Laboratoire des Sciences de l'Image,
de l'Informatique et de la Télédétection

UMR 7005 UDS/CNRS

École Doctorale Mathématiques, Sciences de l'Information et de l'Ingénieur

Thèse

présentée pour l'obtention du grade de

Docteur de l'Université de Strasbourg
Discipline : Informatique

par

Ogier Maitre

GPGPU for Evolutionary Algorithms

Soutenue publiquement le 12 décembre 2011 (Version 1er mars 2012)

Membres du jury

Directeur de thèse :	Pierre Collet, Professeur, Université de Strasbourg
Co-Encadrant de thèse :	Nicolas Lachiche, Maître de conférence, Université de Strasbourg
Rapporteur :	Enrique Alba, Professeur, Université de Málaga, Espagne
Rapporteur :	Wolfgang Banzhaf, Professeur, Univesité de Terre-Neuve, Canada
Examinateur :	Paul Bourgine, Directeur de recherche, École Polytechnique Paris, France

Contents

Remerciements 7

Introduction **8**

Emergence of multi-core architectures for mainstream users 9
Exploitation of multicore architectures 9
General-Purpose Graphics Processing Units 10
Evolutionary algorithms and GPGPUs 11
Outline 11

I State of the art 13

1 Evolutionary algorithms **15**

1.1 Introduction 15
1.2 History and principles 15
1.3 Internal elements 16
1.3.1 Individual representation 16
1.3.2 Evaluation function 18
1.3.3 Population initialization 19
1.3.4 Evolutionary loop 19
1.3.5 Variation operators 20
1.3.6 Selection and replacement operators 22
1.3.7 Stopping criteria 22
1.3.8 Parameters 23

2 EASEA software, before 2008 **25**

2.1 Specification 25
2.2 Code generation 26
2.3 Specification format 27

3 Classical parallization of EAs **31**

3.1 Classical architectures 32
3.2 Classification 33
3.3 Global parallel evolutionary algorithm 33
3.3.1 Standard approach 33
3.3.2 Non-standard approaches 36
3.4 Distributed evolutionary algorithms 38
3.4.1 Standard approach 38
3.4.2 Hardware oriented approach 40

3.5 Cellular evolutionary algorithms . . . 40

4 GPGPU Hardware 43
4.1 History . . . 43
4.2 System overview . . . 44
4.3 Hardware part . . . 45
4.3.1 Computing elements organization . . . 45
4.3.2 Memory hierarchy . . . 48
4.3.3 Register . . . 48
4.3.4 Shared memory . . . 49
4.3.5 Global memory . . . 49
4.3.6 Considerations about memory spaces . . . 49
4.4 GPGPU Software . . . 50
4.4.1 Compilation chain . . . 51
4.4.2 Thread organization . . . 51
4.4.3 Threads scheduling . . . 52

II Contributions 55

5 Common principles 57
5.1 Analysis of EA parallelization . . . 57
5.2 Load Balancing . . . 60
5.2.1 Inter GPGPU load balancing . . . 60
5.2.2 Intra GPGPU load balancing . . . 60

6 Parallelization of a GA/ES on a GPGPU card 63
6.1 Rationale . . . 63
6.2 Related works . . . 63
6.3 Proposed Master-Slave implementation . . . 64
6.3.1 Algorithm . . . 64
6.3.2 Evaluation . . . 66
6.4 Experiments . . . 66
6.5 Conclusion . . . 70

7 Genetic Programming on GPGPU cards 71
7.1 Introduction . . . 71
7.1.1 Option 1: SIMD parallelization of a single individual evaluation over the learning set . . . 72
7.1.2 Option2: MIMD parallelization over the whole population . . . 73
7.2 Related work . . . 75
7.3 Implementation . . . 76
7.3.1 Algorithm . . . 76
7.3.2 Evaluation . . . 77
7.4 Experiments . . . 77
7.4.1 Experimental process . . . 77
7.4.2 Evaluation function . . . 78
7.4.3 Experiments on the complete algorithm . . . 82
7.4.4 Example on a real world problem . . . 84

8 A complete EA on GPGPU 89
8.1 Tournament study . . . 90
8.1.1 Analysis of the population distribution . . . 91
8.1.2 Reproduction rate . . . 92
8.1.3 Selection intensity . . . 92
8.1.4 Loss of diversity . . . 93
8.1.5 Experimental measurements on tournament for reduction . . . 95
8.2 DISPAR-tournament . . . 95
8.2.1 Principles . . . 95
8.2.2 Experiments with DISPAR . . . 96
8.3 DISPAR-tournament on GPGPU . . . 97
8.4 Complete generational algorithm on a GPGPU card . . . 99
8.5 Experiments . . . 101
8.5.1 Speedup . . . 101
8.5.2 Timing distribution analysis . . . 103
8.5.3 Qualitative behaviour on the Weierstrass function . . . 104

9 Modern version of EASEA 107
9.1 Diffusion of research results using EASEA . . . 108
9.2 Internal structure of EASEA . . . 109
9.2.1 Code generator and templates . . . 109
9.2.2 LibEASEA . . . 111
9.3 Island Model . . . 112

10 EA applications 115
10.1 Data-driven multi-objective analysis of manganese leaching . . . 115
10.1.1 Presentation of the problem . . . 115
10.1.2 Results . . . 117
10.2 Propositionalisation using EA . . . 119
10.2.1 Introduction . . . 119
10.2.2 Problem presentation . . . 119
10.2.3 Evolutionary Algorithm description . . . 120
10.2.4 Experiments . . . 122
10.2.5 Conclusion . . . 124
10.3 Zeolite structure optimisation . . . 125
10.3.1 Problem description . . . 125
10.3.2 EAs for zeolite structure optimization . . . 126
10.3.3 Speedup on zeolite problem . . . 126
10.3.4 The zeolite search continues . . . 127

11 Discussion and perspectives 129
11.1 Porting EA on many-core architectures . . . 129
11.1.1 Genetic algorithm and Evolution strategies . . . 129
11.1.2 Population size adaptation . . . 131
11.1.3 Distributed EAs on GPGPUs . . . 132
11.1.4 Asynchronous evolutionary algorithms on GPGPUs . . . 133
11.2 Metrics to compare parallel EA . . . 134
11.2.1 Metric for GA/ES on GPGPU and CPU algorithms . . . 134
11.2.2 Metric for Genetic Programming . . . 135

Conclusion **139**

A Personal Bibliography **155**
A.1 Book chapters 155
A.2 Journal articles 155
A.3 Conference papers 155

Remerciements

Je tiens à remercier mes parents, pour leur aide et leur soutien pendant toute la durée de mes longues études. Je tiens aussi à remercier Cornelia Bujeniţã, pour sa patience, son aide et son soutien sans faille pendant la rédaction de cette thèse et des publications associées, ainsi que pendant tous mes travaux de recherches.

De même, je remercie mon directeur et mon encadrant, pour leurs conseils et aide prodigués tout au long de ma thèse. Cette formation à la recherche que fut ma thèse a bien sûr été possible principalement grâce à eux.

Je souhaite aussi remercier, Arijit Biswas pour sa collaboration, notamment sur la partie programmation génétique et pour nos travaux communs. Au chapitre des collaborateurs, je tiens tout particulièrement à remercier Laurent Baumes, grâce à qui j'ai pu travailler sur un problème réel, de grande complexité. Je n'oublie pas, Deepak Sharma dont les conseils dans nos travaux communs et explications avancées sur l'optimisation multi-objectifs m'ont donné un recul nouveau sur notre discipline. De même, je remercie Stéphane Querry, pour ses problèmes d'optimisation liés à l'avionique et pour le travail qu'il a réalisé lors de nos publications communes. Ces problèmes, en plus d'être passionnants et ambitieux, m'ont confirmé l'importance que l'EA peut avoir dans l'optimisation de problèmes réels.

Je souhaite remercier Frédéric Krüger, pour son aide, ses idées dans le développement d'EASEA et lors de nos travaux communs.

Merci à mon équipe de recherche BFO, pour son accueil et sa chaleureuse ambiance, ainsi qu'à l'ICPS et à Vincent Loechner pour m'avoir fait découvrir la parallélisation et les processeurs multicoeurs, lors de mon stage de master. Je tiens tout particulièrement à remercier Stéphane Marchesin, de m'avoir initié à l'utilisation de GPGPU, à travers son projet Nouveau.

Je remercie Enrique Alba, pour avoir accepté la charge de relecteur et donc de prendre de son temps pour lire ce document et écrire un rapport, ainsi que pour ses commentaires et questions lors de notre dernière rencontre, ainsi que Wolfgang Banzhaf, également pour son temps et son énergie pour ce même travail, mais aussi pour son point de vue et son ouverture d'esprit lors de sa visite dans notre équipe.

Je tiens à remercier Paul Bourgine d'avoir accepté de participer au jury de cette thèse. Je suis sûr qu'il apportera plein d'idées de futur dénveloppements.

Je tiens à remercier NVIDIA, pour leurs dons qui m'ont permis d'utiliser du matériel professionnel récent durant les expériences de cette thèse et des publications associées. Finalement, je tiens à remercier le CROUS pour le financement des mes premières années d'études, le système universitaire et scolaire français, pour m'avoir donné la chance de les accomplir.

Introduction

Contents

Emergence of multi-core architectures for mainstream users 9
Exploitation of multicore architectures 9
General-Purpose Graphics Processing Units 10
Evolutionary algorithms and GPGPUs 11
Outline 11

Emergence of multi-core architectures for mainstream users

The context is changing in computer science: until recently, CPU manufacturers still managed to increase clock speed, allowing them to improve the power of their chips while staying within a single core architecture. Unfortunately, it seems that a limit has recently been reached for air-cooled personal computers with Intel's NetBurst architecture used in Pentium IV, that topped at 3.8GHz in 2005.

These chips were plagued with problems related to their very high frequency (heat dissipation problems due to their high 110W Thermal Design Power) and their architecture also pushed pipelining to its limit, with up to 31 stages on Prescott micro-architecture, making prediction errors particularly costly, compared to standard pipelines.

The single-core architecture paradigm was pushed to an end, leading to new trends in processor architecture design. A new direction was taken, with the introduction of multi-core processors into desktop machines (Pentium D, Core Duo and Core 2 Duo, Athlon 64 X2), ending the race in clock frequency. Still due to heat dissipation problems, the first processors using this dual-core architecture came out with a frequency reduced to 1.8GHz. Officially, frequency can no longer be used to compare CPU performance, as their micro-architecture must now be taken into account. This trend was common to all CPU manufacturers.

In multi-core processors, the user now runs multiple threads in parallel, as was the case in old multi-processor systems used in professional environments (multi-user servers, web servers, High Performance Computing (HPC) clusters, a.s.o.). The number of cores then increased to 4. Using Simultaneous Multi Threading (SMT), such processors could run up to 8 concurrent threads, with the drawback that in order to attain its peak computing power, such processors *must* perform several tasks simultaneously.

