedings of the seventh International Conference on Genetic Algorithms*, pages 113–121. Morgan Kaufmann, 1997.

[21] V. Chvátal. A greedy heuristic for the set-covering problem. *Mathemathical Operations Research*, 3:233–235, 1979.

[22] B. Codenotti and M. Leoncini. *Introduction to Parallel Processing.* Addison-Wesley, 1993.

[23] J. P. Cohoon, S. U. Hedges, W. N. Martin, and D. Richards. Punctuated Equilibria: A Parallel Genetic Algorithm. In *Proceedings of the second International Conference on Genetic Algorithms*, pages 148–154. Lawrence Erlbaum, 1987.

[24] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development - The Fusion Method.* Prentice Hall Object-Oriented Series, Englewood Cliffs, NJ, 1995. ISBN 0-13-338823-9.

[25] A. Colorni, M. Dorigo, and V. Maniezzo. Distributed optimization by ant colonies. In *Proceedings of ECAL91 - European Conference on Artificial Life,*. Elsevier Publishing, 1991.

[26] A. Colorni, M. Dorigo, and V. Maniezzo. Distributed Optimization by Ant Colonies. In MIT Press, editor, *First European Conference on Artificial Life*, pages 134–142. Bradford Books, 1991.

[27] A. Colorni, M. Dorigo, and V. Maniezzo. An investigation of some properties of an ant algorithm. In R. Männer and B. Manderick, editors, *Second European Conference on Parallel Problem Solving from Nature*, pages 509–520. Elsevier Publishing, Brussels, 1992.

[28] A. Colorni, M. Dorigo, and V. Maniezzo. An investigation of some properties of an ant algorithm. In *Proceedings of the Parallel Problem Solving from Nature Conference (PPSN 92)*. Elsevier Publishing, 1992.

[29] J. O. Coplien. *Advanced C++ Programming Style and Idioms.* Addison Wesley, 1992.

[30] M. Cosnard and D. Trystram. *Parallel Algorithms and Architectures.* International Thomson Computer Press, 1995.

[31] D. Costa and A. Hertz. Ants can colour graphs. *Journal of the Operational Research Society*, (48):295–305, 1997.

[32] D. Costa, A. Hertz, and O. Dubuis. Embedding a Sequential Procedure Within an Evolutionary Algorithm for Coloring Problems in Graphs. *Journal of Heuristics*, 1:105–128, 1995.

[33] L. Davis. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, NY, 1991.

[34] M. Dorigo and L. M. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

[35] R.-C. Duh and M. Fürer. Approximation of k-set cover by semi-local optimization. In *Proceedings of the 29th Annual ACM Symposium on Theory Computation*, pages 256–264, 1997.

[36] U. Feige. A Threshold of $\log n$ for Approximating Set Cover. In *Proceedings of the 28th ACM Symposium on Theory of Computing*, 1996.

[37] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21:948–960, September 1972.

[38] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution.* Wiley, New York, 1966.

[39] P. Galinier and J. Hao. Hybrid Evolutionary Algorithm for Graph Coloring. Technical report, EERIE, Nimes, France, April 1999.

[40] M. R. Garey and D. S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-completeness.* Freeman and Co., 1979.

[41] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing.* Scientific and Engineering Computation. MIT Press, 1994.

[42] F. Glover. Heuristics for Integer Programming Using Surrogate Constraints. *Decision Sciences*, (8):156–166, 1977.

[43] F. Glover. Genetic Algorithms and Scatter Search: Unsuspected Potentials. *Statistics and Computing*, (4):131–140, 1994.

[44] F. Glover and M. Laguna. *Tabu Search.* Kluwer Academic Publishers, Norwell, MA, 1997.

[45] D. E. Goldberg. *Genetics algorithms in search, optimization, and machine learning.* Addison-Wesley Publishing Company, Inc., 1989.