Exploitation of multicore architectures

Exploitation of these resources is possible thanks to the task scheduler, which should assign a task (a program) to each of the different cores. This method is similar to what was used in multi-user

systems, where a large number of tasks owned by all logged users was scheduled on the different processors of the computer. On modern systems, the scheduler distributes the tasks of one user on all the cores of the CPU (web browser, music listening program, system daemons).

This technique presents several advantages, including a strong system responsiveness to user interaction and use of already existing software and hardware concepts. Indeed, standard task scheduling of multi-processor can be adapted to this case.

However, some questions remain open: load balancing in particular is a difficult problem. Due to the single-user aspect, the number of heavy tasks is not necessarily high enough. Users typically executes one main task, which generally is the CPU-time consuming one.

The larger task uses all the available processing power only if it is programmed in order to be run on a multi-processor system, *i.e.* if it is a parallel application. But in the general case, it is not, as most applications are still developed in a sequential fashion. In order to overcome this problem, some mechanisms have been implemented in order to increase the power of one of the cores at the expense of the others.

Multi-core architecture also increases the pressure on memory, as all system cores share access to the same memory device. If an application is mainly limited by the memory speed of the machine, running it on a multi-core architecture does not bring any improvement, even if it can be decomposed in sub-tasks suitable for multi-core execution. The same goes for cache memory, that is shared by all the cores.

Multi-core processors are also useful for high performance computing (HPC). They can be used in multiprocessor systems in a transparent way, either by reducing the price of the system or by increasing the number of cores available on each computing node. Using standard HPC parallelization techniques brings some advantages as, for example, load balancing is simplified. Indeed, parallelization is often a regular division of a given task into a set of equal sub-tasks. A single application running on different cores of a computing node allows to share data between the cores. Such sharing can include smart cache sharing. Reuse of data increases, as does memory efficiency of the application. The power used by an application can approach the theoretical peak of these processors, especially for applications where the bottleneck is computing power.

But unfortunately, such techniques do not apply to some applications, that do not scale to a large number of tasks. Some cannot be parallelized at all, but they already were problematic in the case of standard parallelization over multi-processor parallel architectures.

Multi-core architectures bring a real advantage in computation clusters, where parallelism between the different computation nodes, between the multiple processors and between the multiple cores is exploited. If the last two levels of parallelism are just slightly different, clusters with single processor computers are nowadays pretty rare and every application using such cluster architectures must use different parallelization paradigms, one for the shared memory machines and another for distributed memory machines (for instance: OpenMP and MPI).

General-Purpose Graphics Processing Units

Video games require a lot of computing power, especially in order to perform 3D rendering. Computer Aided Design (CAD) and animated films techniques are used here to render real-time scenes of the worlds in which the players evolve. This market lead to the development of hyper-specialized devices installed in expansion boards in order to help CPUs performing these 3D rendering computations.

Early developments of theses graphics cards start in the 90's, but it is necessary to wait until the 2000s in order to use them to speedup non-3D rendering computations, with General-Purpose Graphics Processing Units (GPGPUs). The processors of these devices embed a large number of

cores (several hundreds on modern processors), implement a Simultaneous Multi-Threading (SMT) mechanism (several dozens contexts) and host a fast and large memory.

They have an interesting power/cost ratio, as they are developed by the video game industry which has an important number of customers that can absorb developing costs more easily than the much smaller HPC market. This makes it a particularly common equipment on desktop computers and modern video game consoles. With the development of scientific computing on these architectures, parallel machines are now also equipped with this type of processors. For example, the number 1 fastest supercomputer in the world in September 2011, Tianhe-1A, is a heterogeneous architecture using GPGPUs.

But using GPGPUs remains difficult for HPC due to their numerous hardware quirks. These architectures are very special and are still developed to perform 3D renderings. They remain difficult to exploit efficiently, even though some manufacturers are making efforts to generalise their architecture. Similarly, only a small number of programming libraries are available for these architectures and each development requires to recode almost everything, in a GPGPU-adapted way. But as for the architecture, things are improving and more and more sophisticated tools are emerging in order to help develop generic programs on GPGPUs.

GPGPUs are based on an SIMD architecture (with some flexibility) which makes them difficult to use, as many applications cannot use this paradigm. In addition, in order to maximise computing speed, it is necessary to use as many cores as possible, resulting in scalability problems, because there can be hundreds of them. GPGPU chips go beyond the concept of multi-core architecture. They are many-core processors.

The scope of these architectures is thus limited and only some applications can benefit from them, but there are nevertheless many attempts to port existing applications on these GPGPUs, due to their very attractive power / cost ratio.

In fact, the principles used in GPGPU architectures was commonly found in older parallel processors. SIMD architectures such as shared memory computers, with a lot of cores, have existed, but were abandoned when easier and/or more efficient ways of developing more computing power were invented (shared memory clusters).

Evolutionary algorithms and GPGPUs

Evolutionary algorithms (EAs) are inspired from natural evolution. They allow to find solutions that are not necessarily optimal —but often satisfactory enough— to many complex inverse problems but they use a lot of computing power.

What makes nature inspired algorithms in general and EAs in particular interesting in the context of GPGPU computing is that they are *intrinsically parallel*. Indeed, they realize an exploration / exploitation of the search space by evolving a set of possible solutions. Only a small number of their steps need a global communication. As is the case in many intensive computing domains, many works are being carried on the parallelisation of these algorithms over various architectures, including GPGPU processors, as these algorithms, which are intrinsically parallel, can use the very large number of processing cores that are found in GPGPU chips.

Finally, the SIMD character of GPGPU parallelism is not that much of a limitation to EAs and their synchronous aspect makes them very good candidates for an efficient use of GPGPUs.

Outline

Part I of this document will start with a state of the art on the topic. Then, starting with a description of the evolutionary algorithm that will be used throughout this document, we will

explore the implementation of different evolutionary algorithms on classical parallel architectures. Then we will describe the structure of GPGPU cards, before presenting EASEA, an evolutionary algorithm specification framework which will serve as a support for this thesis.

Part II will contain the heart of the subject by analyzing how an evolutionary algorithm can be parallelized, and also by going through the foundations of a task distribution algorithm for GPGPU. We will detail various implementations of evolutionary algorithms on GPGPU, before presenting the current state of the EASEA platform after this thesis.

Part III will present some applications of evolutionary algorithms developed during this PhD thesis, using or not using GPGPUs, and will include a discussion on massively parallel evolutionary computation and its potential developments.

Finally, a conclusion will end this document.

Part I

State of the art

Chapter 1

Evolutionary algorithms

Contents

1.1 Introduction **15**
1.2 History and principles **15**
1.3 Internal elements **16**
1.3.1 Individual representation 16
1.3.2 Evaluation function 18
1.3.3 Population initialization 19
1.3.4 Evolutionary loop 19
1.3.5 Variation operators 20
1.3.6 Selection and replacement operators 22
1.3.7 Stopping criteria 22
1.3.8 Parameters 23

1.1 Introduction

1.2 History and principles

Evolutionary algorithms (EAs) are usually used to search for the best possible solutions to complex inverse problems. They incrementally improve a set of potential solutions that explore / exploit in parallel a search space comprising all the feasible solutions. Basically, they use the evolutionary principle discovered by Charles Darwin, who described the adaptation of species to an environment through reproduction, variations and survival of the fittest individuals.

The idea to implement evolutionary algorithms *in silico* starts independently in a few places, during the late 50's. The goals of the researchers behind these first works are different and include the simulation of natural evolution [1] and the automatic programming of computers [2]. Foundations being laid, it is still necessary to wait for computers to have enough processing power so that evolutionary computation can become competitive with other stochastic optimization techniques.

Surviving trends in these early paradigms are:

- Evolutionary Strategies (ES) [3, 4],
- Genetic Algorithms (GA)[5, 6],

- Evolutionary Programming (EP)[7, 8],
- Genetic Programming (GP)[9, 10]

These different implementations of the evolutionary computation principle survive in their respective fields of application, which can be grossly summed up as optimization of combinatorial problems for GAs, optimization of continuous problems for ES and optimization of problems whose structure is not known for GP.

Finally, [11, 12] suggest a unified view of evolutionary algorithms in a common paradigm. From this point of view, the different variants are instances of a common model for evolutionary computation, each instance having its own parameters, genome definition and operators to manipulate their genome.

1.3 Internal elements

Evolutionary algorithms are broken down into several interrelated parts.

1.3.1 Individual representation

Because of its analogy with the living, EAs evolve *individuals* that compose a population. Each individual is a potential solution to the problem being optimized. It is encoded by a set of genes that describes it. These genes are the genotypic representation of an individual, which are turned into a phenotypic representation (the gene carrier) for evaluation.

The individual representation is a key point in classifying historical evolutionary algorithms. Computers represent all data as binary digits and researchers developing GAs also use this representation in order to encode the genomes of their individuals. If the optimization of discrete problems is well suited to this representation, it is more difficult to optimize continuous problems using this technique. While in a combinatorial problem, inverting a bit in the genome may cause a small displacement in terms of Hamming distance, if the same genome implements an integer value, it is more difficult to quantify this displacement. If a boolean mutation (bit flip) is applied to the least significant bit of an array of bits representing an integer, the displacement of the individual in the search space will be smaller than if the bit flip is applied to the most significant bit.

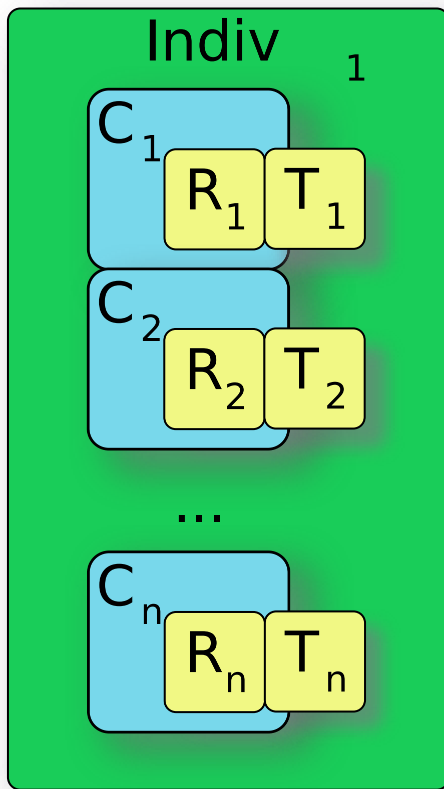
ES users implement a genome representation in the form of actual real-valued variables and develop variation operators which are adapted to this representation, abstracting the fact that the underlying implementation is also based on bits.

Representation introduces biases in the research, which greatly influence how easy or difficult it will be to find the solution. It is easy to understand that even though all computer programs end up being coded with processor instructions, a particular software may be easier to develop in C or JAVA rather than directly in assembly language. Of course, these languages also introduce limitations on the possible programs that can be implemented. Variation operators that manipulate the individuals are also different depending on the kind of representation that is used.

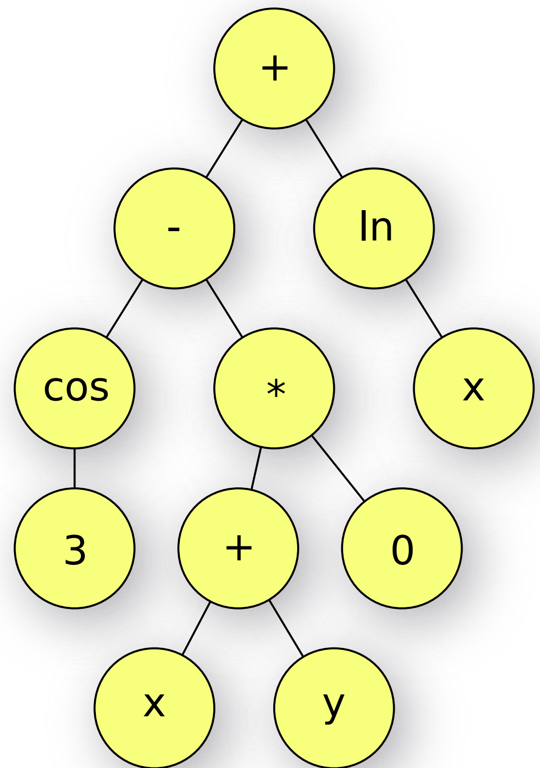
Even with an identical representation, the format of the individual will also distort the search space. In a binary representation where single or multi-point crossovers are used, two interdependent genes separated by many other genes will more likely be disrupted than if they were close one to another.

Finally, restricting the search space to “reasonable” solutions will help the algorithm find a solution within these bounds but may prevent it from finding a better unexpected solution.

Figure 1.1(a) presents the genome of an individual implemented to solve timetabling problems. The genome is a set of n course topics, each described by a room R_i and time-slot T_i. This representation is used in Burke et al. [13].



(a) A genome representation for exams timetabling.



(b) A genetic programming individual representation.

Figure 1.1: Two individual representations for two evolutionary algorithm problems.

Figure 1.1(b) presents a tree-shaped genome, which is commonly used in genetic programming to represent a mathematical expression that is equivalent to formula 1.1:

$$\Big(\cos(3) - \big((x+y) \times 0\big)\Big) + \ln(x) \tag{1.1}$$

where x and y are input variables from the learning set.

1.3.2 Evaluation function

The evaluation function, also called the fitness function, is a key part of an evolutionary algorithm. It allows to rate an individual and is specific to the problem, along with the genome representation. This function should allow to compare between different individuals, possibly with a non total order (in this special case, two individuals are compared in a tournament that will tell which of the two is the best). To continue the analogy with biology, the function evaluates how adapted an individual is to its environment.

The genotypic representation of an individual typically needs to be turned into a phenotypic representation in order to be evaluated (it is difficult to judge on the sex-appeal of a member of the opposite gender by looking at his/her genome only).

Evaluation functions represent the search space. They are often highly multi-modal (*i.e.* they have many local optima) when they are used in evolutionary algorithms. The evaluation function describes a fitness landscape that the algorithm tries to optimize, by sampling it using a population. Indeed, the population contains individuals spread over the fitness landscape and variation operations applied to these individuals move them around, trying to send them to the most interesting parts of the search space.

$$F_{ackley}(z) = -20\exp\left(-0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D} z_i^2}\right) - \exp\left(\frac{1}{D}\sum_{i=1}^{D}\cos(2\pi z_i)\right) + 20 + \exp(1) \tag{1.2}$$

Figure 1.2 presents the fitness landscape associated with the function shown in equation 1.2 for a 2 dimensional problem. Except for toy-problems or benchmarks like this, the fitness landscape is not generally known.

Evaluation functions represent the problem to be solved. They all have a different execution time. The time spent in evaluating the population is often the predominant part in the execution of an EA. Typically, GP evaluation time is very computing intensive, as for symbolic regression, an individual is compared to a training set that can count thousands of cases.

Conversely, for constraints solving problems (like timetables), the evaluation of an individual consists in giving penalties every time a constraint is busted (total time, density of the solution). These computations can be very light, meaning that the evaluation of the population can represent a negligible part in the total execution time of the evolutionary algorithm.

These considerations are relatively important, because if evaluation is very expensive (Navier-Stokes equations, for instance), it is interesting to use complex variation operators that will save on function evaluations.

On the opposite, time to find a solution may be longer if such expensive operators are used, if evaluation is very cheap. In such cases, producing as many individuals as possible and counting on artificial evolution to select the best individuals may be more efficient. In Collet et al. [11], the authors investigate this idea and conclude that the use of complicated evolutionary operators such as Schwefel's adaptive mutation that efficiently reduce the number of evaluations is not justified if evaluation time is very short.

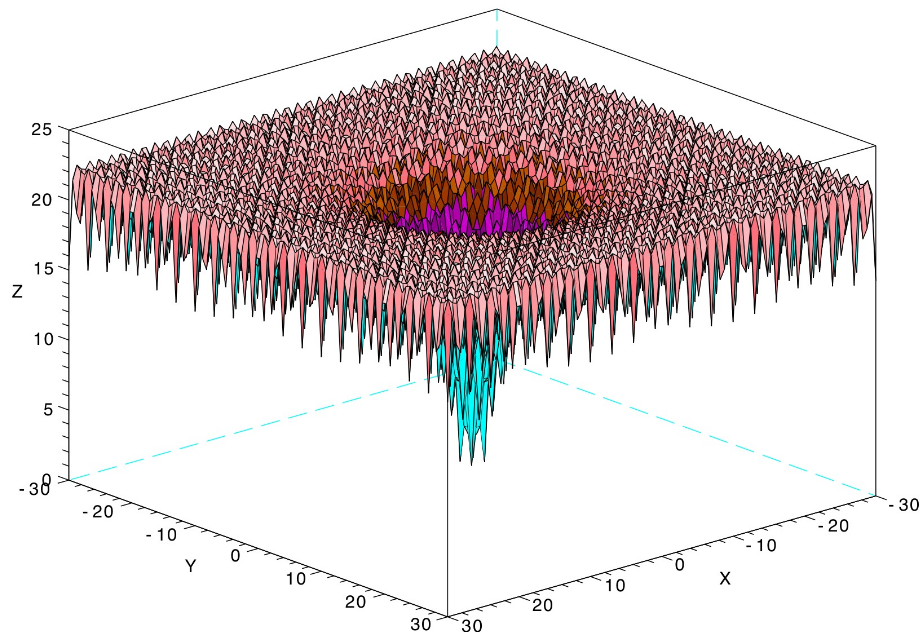


Figure 1.2: The fitness landscape corresponding to a 2D "Ackley path" evaluation function as presented in Figure 1.2.

1.3.3 Population initialization

Using a uniform random generator to initialize the values of the different genes of the initial population allows to uniformly sample the search space on a large scale.

More sophisticated strategies can be used, and expert knowledge can be inserted at this level. Of course, this will induce a bias, that could prevent the algorithm from finding a good solution if the population is initialized in a wrong part of the search space.

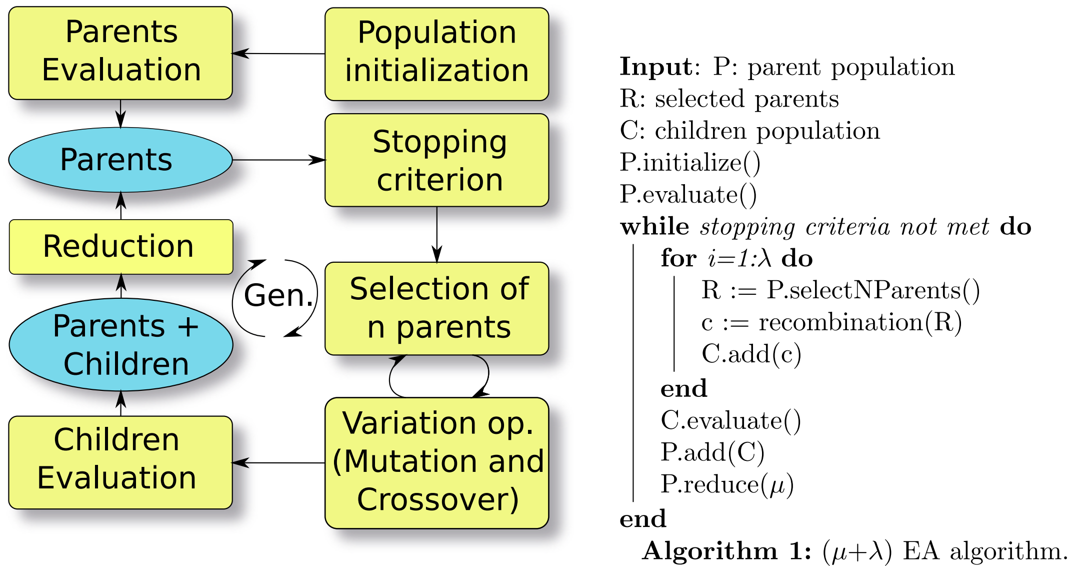
For example, if it is believed that the best solution to propel a boat is to use a paddle wheel and consequently, the initial population is exclusively initialized with paddle wheels, the optimization algorithm will optimize paddle wheels and will probably never find a propeller, that is much more efficient. The bias introduced in the initial population should therefore be greatly considered, as it may prevent from finding unexpected efficient solutions.

1.3.4 Evolutionary loop

Once an initial population as been created, an evolutionary algorithm evolves a population of μ parents in order to improve the quality of the solution, in the sense of the evaluation function. The algorithm performs a loop, as described in Figure 1.3, or in algorithm 1. The execution of a loop iteration corresponds to a generation, in order to pursue the natural analogy.

The initial population is first generated using the initialization operator described above. Then, each individual is evaluated in order to determine its adaptation to the environment (*i.e.* how well it solves the problem) and becomes one of μ potential parents to create a future generation.

The algorithm now enters the evolutionary loop. First of all, the algorithm checks whether a stopping criterion is met or not. Such a criterion can be a time limit, a maximum number of generations, or whether an individual has found a good enough solution. If such a criterion is not detected, λ children are created by repeating the next steps λ times:



Algorithm 1: (μ+λ) EA algorithm.

Figure 1.3: (μ+λ) cEA flow chart.

- Select n parents (n being the number of parents needed to produce a child).
- Create a child using one or several variation operators (as presented in section 1.3.5).
- Add this child to the children population.

When the children population is populated with λ individuals, it is evaluated and potentially added to the μ parent population. To keep a constant number of individuals, it is necessary to remove as many individuals as have just been added, i.e. μ using a population reduction operator. This operator uses a selection mechanism to either remove "bad" individuals (when few children have been created), or to keep only the "good" ones.

The children are not inserted into the parent population during a generation but only at the end of the reduction operation. So until this reduction operation is performed, the children and parents populations are distinct.

This presentation refers to one vision of evolutionary algorithms that can implement most evolutionary paradigms. Some variants may use different flowcharts, but the main principles remain essentially the same. Also, changes in certain parameters eliminate the need for some phases, as discussed in Section 1.3.8.