[46] D. E. Goldberg, H. Kargupta, J. Horn, and E. Cantu-Paz. Critical Deme Size For Serial And Parallel Genetic Algorithms. Technical Report 95002, Illinois Genetic Algorithms Laboratory (IlliGAL), January 1995.

[47] J. J. Grefenstette. Parallel Adaptive Algorithms for Function Optimization. Technical Report CS-81-19, Vanderbilt University, Nashville, 1981.

[48] F. Guidec, P. Calégari, and P. Kuonen. Object-Oriented Parallel Software for Radio Wave Propagation Simulation in Urban Environment. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *EuroPar'97 Parallel Processing (Third International EuroPar Conference, Passau, Germany, August 1997, Proceedings)*, volume 1300 of *Lecture Notes in Computer Science*, pages 832–839. Springer, September 1997.

[49] F. Guidec, P. Calégari, and P. Kuonen. Parallel Irregular Software for Wave Propagation Simulation. In Hertzberger and Sloot, editors, *High-Performance Computing and Networking (HPCN Europe'97, Vienna), Lecture Notes in Computer Science*, volume 1225 of *Lecture Notes in Computer Science*, pages 84–94. Springer Verlag, April 1997.

[50] F. Guidec, P. Calégari, and P. Kuonen. Parallel irregular software for wave propagation simulation. *Future Generation Computer Systems (FGCS), N.H. Elsevier*, 13(4-5):279–289, March 1998. ISSN 0167-739X.

[51] F. Guidec, P. Calégari, P. Kuonen, and M. Pahud. ParFlow++: a parallel irregular radio wave propagation simulation code. Technical report 98/260, Swiss Federal institute of Technology (EPFL), Computer Science Department, January 1998.

[52] F. Guidec, P. Calégari, P. Kuonen, and M. Pahud. Object-Oriented Parallel Software for Parallel Radio Wave Propagation Simulation in Urban Environment. *Computers and Artifical Intelligence*, (6), 1999. (in print).

[53] F. Guidec, P. Kuonen, and P. Calégari. ParFlow++: a C++ Parallel Application for Wave Propagation Simulation. *SPEEDUP, Proceedings of the 20th SPEEDUP Meeting*, 10(1/2):68–73, December 1996.

[54] F. Guidec, P. Kuonen, and P. Calégari. Radio Wave Propagation Simulation on the Cray T3D. In E.H. D'Hollander, G.R. Joubert, F.J. Peters, and U. Trottenberg, editors, *Parallel Computing: Fundamentals, Applications and New Directions*, volume 12 of *Advances in parallel computing*, pages 155–162. Elsevier Science B.V., 1998. ISBN 0-444-82882-6.

[55] A. Gupta and V. Kumar. Analysing scalability of parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 22(3):379–391, 1994.

[56] S. Gupta and S. Khuller. Greedy strikes back: Improved facility location problems. In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms*, pages 649–657, 1998.

[57] J. L. Gustafson. Reevaluating Amdahl's law. *Communication of the ACM*, 31(5), May 1988.

[58] J. Heitkötter and D. Beasley. The Hitch-Hiker's Guide to Evolutionary Computation (FAQ for comp.ai.genetic), 1996-1999. Available at ftp://ftp.krl.caltech.edu/pub/EC/Welcome.html.

[59] M. Henricson and E. Nyquist. Programming in C++ - Rules and Recommendations. Technical report, Ellemtel Telecommunications Systems Laboratories, Sweden, April 1992.

[60] R. W. Hockney and C. R. Jesshope. *Parallel Computers*. Adam Hilger Ltd, Bristol, 1981. ISBN 0-85274-422-6.

[61] W. J. R. Hoeffer. The Transmission-Line Matrix Method: Theory and Applications. *IEEE Transactions on Microwave Theory and Techniques*, 33(100):882–893, October 1985.