1.3.5 Variation operators

Variation operators are used to create new individuals. The underlying idea is that the creation of an individual from parents selected for their goodness should be able to create a "better" child, thanks to a possible recombination of what was good in both parents.

Variation operators take as input a number of parents (n in algorithm 1 and Figure 1.3) and produce a number of children as output. The variant presented in algorithm 1 uses n parents to produce one individual.

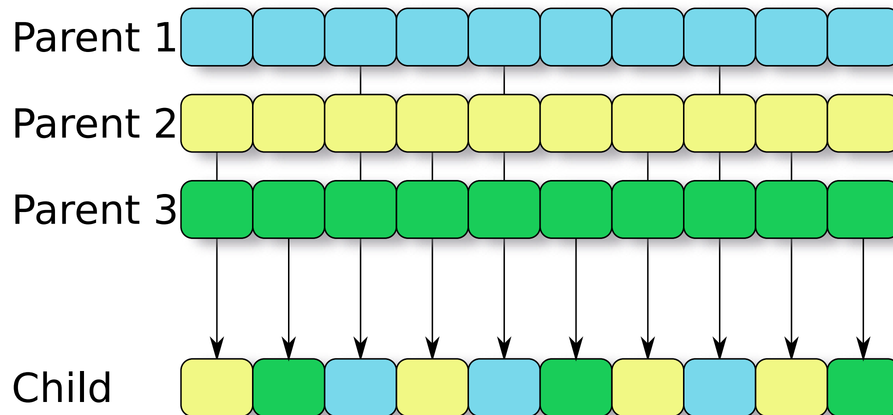


Figure 1.4: Uniform crossover between 3 parents, producing one child.

Crossover (or recombination) operator

An instance of the variation operators is the crossover operator, also called recombination operator. This operator usually takes several parents in order to produce one or several children after recombining the genes of the parents. Figure 1.4 presents a "uniform crossover" operator, that randomly selects a gene from three parents to produce a child. Many different crossovers exist, adapted to and/or inspired by the different individual representations. Indeed, this uniform crossover is initially adapted to Genetic Algorithms and their binary representation. A typical Evolution Strategy operator (using real value genes) could be a barycentric crossover, where the gene of a child is the weighted center of gravity of the parents genes, but more evolved operators do exist: the Simulated Binary Crossover (SBX [14]) creates a real valued gene whose value is close to that of one of two parents. Finally, one parent may be chosen as the only parent and the resulting child is therefore a clone of the chosen parent.

The crossover should take into account the structure of the individual. In the example of the timetabling problem mentioned above, creating a gene by mixing the value of a room and a timeslot gene makes no sense.

Crossover in Genetic Programming is also different, and is studied for example in Koza [10], Koza and Rice [15], Koza et al. [16], Koza and al. [17]. The problem here is to cross two tree-structures representing mathematical functions, for instance. This is done by creating a child that is a copy of one of 2 parents, then by grafting a subtree from the other parent into the child, at a random crosspoint. Some constraints must be met, such as limiting the depth of the tree or choosing an interesting sub-tree (in a full binary tree, one-half of the nodes are leaves).

Mutation operator

The second variation operator is mutation, which is a unary operator that usually takes the recently created child as a parameter. Mutation is intended to slightly move the child into the search space so as to possibly explore areas unreachable by a crossover only. Many people have been working on this operator, which was historically the only variation operator of Evolutionary Strategies. For a representation based on real numbers, adding a gaussian noise to one or more of the individual genes is a common method, but in ES, a self-adaptive mutation operator (where the variance of the gaussian is evolved with each gene of the genome) is used [18]. In the same way, a covariance matrix can be added to the mutation, that, with standard deviation vector can give a complete description

of the generalized multi-dimensional normal distribution to mutate the children. This very efficient auto-adaptive mutation is widely used in ES, where each individual gene has a specific variance. Many papers were published on the auto-adaptative mutation by Schwefel and Rechenberg, *c.f.* [19, 20] and also [3]. This type of mutation has also been tested on Genetic Algorithms [21].

Finally, the latest evolution of Schwefel's self-adapting mutation is the operator of the Covariance Matrix Adaptation – Evolution Strategy (CMA-ES) that still does not use a crossover operator.

In the boolean representation of individuals, the mutation is a simple bit-flip. The value of a random gene is reversed. This can be repeated for several genes per individuals.

In Genetic Programming, Koza suggests to select a subtree and replace it with a new random tree [10].

1.3.6 Selection and replacement operators

Two phases of the algorithm need to select individuals from the population: in order to create a child, n "good" parents need to be selected, and once the children population has been created it is necessary to remove λ individuals to get the global $(\mu + \lambda)$ population size back to the original parents size μ.

The fitness function evaluates the quality of individuals. It is then easy to compare the individuals and sort the population to take only the best, for instance, which is what the Truncation Selection [22] does.

However, this type of mechanism, where the best ones are used deterministically, exhibits premature convergence problems and makes no sense for parents selection (all the children would be created out of the best two individuals, for instance). In the reduction phase, keeping in the new population some individuals that are not the best, but that are still "good enough" may help to maintain some diversity into the population, which can prevent such premature convergence.

The use of stochastic operators for selection, was suggested very early (in Bäck [18], the authors attribute to Holland [23]). The idea is to use a stochastic selection, which is biased according to the value of fitness. The "Fitness Proportional Selection" or "Roulette wheel" follows this principle, by assigning a selection probability to an individual which is proportional to its fitness value.

More modern selectors include the very good Tournament Selection, which randomly picks t individuals in the population and returns the best. This Tournament Selector is widely used nowadays.

The major difference between the selection and reduction phase, is that selecting multiple times the same individual is problematic for population reduction, while it is desirable in the selection of the parents. Selecting an individual multiple times to populate the next generation will create unnecessary clones, which will result in a loss of diversity in the new population and possibly a premature convergence.

On the other hand, the repeated use of an individual as a parent is not a problem, and allows the exploitation of the"good" individuals. Furthermore, the creation of clones by this mechanism is less likely, because, as we have seen, the child undergoes a certain number of variations compared to its parents.

1.3.7 Stopping criteria

The stopping criterion allows the algorithm to stop when a certain condition is met. This test is usually related to the number of evaluations, number of generations, or CPU time consumed by the algorithm as it is often impossible to know that a global optimum has been reached.

It is however possible to stop the algorithm when a fitness threshold has been reached, such as a sufficiently good value. A population diversity measurement can also serve as a stopping criterion, as it is commonly agreed that a population trapped into a local minimum has little chance to escape

it. Starting from this principle, if all individuals are too similar, the evolution can be stopped and the algorithm can possibly be restarted. This last idea brings forth several questions about a similarity notion (genotypic, phenotypic or in a matter of fitness) and on a similarity threshold.

There are many stopping criteria and none are considered as standard ones. They are often linked to the problem being optimized, to the type of expected results (good or better than using another method), to the type of experience (evaluation of an evolutionary method or search for a real result) or to the type of algorithm.

However, the evaluation of the stopping criterion is performed only once per generation and is not considered, in the general case, as a bottleneck of an evolutionary algorithm. Still, it remains an important part of the algorithm.

1.3.8 Parameters

Finally, evolutionary algorithms have many parameters, which are often difficult to set efficiently. They heavily influence the behavior of the algorithm and they are supposed to be adapted to the problem that is optimized, otherwise they could penalize the search. The difficulty of setting these parameters is partly due to the lack of theoretical study at this level and the high correlation of theses parameters on a given effect.

Among these parameters, the first are the parent (μ) and children (λ) population sizes. The size of the parent population gives the number of solutions that compete for survival. In addition, this size gives the degree of parallelism of the search and if the population reduction is deterministic, this population size is the number of elites.

The size of the children population influences the degree of exploitation of the algorithm. Indeed, with a large children population, the algorithm strongly exploits the current parent population to find new solutions. But this parameter can also influence the selection pressure. As will be seen later, for certain paradigms, the ratio between the size of the parent and children population also influences this pressure, which is also usually tuned by tournament size.

Finally, the parameters are used to differentiate between classes of algorithms. This theory is supported by some publications [11, 12], which use these parameters to instantiate a generic model of artificial evolution to become an instance of a kind or another. The different kinds of algorithms are studied and the user will prefer to use the kind of evolutionary algorithm that seems best suited to solve his problem.

Parameter setting of an evolutionary algorithm is therefore necessarily a critical point.

Although many papers exist on this subject, all the interactions are not clearly defined and there is no generic method yet to optimize the values of these parameters.

Chapter 2

EASEA software, before 2008

Contents

2.1 **Specification** 25
2.2 **Code generation** 26
2.3 **Specification format** 27

EASEA is a platform designed to help the creation of evolutionary algorithms. Work started on this platform in 1998 and it was presented for the first time in [24].

Several ideas lie behind the creation of such a software. First, EASEA uses a unified approach, which describes most of the evolutionary algorithm variants by a single algorithm, with several parameters, which allow to switch from an algorithm to another. This approach was published in [25] and is similar to De Jong's approach, published for instance in his book [12].

This unified vision is reflected in the structure of the algorithm, which is the same for ES, GA and GP. The differences between these instances are described by parameters and individual implementation.

For example, going from a Steady-State GA to a generational GA is done by changing the number of children produced per generation and by reducing to zero the population of parents, before moving to the reduction phase.

EASEA also implements an approach where the individual is manipulated through an interface, allowing it to: initialize, evaluate, mutate and produce a new individual from several others. Apart from these four steps, the individual is fully abstract for the algorithm, which means that any individual implementing these methods can be used by EASEA.

EASEA has been running on any machine with a POSIX interface, such as on Windows, Linux, MacOSX.

Finally, EASEA was chosen to be the programming language of the DREAM European project (Distributed Resource Evolutionary Algorithm Machine, [26]), allowing the instantiation of a special island model onto thousands of loosely coupled personal computers running EASEA as a screen saver (like the SETI at home project).

2.1 Specification

EASEA takes as input a *.ez* file, which contains a specification of the evolutionary algorithm desired by the user. This file contains a set of entries in *C++*, describing how to instantiate the EASEA underlying generic algorithm.

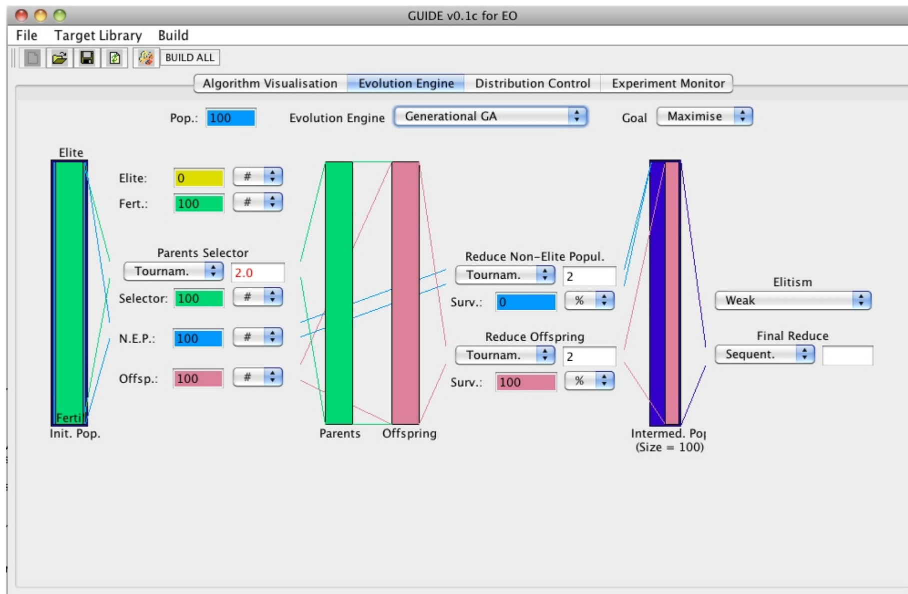


Figure 2.1: The population sizes configuration of EASEA, accessed using the GUIDE graphical user interface.

Of course, the individual is first described, beginning with its genome, which may be composed of a set of basic C-types and EASEA objects. The user writes the initialization function (which will instantiate the entire initial population), then the crossover and mutation functions which create a child from a variation of its ancestors and finally, an evaluation function that will compute the fitness value of each individual. Optionally, the user can describe how to display an individual, or let EASEA automatically create such a function, from the genome declaration.

These functions descriptions can use several keywords, which allow to access some of the fundamental algorithm objects.

In addition, some parameters of the algorithms can be specified in this *.ez* file. These parameters influence the behavior of the algorithm and make it tend towards a type of algorithm over another. These parameters are for example, population sizes, the number of generations or whether the evolution should minimize or maximize fitness.

Finally, the pre-2008 EASEA came with a graphical interface allowing an easy edition of the specification file. Figure 2.1 presents the part of the GUI that allows to configure the sizes of the populations.

2.2 Code generation

EASEA contains a code generator written in lex/yacc. This generator is based on the analysis of a skeleton file (called a template file) that describes the generic algorithm, which will be adapted according to the user's specification. It travels through the template file and inserts, after processing if necessary, portions of the specification into an instance of the template. This process is detailed in Figure 2.2

Pre-2008 EASEA had 3 skeletons. Two of them produce evolutionary algorithms coded in *C/C++* using either the EO or GAlib libraries (a comparison of identical files compiled for these two libraries was published in [27]). The latter is a Java template, designed to produce algorithms

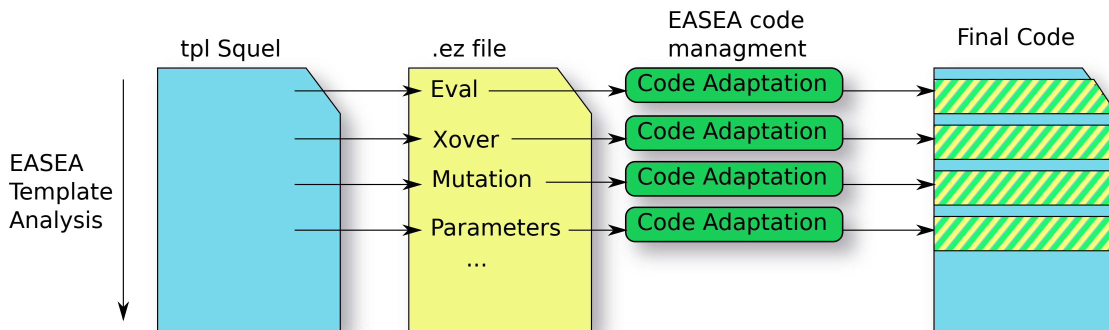


Figure 2.2: The EASEA code generation process.

for DREAM.

The use of such *C++* templates allows EASEA to produce a code which is human-editable and understandable. Indeed, the code rewriting process from the specification in *C/C++* to the *C++* skeleton is relatively straightforward. The code therefore has a standard structure, and is almost fully hand-coded (template and instantiation).

2.3 Specification format

```
\User classes:
  Element { int        Value;
            Element *pNext; }
  GenomeClass {
    Element *pList;
    int        Size; }
\end
```

EZ 1: A genome object

```
\GenomeClass::crossover:

  int locus=random(0,SIZE-1);
  cross(&child, &parent2, locus);

\end
```

EZ 2: A crossover for genome 1

An EA individual must own a genome, an evaluation and a mutation function. It must also be possible to make a crossover for the individual species being used. Thus, the user must define the shape of the genome and the functions accessing it.

Genome definition: Code snippet EZ 1 shows the definition of such a genome. Here, it is a linked list containing integers. The aim of this (toy) problem is to order the whole list in an ascending order, using an evolutionary heuristic. In this code snippet, the user describes an EASEA object named Element, which contains an integer and a pointer to the next item of the list. This is a classic linked list definition. The genome, described in the second part of the code, contains a pointer to the first list element, and a counter giving the length of the list. The latter element remains fixed during evolution.

For these two objects (Element and Genome), EASEA creates two *C++* classes and some related functions, containing a function to create an object from another (deep copy constructor),

```
\GenomeClass::initialiser:

  Element *pElt; Genome.Size=0;
  Genome.pList=NULL; // creation of
      a linked list of SIZE elts

  for(int i=0;i<SIZE;i++){

    pElt=new Element;
    pElt->Value=Random(0,i);
    pElt->pNext=Genome.pList;
    Genome.pList=pElt;
    Genome.Size++;

  }
\end
```

EZ 3: An EASEA initializer.

```
\GenomeClass::mutator:
  int NbMut=0;
  Element *p=Genome.pList;
  while (p->pNext){
    if (tossCoin(pMutPerGene)){
      //We swap the current value
          with
      // the next
      swap(p->Value,
        p->pNext->Value);
      NbMut++;
    }
    p=p->pNext;
  }
  return NbMut;
\end
```

EZ 4: An EASEA mutator.

a standard constructor, a deep destructor that reclaims all elements of the linked list, a display function, and overloads several operators such as ==, that will deeply compare two genomes. All these functions are automatically created using the description provided by the user so that even non expert programmers can use EASEA without needing to know much about object programming.

Crossover: The block describing the function provides three objects to the user. The first object is the child resulting from the crossover (where the result will be stored). It is deeply initialized as a copy of the first parent.

Furthermore, the user can manipulate two parents, selected by EASEA, to participate in this crossover. In code snippet EZ 2, the user calls a function that is defined above, in the "User defined function" section (where he can place any *C/C++* code). In addition, EASEA provides access to a random function, automatically adapted to the type that is requested (int, float, double). It is used here to choose the crossover point.

The crossover is called, only if the random toss of a coin, weighted by a parameter described below, is positive. Otherwise, the child is not produced by the crossover, but simply by copying one of its two parents.

Initializer: The initialization function is called to instantiate the parent population at the beginning of the algorithm execution. In snippet EZ 3, a loop initializes *SIZE Element*s using the constructor generated by EASEA (*new Element*) and sets the value of the item using the same random number generator as previously.

The initializer is an individual member function and access to it can be achieved by the pointer *this*, or more simply by the *Genome* object. These two expressions are equivalent and the latter is transformed into *this* by the code generator part of EASEA.

Mutator: A mutation function accesses the individual the same way as the previous function. It is called onto the current child if a random toss of a coin weighted by a parameter described below is positive.

In code snippet EZ 4, a second source of randomness is incorporated by the parameter pMut-PerGen. It weighs a random parameter, to achieve a mutation on the current gene. Here, the

```
\GenomeClass::evaluator:
  int i=0,eval=0;
  Element *p=Genome.pList;
  while(p->pNext){
    if(p->Value==++i)
      eval+=10;
    if(p->Value<p->pNext->Value)
      eval+=4;
    else eval-=2;
    p=p->pNext;
  }
  if (p->Value==SIZE) eval+=10;

  return (eval<0 ? 0 : eval);
\end

\GenomeClass::display:
  os << "Size:" << Genome.Size << "\n
      ";
  os << "pList:" << *(Genome.pList)
      << "\n";
  os << "This was MY display function
      !\n";
\end
```

EZ 5: An EASEA evaluator.

```
\Default run parameters:

Mutation probability: 1.0
Crossover probability: 0.9

// Evolution Engine:
Evaluator goal: maximize

Number of generations: 5
Population size: 5

Selection operator: Tournament 2
Offspring size: 100%
Reduce offspring operator: Roulette

Surviving offspring: 5
Reduce parents operator: Tournament 2
Surviving parents: 5
Final reduce operator: Tournament 2

Elite: 5
Elitism: Weak

\end
```

EZ 6: Parameters as defined in an EASEA file.

mutation is the exchange of the current gene with the following one. Finally, the mutation returns the number of modifications performed on the child.

Evaluator: This function evaluates the current individual, *i.e.* giving it a quality note. As for the mutation and for the initialization functions, they access the current child using the keyword: *Genome*.

The function described in the first part of code snippet EZ 5 marks each correctly placed element (absolute position), by adding 10 to the fitness value of the current individual. For the other kinds of genes, it adds 4 to the fitness value if they are smaller than their followers (relative position), otherwise it subtracts 2. Eventually, every individual whose fitness is negative will be denoted 0.

This function gradually converges to a solution of good quality, strongly rewarding the well positioned elements (+10), reasonably rewarding the correct sequences (4) and penalizing the others.

Display: This section allows the user to describe the display function for the individual. This function can be called for instance to show the user the best individual at the end of the algorithm or at every generation. Of course, the user is best placed to know what needs to be displayed, thus what interests him in the genome of the individual.

For example, in the second part of code example 5, nothing is important but the integer contents of the *Element*s. The values of the pointers are not interesting to display and the length of the list remains constant.

Parameters: This section describes the parameters of the evolutionary algorithm, as in code snipet 6. It allows to modify the following parameters:

- Population sizes for parents, children and elites.

- Numbers of parents and children that participate to the population reduction step. This allows to pre-reduce each subpopulation.
- Elitism type (weak: elites are drawn from the parents+children populations; strong: elites are drawn from the parents population only).
- Crossover probability: Determines the probability to produce a child by crossover. Otherwise, it will simply be the clone of a parent.
- Mutation probability: Determines the probability to apply mutation on the produced child.

Finally, the parameters allow the use of such a sub-type of evolutionary algorithm (steady-state, generational ...) and thus no change has to occur in the code of the user.

Chapter 3

Parallelization of EAs on classical parallel architectures

Contents

3.1 Classical architectures **32**
3.2 Classification **33**
3.3 Global parallel evolutionary algorithm **33**
3.3.1 Standard approach 33
3.3.2 Non-standard approaches 36
3.4 Distributed evolutionary algorithms **38**
3.4.1 Standard approach 38
3.4.2 Hardware oriented approach 40
3.5 Cellular evolutionary algorithms **40**

Parallelizing Evolutionary Algorithms is not a new idea. Indeed, in Cantú-Paz [28], the author cites Holland [29, 30], where Holland presents a class of parallel machines, that could run an evolutionary process, which he recognizes as parallel. In the same survey, Cantú-Paz gives Bethke [31] as likely to be the first publication about parallel evolutionary algorithms. This paper describes two algorithms, (similar to what is now known as the Master-Slave model), and compares them with a Gradient-Based optimizer. Later, Grefenstette studied 4 different implementations of parallel evolutionary algorithms [32], 3 of which are variants on the Master-Slave model, the last one being a Distributed EA, as claimed in Cantú-Paz [28].