[62] F. Hoffmeister. Scalable Parallelism by Evolutionary Algorithms. In M. Grauer and D. B. Pressmar (Eds.), editors, *Parallel Computing and Mathematical Optimization*, number 367 in Lecture Notes in Economics and Mathematical Systems, pages 177–198. Springer-Verlag, 1991.

[63] J. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, 1975.

[64] Ian Joyner. C++?? A critique of C++ and Programming and Language Trends of the 1990s (3rd Edition), 1996. Page created in February 1996, http://www.progsoc.uts.edu.au/ geldridg/cpp/cppcv3.html.

[65] R. M. Karp. Reductibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103, New York, 1972. Plenum Press.

[66] M. J. Kearns. *The Computational Complexity Problems*. MIT Press, 1990.

[67] D. Kobler. *Modèles biologiques en optimisation combinatoire et modèles mathématiques en génétique*. PhD thesis, number 2005, Swiss Federal Institute of Technology (EPFL), Switzerland, July 1999.

[68] D. Kobler, P. Calégari, G. Coray, A. Hertz, and P. Kuonen. A Taxonomy of Evolutionary Algorithms in Combinatorial Optimization. In EPFL, editor, *Program and abstracts of the* 16$^{\text{th}}$ *International Symposium on Mathematical Programming, Lausanne, Switzerland*, page 149, August 1997.

[69] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Complex Adaptive Systems. The MIT Press, Cambridge, 1992.

[70] P. Kunst and D. Snyers. Emergent Colonization and Graph Partitioning. In *Proceedings of the Third International Conference of Adaptive Behavior*, pages 494–500. MIT Press, 1994.

[71] P. Kuonen. *La Programmation Parallèle Asynchrone et son Application aux Problèmes Combinatoires.* PhD thesis, number 1205, Swiss Federal Institute of Technology (EPFL), Switzerland, 1993.

[72] P. Kuonen, F. Guidec, and P. Calégari. Multilevel Parallelism applied to the optimization of mobile networks. In A. Tentner, editor, *Proceedings of the High-Performance Computing (HPC'98)*, pages 277–282. Society for Computer Simulation International, April 1998. ISBN 1-56555-145-1.

[73] F. T. Leighton. *Journal of Research of the National Bureau of Standards*, 84:489–505, 1979.

[74] D. Levine. *A Parallel Genetic Algorithm for the Set Partitioning Problem.* PhD thesis, Illinois Institute of Technology, Department of Computer Science, 1994.

[75] P. O. Luthi, B. Chopard, and J.-F. Wagen. Wave Propagation in Urban Micro-cells: a Massively Parallel Approach using the TLM Method. In *Proceeding of PARA'95, Workshop on Applied Parallel Scientific Computing, Copenhagen*, August 1995. Also in COST 231 TD(95) 33.

[76] B. Meyer. *Object-Oriented Software Construction (2nd Edition).* Prentice Hall, 1997 (first edition 1988). ISBN 0-13-629155-4.

[77] S. Micali and V. Vazirani. An $O(|e|\sqrt{|V|})$ algorithm for maximum matching in general graphs. In *Proceedings of the 21st IEEE Annual Symposium on the Foundations of Computer Science*, pages 17–27, 1980.

[78] P. Moscato. On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Towards Memetic Algorithms. Technical report, California Institute of Technology, Pasadena, CA 91125, USA, 1989.

[79] P. Moscato. Memetic algorithms' home page, 1999. Page created in February 1996, http://www.ing.unlp.edu.ar/cetad/mos/memetic_home.html.

[80] S. Näher and C. Uhrig. *The LEDA User Manual Version R 3.3.c*, 1996.

[81] M. Pahud. *Une Méthode de Prédiction de Performance pour les Programmes Parallèles Irréguliers.* PhD thesis, number 1911, Swiss Federal Institute of Technology (EPFL), Switzerland, December 1998.

[82] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization, Algorithms and complexity.* Prentice-Hall, 1982.

[83] V. Paschos. A survey on approximately optimal solution to some covering and packing problems. *ACM Computing Surveys*, 29(2):172–209, June 1997.