If a population size needs to be adapted to a given problem [33], determining its size is a difficult task. Nevertheless, complex problems require a large population. Indeed, evolutionary algorithms that optimize these kinds of problems search the space more effectively if a large number of samples is available (individuals in the population). But unless one can use infinite computing power, the use of a large population leads to prohibitive computation times.

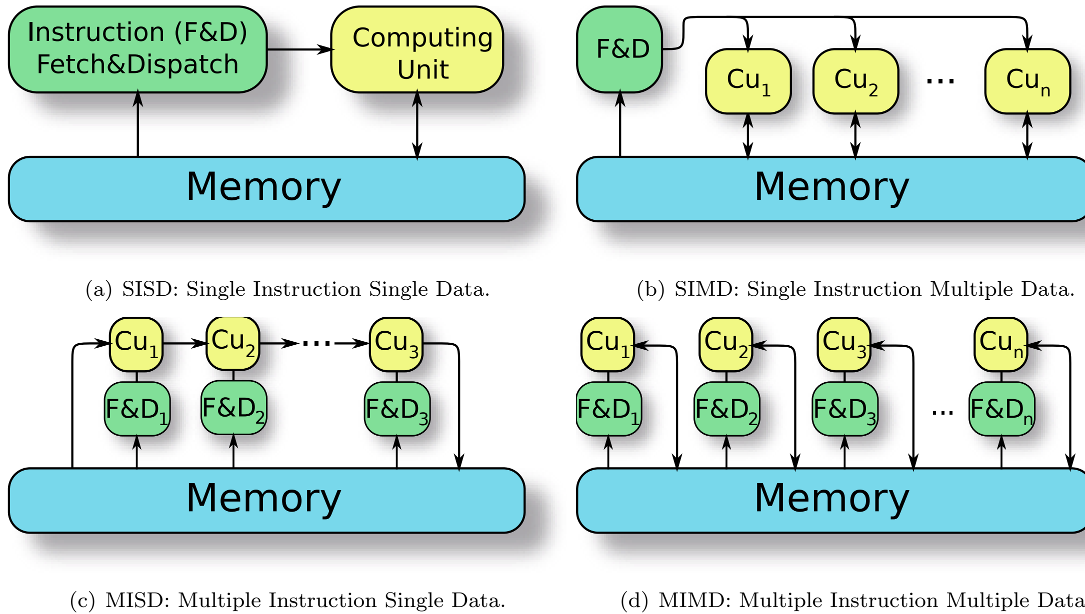
The use of simulators can also turn a small sized algorithm into a greedy one, with respect to computing time. Genetic Programming is a good example of evolutionary computation which requires high computational resources. In Koza et al. [34], where the authors optimize an assembly of lenses using an optical simulator, a population of 75,000 people needs to be used and is spread over 150 computing nodes.

All these considerations oriented many researchers towards parallel architectures in order to optimize complex problems.

3.1 Classical architectures

Parallel architectures have long been developed to help overcome the limitations faced by standard processors. Interest in these architectures has been changing in recent decades, depending on serial processor performance. As sequential processors became more and more efficient, researchers lost their interest in these complex parallel machines, but the recent stagnation in single processor clock frequency rekindles the interest in parallel architectures.

There are many kinds of parallel architectures. A very well-known classification is Flynn's taxonomy (SISD, SIMD, MISD, MIMD presented respectively in Figures 3.1(a), 3.1(b), 3.1(c) and 3.1(d)), based on their capability to apply one or several instructions on one or several pieces of data at the same time. However, this taxonomy is a bit simplistic, as it does not necessarily capture all the features of parallel machines. Consequently, it is rare for a complex modern architecture to be classifiable as-is in one of Flynn's categories.



(a) SISD: Single Instruction Single Data.

(b) SIMD: Single Instruction Multiple Data.

(c) MISD: Multiple Instruction Single Data.

(d) MIMD: Multiple Instruction Multiple Data.

Figure 3.1: Flynn's taxonomy of parallel systems.

Indeed, most modern architectures implement several of these paradigms, such as modern multi-core processors. These are MIMD because of their multi-core implementation, but also SIMD because of their instruction pipelines and their vector instruction set.

Another classification, which focuses on the memory architecture is possible. In fact, the parallel machines implement complex memory systems, sometimes shared between processors or distributed to each of them.

These systems often have an impact on the implementation of parallel algorithms, because obviously, they must run on an architecture, and too much difference between implementation and hardware support will not perform satisfactorily.

3.2 Classification

The parallelization of evolutionary algorithms follows several main ideas, the first being that evolutionary algorithms are inherently parallel. Indeed, in these algorithms, the population contains individuals, which are independent of one other, and only few steps of the algorithm require an interaction between them, mainly the crossover and the reduction of the population [35].

According to Cantú-Paz [28] and Alba and Tomassini [36], Grefenstette [32] is the first to introduce several types of parallelism of evolutionary algorithms. In this paper, the author implements an algorithm for synchronous master-slave, another asynchronous one, and coarse-grained model.

These two categories are now broadly used, especially in Cantú-Paz [28], Alba and Tomassini [36]. They are based primarily on population structuration. In the global model, one population is maintained and all processors access the same pool of individuals. We will see this category in more detail in section 3.3.

In the second category, the population is structured into subsets and is called Distributed Parallel Evolutionary Algorithm. This category will be presented in Section 3.4

Finally, Section 3.5 presents Cellular Evolutionary Algorithms, which push further population structuration, by considering an individual and its immediate neighborhood as a subpopulation.

But these categories are not strict. There are also mixed models, where several levels of parallelism are present.

3.3 Global parallel evolutionary algorithm

3.3.1 Standard approach

Evolutionary algorithms are major CPU time consumers. In some applications, much of the CPU time is spent in population evaluation. The fitness function, as presented in section 1.3.2, evaluates an individual in the population, by rating its quality with respect to the current problem to solve. The evaluation of a particular individual is, in general and the most common case, independent of the evaluation of the other individuals. Indeed, in order to evaluate an individual, one only needs to access its genome in a read-only manner. Finally only the fitness value must be returned.

On a parallel architecture, it is possible to cut the population (children or parents) into subsets and assign each subset to a processor in order to be evaluated. This parallelization does not change the behaviour of the algorithm. Only evaluation time is reduced by using several processors to evaluate the population.

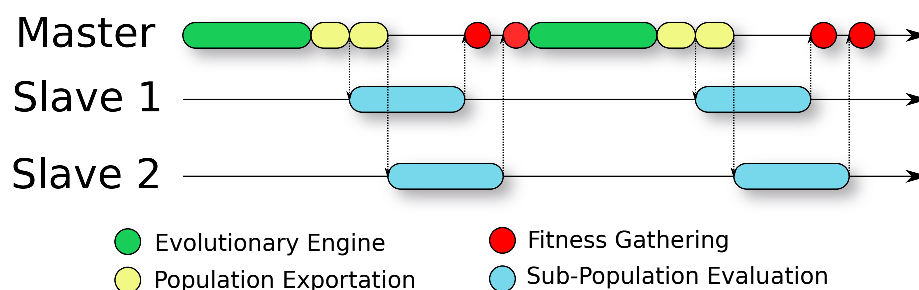
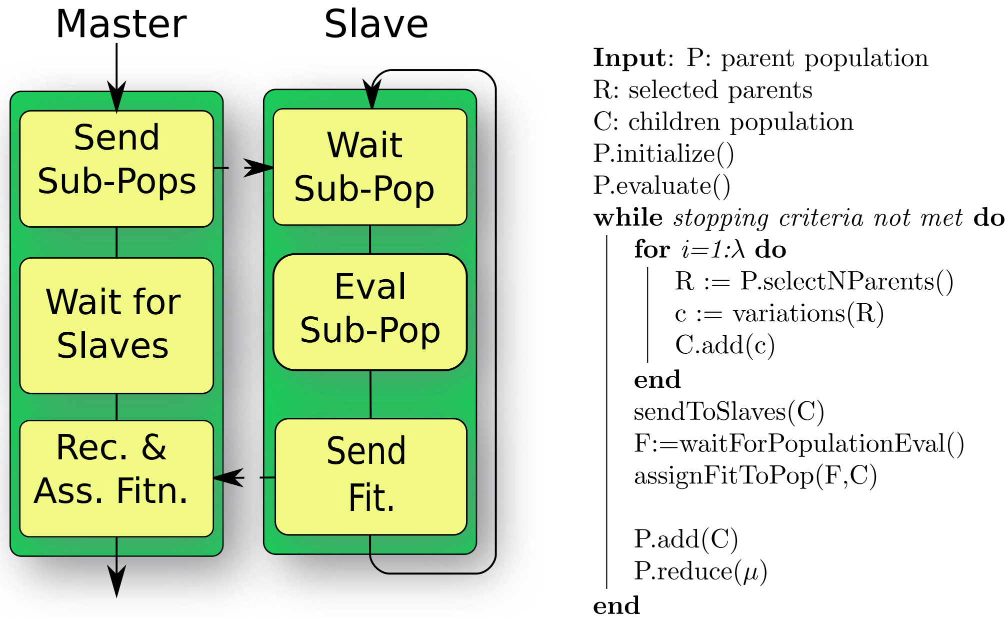


Figure 3.2: An execution scenario in master-slave model.

The population is split into subpopulations that are sent to nodes or processors that perform the evaluation. Once this is done, the fitness values are sent back to the main node, which runs

the evolutionary loop and assigns the just-received fitness values to the individuals. After these parallel evaluation steps, the evolutionary loop can return to its normal sequential operation. This type of parallelization is shown in Figure 3.2 and corresponds to the algorithm presented in 2. The evaluation phase can be presented as in Figure 3.3.



Input: P: parent population
R: selected parents
C: children population
P.initialize()
P.evaluate()
while *stopping criteria not met* **do**
  for i=*1*:λ **do**
    R := P.selectNParents()
    c := variations(R)
    C.add(c)
  end
  sendToSlaves(C)
  F:=waitForPopulationEval()
  assignFitToPop(F,C)

  P.add(C)
  P.reduce(μ)
end

Algorithm 2: EA using M-S model.

Figure 3.3: Evaluation in M-S model.

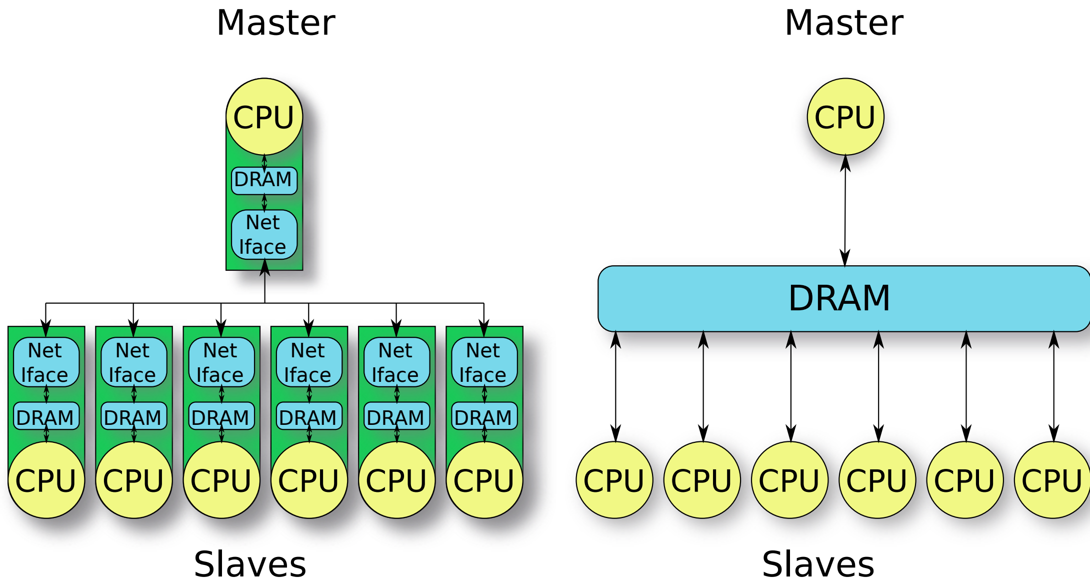
This parallelization method is called: Global Parallel Evolutionary Algorithm, Farmer or Master-Slave Model (because of the processor hierarchy) or Parallel Panmictic Evolutionary Algorithm. It is one of the first four proposals for parallel algorithms that [37, 36] attribute to [32]. Parallelization using the master-slave model has the advantage of being comparable to an equivalent sequential algorithm (conducting the evaluations sequentially rather than in parallel would lead to an identical algorithm).

This simplifies the comparison between sequential and parallel algorithms and the calculation of the speedup in particular can be done on a single run. Last but not least, all experience and theory built on sequential algorithms can be applied to this kind of parallel algorithm.

In addition, this model does not impose a specific hardware architecture, as there exist variants on many types of machines, which are either shared or distributed memory, MIMD or SIMD processors. Figure 3.4(a) shows the implementation of such a model on a distributed memory machine, while 3.4(b) uses a shared memory machine.

However, the speedup achievable with such a model is dependent on the problem and the hardware architecture, as it is limited to the parallelisable part of the algorithm. Indeed, if the evaluation of the population does not take an important part of the total execution time, the speedup would be limited, because only evaluation is accelerated and no gain will be made on the rest of the algorithm. This is Amdahl's law, which states that the maximum achievable speedup in an algorithm that contains sequential and parallel parts is limited to the proportion between the different parts. If 10% of an algorithm is sequential and 90% parallelisable, the maximum possible speedup due to infinite parallelization will only be 10.

This is theoritical, as communication time between the parallel nodes is also important. In



(a) Master-Slave model onto distributed memory architecture.

(b) Master-Slave model onto shared memory architecture.

Figure 3.4: Master-Slave model for EA parallelization

[38], the author gives lower bound of a master-slave parallelization of an algorithm, based on the evaluation time of an individual, the communication time between nodes, and the number of nodes. This term is used to calculate the optimal number of nodes to be used on a given problem. However, the communication time is estimated to be linear with the size of the individual and the rest of the algorithm is ignored, in terms of computation time. This is consistent with the target architecture in this work, which involves using a physical network between nodes.

This parallelization method offers several advantages. The first is of course, as was noted above, to be totally comparable with the sequential algorithms.

The second is the portability of this model, which can be implemented on a large number of parallel architectures. It is possible to use a distributed memory architecture if there is a way to transfer the subpopulations to the computing nodes.

On shared memory architectures, no transfers are necessary. In this last case, the different processors evaluate their own subpopulation directly into the main-system memory.

A final advantage is the absence of additional parameters compared to sequential algorithms. Indeed, only the number of nodes must be specified, but it does not affect the behavior of the algorithm.

However, this model also has disadvantages: it is a synchronous model, where nodes work step-by-step. The master node sequentially sends subpopulations to the slave nodes making its interface with the slave nodes a bottleneck. Then the master node is idling as it waits for the slave nodes to compute the fitness values even if it is quite possible to assign it a subset to evaluate. The slave nodes send the results to the master node as soon as they are computed. Again, the communication interface of the master node becomes a bottleneck. Finally, the master node goes back to executing the standard sequential algorithm, meaning that the slave nodes are then idling.

So there are several synchronization, exchange and wait operations in this model. The critical points are the speed of the communication interface of the master node, and the ratio of time

between the evaluation of the population and the rest of the algorithm (evolutionary engine).

The sequential part of the algorithm bounds the maximum speedup (*cf.* Amdahl's law above). If this sequential part is assumed to be fast in EAs, as is assumed in [38], it may also become important if the evaluation time of an algorithm is short.

Finally, parallelization using this model allows to share computing resources, but it does not take advantage of the size of the distributed memory because the population is still stored on the master node. However, it makes it possible to use an asymmetrical architecture, where the primary node is a powerful machine with a large memory space, and the slave nodes are smaller with potential access to less memory.

This approach is obviously applicable and profitable in the cases where the evaluation function takes an important part in the algorithm execution time.

3.3.2 Non-standard approaches

There are implementations of less standard panmictic algorithms. In this case, they lose their comparability with the sequential algorithm, but the underlying principle still remains the same.

Distributed panmictic EA

In [35], the authors implement a panmictic distributed algorithm. Two criteria seem to motivate this work. First, the acceleration induced by a multi-node parallelization is naturally sought. But storing a large population on a distributed memory space can allow to tackle larger problems.

This algorithm performs a large number of individual exchanges between the nodes, before the global steps of the algorithm. Indeed, in a traditional evolutionary algorithm, two steps use the whole population, specifically crossover and reduction. To implement this panmictic distributed algorithm, a local population is first randomly cut into P sets (where P is the number of nodes), before crossover and reduction are performed. Each step is sent to one of the nodes of the parallel machine and incorporated into the local population. Thus, each individual has the opportunity to be crossed (or compared) to any other individual in the population. Influence of individuals is no longer restricted to a subpopulation.

The authors perform a population exchange before the crossover and K_S (a constant depending on the selection method) before the reduction, K_S being dependent on the number of individuals necessary for the selection operator.

Starting from a $\gamma = T_f/(K_S + 1)T_c$ ratio between the evaluation time of an individual and the communication time T_c of the subpopulations, they analyze the theoretical speedup of their implementation. In addition, they are able to get an optimal and maximum number of nodes to be used according to this ratio. They get a theoretical speedup of about $10\times$ for 30 nodes, with a ratio of 1, $50\times$ with 100 nodes for $\gamma = 10$ and $100\times$ with 300 nodes and $\gamma = 100$.

Again, this model is strongly linked to communication and evaluation function time. However, being able to compute the maximum number of nodes (before the speedup falls), or the optimal number of nodes, (before the efficiency drops and the speedup becomes less than linear) is very interesting. The ability to implement an algorithm similar in principle with a sequential algorithm that increases the available memory space is also very positive.

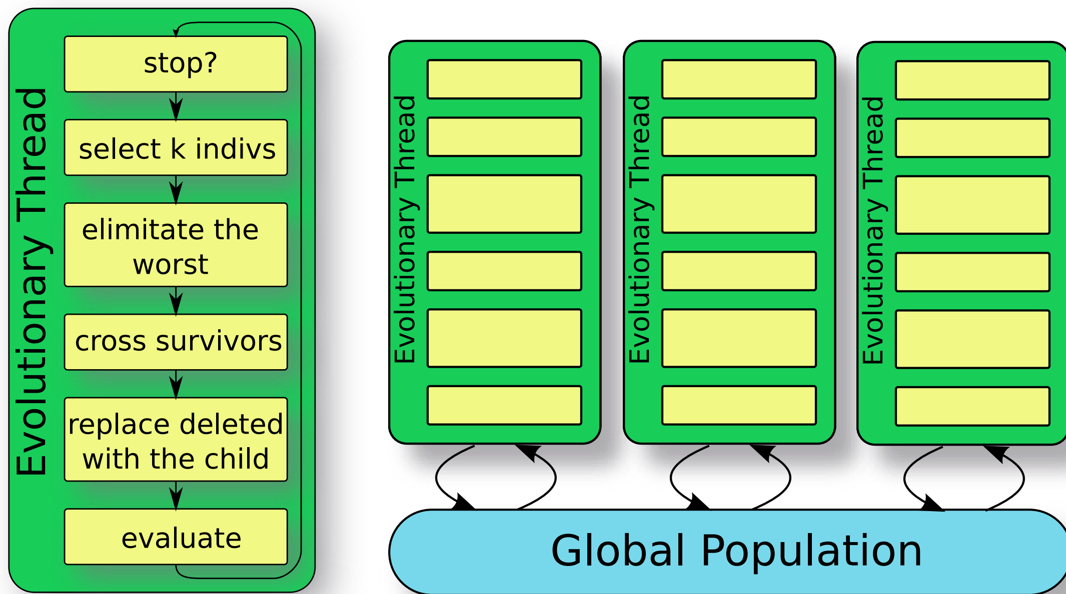
Asynchronous panmictic EA

Parallel evolutionary algorithms are developed considering the available hardware architectures. Starting from this observation, Golub *et al.* published in [39, 40, 41] an algorithm that focuses on the use of shared memory parallel machines of the 2000's. During these years, parallel desktop

architectures emerge with 2 or 4 cores. Using a Master-Slave model provides useful speedup in some cases, but exhibits restrictions for certain types of algorithms.

Typically, synchronizations are equal to a loss of efficiency. Synchronization of reproduction between species is mostly absent in Nature, even though seasons play a large role. Evolution is not really synchronous and there is no synchronous reduction step of the populations.

Therefore, Golub *et al.* propose an algorithm based on thread mechanism, as they share the same memory space. Here, each thread starts by selecting an individual to replace and produces the child that will replace it right away. The authors use an anti-tournament (a tournament selecting the worst individual among its set) of size k, where the loser is replaced, while others become parents.



(a) An evolving thread.

(b) Population and workers.

Figure 3.5: Asynchronous panmictic EA principle.

This algorithm is indeed very suitable for a shared-memory architecture, with a restricted number of processors (threads). The absence of synchronization and communication between cores allows a speedup which is linear with respect to the number of cores, at least in terms of number of evaluations. However, even if the algorithm is no longer exactly comparable with the sequential one, the authors note that the quality of results is also further away from the optimal solution. Indeed, there is a likelihood that multiple threads select the same individual to be replaced. In this case, all threads that have selected this individual prior to the last thread work in vain.