[84] C. C. Pettey and M. R. Leuze. A Theoretical Investigation of a Parallel Genetic Algorithm. In *Proceeding of the Third International Conference on Genetic Algorithms*, pages 398–405, 1989.

[85] M. J. Quinn. *Designing Efficient Algorithms for Parallel Computers.* Computer Science Series. McGraw-Hill, 1988. ISBN 0-07-100249-9.

[86] I. Rechenberg. Cybernetic solution path of an experimental problem. Technical Report Library translation 1122, Royal Air Force Establishment, Farnborough, Hants., UK, 1965.

[87] Y. Rochat and E. D. Taillard. Probabilistic diversification and intensification in local search for vehicule routing. *Journal of Heuristics*, 1:147–167, 1995.

[88] H.-P. Schwefel. Numerische Optierung von Computer-Modellen mittels der Evolutionsstrategie. Volume 26 of *Interdisciplinary Systems research.* Birkhäuser, Basel, 1977.

[89] P. Slavík. Improved Performance of the Greedy Algorithm for Partial Cover. *Information Processing Letters*, 64(5):251–254, December 1997.

[90] P. Slavík. A Tight Analysis of the Greedy Algorithm for Set Cover. *Journal of Algorithms*, 25(2):237–254, November 1997.

[91] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference.* Scientific and Engineering Computation Series. MIT Press, 1996. ISBN0-262-69184-1.

[92] W. M. Spears and K. DeJong. An analysis of multi-point crossover. In G. J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 301–315. Morgan Kaufmann, 1991.

[93] P. Spiessens and B. Manderick. A Massively Parallel Genetic Algorithm. In L. B. Booker (eds.) R. K. Belew, editor, *Proceedings of the 4th International Conference on Genetic Algorithms*, pages 279–286. Morgan Kaufmann, 1991.

[94] T. Starkweather, D. Whitley, and K. Mathias. Optimization using distributed genetic algorithms. In R. Männer and H.-P. Schwefel (Eds.), editors, *Proceedings of the First International Conference on Parallel Problem Solving from Nature*, pages 176–186. Springer, 1991.

[95] B. Stroustrup. *The C++ Programming Language (3rd Edition).* Addison Wesley, 1997 (first edition 1986). ISBN 0-201-88954-4.

[96] G. Syswerda. Uniform crossover in genetic algorithms. In J. D. Schaffer, editor, *Proceeding of the Third International Conference on Genetic Algorithms*, pages 2–9. Morgan Kaufmann, 1989.

[97] E.-G. Talbi. A Taxonomy of Hybrid Metaheuristics. Technical Report AS-183, Laboratoire d'Informatique Fondamentale de Lille, LIFL, USTL, 59655 Villeneuve d'Ascq, France, May 1998. To be published in Journal of Combinatorial Optimization, Kluwer Academic Publishers, Boston.

[98] R. Tanese. Parallel Genetic Algorithms for a Hypercube. In *Proceeding of the Second International Conference on Genetic Algorithms*, pages 177–183, 1987.

[99] R. Tanese. Distributed genetic algorithms. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 434–439, George Mason University, June 1989. Morgan Kaufmann.

[100] M. Trick. Graph Coloring Instances, 1999. Page created in October 1994, http://mat.gsia.cmu.edu/COLOR/instances.html and color.html.

[101] L. G. Valiant. Bulk-Synchronous Parallel Computers. In M. Reeve and S. E. Zenith, editors, *Parallel Processing and Artificial Intelligence*, Chichester, UK, 1989. Wiley.

[102] D. Whitley. A genetic algorithm tutorial. Technical Report CS-93-103, Colorado State University, 1993.

This report can be referenced as:
P. Calégari. *Parallelization of population-based evolutionary algorithms for combinatorial optimization problems.* PhD thesis, number 2046, Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland, September 1999.

Index

adaptative memory, 16
agent, 81
age of Universe, 90
algorithm
  definition, 20, 21
Amdahl's law, 24
ant
  trail, 14, 15
  visibility, 14, 15
ant colony system, 13, 15
ant system, 13
  TEA description, 43
  application, 98, 111
  parallel, 62
APPEAL, 69, 80
  implementation, 79
  object model, 71–78
application
  graph coloring, 107
  transceiver siting, 85
archipelago, 38
AS, *see* ant system
asynchronous model, 27, 82

bottleneck, 28
BSP, 21, 50
BTS, 85

cell, 86
chromatic number, 107
classification
  parallel computer architecture, 21
  parallel EAs, 28
coarse-grain parallelism, 25
combinatorial optimization, 5
  classes of problems, 7
  classical methods, 8
  constructive approach, 8
  sequential approach, 8
combinatorial problem, 5
communication load, 25
constructive approach, 8
contention, 25, 55, 56
COW, 22
crossover
  one-point, 11, 32

darwinism algorithm, 104
decision problem, 6
deme, 18
dependency, 25
diversification, 10

EA, *see* evolutionary algorithm
efficiency, 23, 24
emergent colonization algorithm, 10
encoding, 34
entity, 51, 53, 54
$\epsilon-$net, 104
ES, *see* evolution strategy
evolution, 9, 34, 57, 123
evolutionary algorithm, 9
  classification, 31
  glossary, 123
  ingredients, 32–36
evolutionary programming, 13
evolution strategy, 12
exploitation, 10
exploration, 10

farmer/worker model, 28
fine-grain parallelism, 25

fitness, 9, 123
frequency allocation, 108
Fusion method, 71

GA, *see* genetic algorithm
generation, 123
generational replacement evol., 34, 57
genetic algorithm, 11
  TEA description, 41
  application, 94, 109
  parallel, 60
genetic ant algorithm, 46
  TEA description, 47
  motivation, 46
genetic programming, 12
genotype, 9, 123
global parallelization, 53
granularity, 25, 50
graph coloring, 107
  instances, 108
greedy-like algorithm, 8, 90
  application, 90, 109

heuristic, 7
history
  of the element, 56
  of the individual, 34
  of the population, 33, 43
  of the set, 40
hitting set, 89
hybrid algorithm, 18, 35
  application, 100, 114
  genetic ant algorithm, 46, 47
  parallel, 66
  parallelization, 58

IAS, 62, 63
  application, 98, 111
  parallel, 62
IGA, 38
  application, 94, 109
  parallel, 60
IGAA, 46
  application, 100, 114
  parallel, 66
improving algorithm, 35, 56
individual, 9, 123
ingredient, *see* evolutionary algorithm
integer part notation, 55
intensification, 10
island-based genetic ant algorithm, 47
island model, 18, 38
  TEA description, 41
  asynchronous, 82
  description level, 40
  genetic algorithm, 29, 38
  phenomenon, 45
  structured space, 39

$L_\emptyset$, 51, 54
LEOPARD project, vii
level
  in the TEA, 37
  of parallelization, 50–55
libraries
  APPEAL, 71–80
  existing, 69
  LEDA, 79
  MPI, PVM, 79
load balancing, 26
local search, 8, 19

master/slave model, 28
memetic algorithm, 18
meta-heuristic, 18
migration, 45
  rate,time-scale, 45
MIMD, 22, 29, 50

neighborhood, 5, 33
network configuration, 91
noise, 35, 57
NOW, 22
NP-complete, 7
NP-hard, 7

object-oriented model, 71–78
operator +, 54

operator //, 53, 54
optimal solution, 5
optimization problem, 5

ParaGene, 106
parallelization, 21
 criteria, 58
 level, 50–55
parallelization rules
 influence of the TEA ingredients, 55
 influence of the TEA levels, 52
parallel algorithm
 definition, 21
 models, 26
parallel computer architecture, 21
parallel computing, 20
 constraints, 22
parallel EA
 efficient, 60
 experiments, 94–106, 109–117
 parallelization, 49
 why?, 31
ParFlow++, 86
partitioning model, 27
PBIL, 16
 TEA description, 43
PGA, 29
phenotype, 9, 123
pheromone, 14
pipeline model, 26
population, 9, 32, 123
population-based incremental learning, 16
PRAM, 21, 50

radio network planning, 85
radio wave propagation simulation, 85
randomization, 10, 35
recognition problem, 6

scalability, 25
scatter search, 16, 32
 TEA description, 42
selection pressure, 44, 45
set cover problem, 89
set system, 89
SIMD, 22, 29, 50, 52
speed-up, 23, 24
 experiments, 94–101, 109–115
 pipeline, 27
steady state evolution, 34, 57
STORMS project, vii, 85, 106
structured space, 33, 38, 55
 phenomenon, 44
sub-population, 28, 53
super-linear, 92

tabu search, 9
task mapping, 26, 29
taxonomy, 31
 hybrid meta-heuristic, 18, 28
TEA
 basic, 36
 complete, 39
 examples of use, 41
 motivation, 31
 parallelization analysis, 55
topology, 22, 33, 44, 56
transceiver siting, 85
 instances, 87
traveling salesman problem, 6
TSP, 6

weighted set system, 89, 91

Curriculum Vitae

1. In English

Born in 1969, Patrice Calégari spends 3 years at the "École normale supérieure de Lyon" where he obtains a M.Sc. of theoretical computer science and an engineer diploma of computer science and modeling in 1994 (University of Lyon I, France). During that period, he pursued training periods on parallel algorithms of reaction-diffusion at the University of Geneva (Switzerland), on a pattern recognition software at the University of Helsinki (Finland), and on hand-writing recognition software in Paris (France) for the SLIGOS company.

In 1995, after achieving his obligatory military service, he becomes assistant at the Swiss Federal Institute of Technology in the computer science theory laboratory (LITH) where he participates actively to the European project STORMS (object-oriented design of the global software, realization of libraries and parallel software, etc.). In 1996, he starts his PhD. thesis in the same laboratory where he also teaches (parallel computing, C++ *language). The list of his publications includes [11, 12, 13, 14, 15, 16, 17, 18, 48, 49, 50, 51, 52, 53, 54, 72].*

His external activities are: B&W and color artistic photography (landscape, portrait, etc.), French and International gastronomy, discovery of nature and of different cultures through numerous trips, trek, ski, sub-diving and sailing.

2. En français

Né en 1969, Patrice Calégari passe 3 ans à l'École normale supérieure de Lyon où il obtient un DEA d'informatique théorique et un Magistère d'informatique et modélisation en 1994 (diplômes de l'Université Claude Bernard Lyon I, France). Pendant cette période, il effectue des stages sur des algorithmes parallèles de réaction-diffusion à l'Université de Genève (Suisse), sur un logiciel de reconnaissance de formes à l'Université d'Helsinki (Finlande), et sur la reconnaissance de l'écriture manuscrite à Paris (France) dans la société de service informatique SLIGOS.

En 1995, après avoir effectué son service militaire obligatoire, il entre comme assistant au Laboratoire d'Informatique Théorique de l'École Polytechnique Fédérale de Lausanne où il participe activement au projet européen STORMS (conception orientée objet de l'application globale, réalisation de librairies et d'applications parallèles, etc.). En 1996, il débute son travail de thèse dans ce même laboratoire où il fait également de l'enseignement (parallélisme, langage C++*). La liste de ses publications comprend [11, 12, 13, 14, 15, 16, 17, 18, 48, 49, 50, 51, 52, 53, 54, 72].*

Ses activités externes sont : la photographie artistique N&B et couleur (paysage, portrait, etc.), la gastronomie française et internationale, la découverte de la nature et de différentes cultures à travers de nombreux voyages, la randonnée, le ski, la plongée sous-marine et la voile.