The authors calculate this probability, which depends on the number of threads, the size of tournaments used in the selection and the size of the population. Using this probability, they can increase the number of iterations for each thread (and therefore the number of individuals produced by each processor) and find result values which make this algorithm comparable with the sequential one. Moreover, they calculate the theoretical speedup of the implementation according to the same parameters as before.

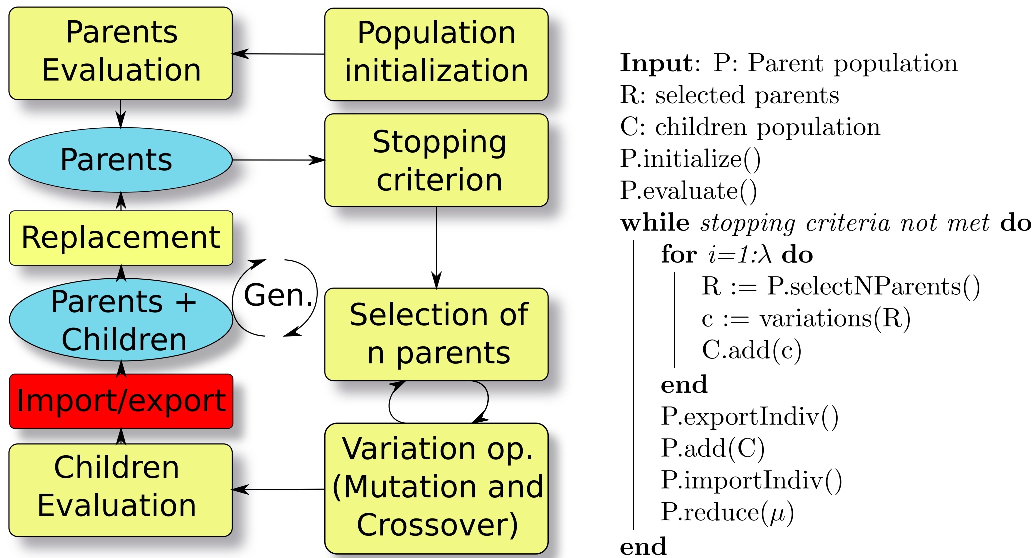
3.4 Distributed parallel evolutionary algorithms (island model)

3.4.1 Standard approach

The use of a single population, where all individuals can potentially mate to produce a child is of course far away from the natural model. In addition, we have seen that this requires either the use of a classical centralized master-slave model, or very frequent exchanges between the subpopulations. This imposes constraints on the hardware, which may not be fulfilled by the architecture being used. These choices are effective only under certain conditions.

In [36], the authors give Grefenstette [32] as the first researcher describing a new architecture for distributing an algorithm population on a number of computing nodes, and organizing migration between these populations. This model, in addition to being close to the natural concept of *deme*, is suitable for distributed memory architectures, which are widely used in high performance computing. This parallelization model is called distributed evolutionary algorithm (dEA), island model, coarse-grained or distributed population model.

Each node runs a classical evolutionary algorithm, but exchanges from time to time some of its (good) individuals with the other nodes in the network. The modification of the classical algorithm is presented in figure 3.6 and in algorithm 3.




```
Input: P: Parent population
R: selected parents
C: children population
P.initialize()
P.evaluate()
while stopping criteria not met do
    for i=1:λ do
        R := P.selectNParents()
        c := variations(R)
        C.add(c)
    end
    P.exportIndiv()
    P.add(C)
    P.importIndiv()
    P.reduce(μ)
end
```

Algorithm 3: Island model.

Figure 3.6: Island model algorithm.

These changes allow a parallelization with little or no synchronization between the nodes and few transfers between them. Moreover, the subpopulations can be seen as a single population distributed across multiple nodes, therefore exhibiting a structure. This distribution can increase the memory available to store the population.

Algorithms using an island model also behave differently than standard sequential evolutionary algorithms. The use of semi-isolated subpopulations affects the movement of the individuals in the search space. The island model can develop subpopulations, which contain different kinds of solutions, before the migration of an individual allows one island to be made aware of a solution found in another island. In other words, according to their initialization, the islands can each focus on a subset of the search space, by preventing the influence of good individuals found in another

part of the search space.

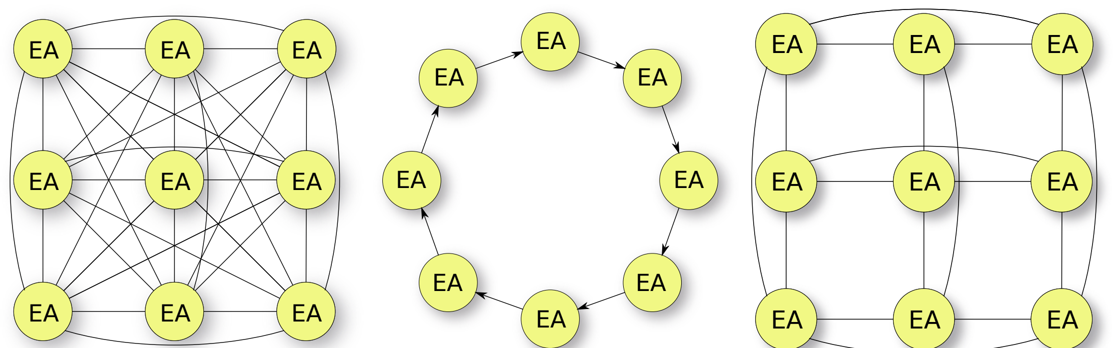
However, the use of such a model requires the introduction of new parameters, sometimes complex, and for which the effects are not always clearly defined. The number of nodes used in the algorithm is one of them. While it is highly dependent on the hardware architecture being used, it affects either the size of the overall population or the size of each subpopulation. This affects the number of search islands that sample the search space and modify the behaviour of the evolutionary algorithm.

The migration rate is also a new and important parameter. It sets the exchange rate of the individuals between the islands, and therefore influences the connectivity/isolation between them.

This leads to the last parameter, which is the connection topology of nodes. There are several common topologies, such as a complete connection, as in Figure 3.7(a). Here, each node can communicate with any of the other nodes.

A connection ring can also be used, where each node communicates with its neigbour, as shown in Figure 3.7(b). A bidirectional ring can of course be implemented, where a node communicates with its two neighbours. In both cases, individuals are diffused to other nodes along the network.

Finally, a hybrid approach between the previous two is to use a multi-dimensional torus, as shown in Figure 3.7(c).



(a) dEA using a fully connected network topology.

(b) dEA using a ring network topology.

(c) dEA using a neigboring network topology.

Figure 3.7: Example of connection tolopogy for distributed EA.

It is to be noted that logical topologies are often adapted to the physical topologies used by the underlying parallel machines. However, in the case of an island model, communication is so sparse that hardware connectivity is not an argument.

The island model, if it is scalable due to its sparse asynchronous communication, is difficult to compare with the sequential model, to the contrary of the master-slave model. Here, parallelization implements a different algorithm and a run does not produce the same results depending on the number of nodes used.

However, it allows a quasi-linear speedup in terms of evaluations, with respect to the number of involved nodes. But it is necessary to compare the quality of the results obtained under a statistical analysis. Using this method, different works show supra-linear speedup [42, 43]. Indeed, the efficiency of such parallelizations are greater than 1 on these problems, but this effect is difficult to fully characterize.

The fact that the island model maintains a larger diversity within the population, because of the relative isolation of its subpopulations is a common explanation for its efficiency. Indeed, using a

relatively low migration rate, subpopulations can explore different areas of the search space, without being attracted to the solution which seems interesting in another subpopulation.

In Whitley et al. [44], the authors advance the hypothesis that the island model is more efficient on linearly separable problems. Subpopulations may be more likely to solve a different subset of the problem, while migration and crossover would be responsible for recombining a global solution.

Finally, in Cantú-Paz [45], the author explains the observed super-linear speedup by an increased selection pressure, induced by the migration of good individuals. This idea seems to be briefly confirmed by [44].

3.4.2 Hardware oriented approach

The island model comes from two observations, one related to the natural mechanism of demes and the other which concerns the architecture used in cluster parallel machines. Following this last idea in Andre and Koza [46], the authors use a network of transputers, running a genetic programming algorithm, which is distributed over the multiple nodes. Transputers are autonomous cards, which have a processor, a memory and four communication interfaces. The authors evaluate the performance of a transputer to be equal to one-half of one of their Intel 486 66MHz machines.

Each transputer runs 4 processes, physically scheduled by the card. The first is used to monitor the local nodes (start, statistics, etc...), the two following processes are responsible for the import/export of migrants and the last runs the genetic programming algorithm.

The islands (and thus the transputers) are organized in a grid, whose nodes are connected logically and physically to their four neighbours. The actual implementation uses 64 transputers to run the evolutionary algorithm, two more being used for monitoring and debugging.

The authors take advantage of this implementation to analyze the optimal migration rate for their problem and they estimate that rate to be 8% of the population. Using this parameter, their implementation performs better than panmictic implementations for a computational effort almost halved. However, the results are reversed for the other tested migration rates, where they are less good (or almost equal) with a higher computational effort.

This shows the difficulties related to set this parameter and the sometimes negative effects, that an improper assignment can cause.

3.5 Cellular evolutionary algorithms

Cellular evolutionary algorithms (cEA) are algorithms using a structured population, where the population notion is greatly reduced compared to the island model. This type of parallel algorithm is also called diffusion model, fine-grained distributed model or massively parallel evolutionary algorithm. The first detailed implementation cited by Alba and Dorronsoro [47] is Robertson [48]. This implementation is tried on a Connection Machine 1 with 16K processors.

Then, Gorges-Schleuter and Mülenbein present ASPARAGOS in Mühlenbein et al. [49], Gorges-Schleuter [50, 51], Mühlenbein [52], Mühlenbein [53], Gorges-Schleuter [54]. They apply their algorithm on the TSP, using a local optimization (2-repair). Alba and Dorronsoro [47] consider this algorithm as the first hybrid-type cEA.

Whitley proposes to use the term “cellular GA” in Whitley [55], where he makes the connection between this kind of evolutionary algorithms and the cellular automata.

These algorithms were originally developed to stick to parallel architectures like the SIMD ones that flourished in those years. But they were soon used on sequential machines, for comparison with the natural paradigm.

The cEA uses a strong notion of neighborhood. According to this principle, the algorithm is distributed in a space (often 2D), where a node interacts with its close neighbors. Figure 3.8 shows

a sample grid and WEST neighborhood-type.

An individual is updated this way: two parents are selected in the neighborhood of the individual, they produce a new child in the usual way (by crossover and mutation), and if the produced child is better than the current individual, it replaces it. Each individual of the population is updated synchronously or not. In Tomassini [56], the author describes four strategies for asynchronously updating the individuals: fixed line sweep, fixed random sweep, sweep New Random and uniform choice, which are classified from the more deterministic to the more random one.

With such a structuring, the overall population appears to be global during the run, but very local during a generation. This feature allows effects on the migration of individuals within the population, which are mainly configured by the shape and the size of the neighbourhood. These effects range from colonization of the population by a good individual, to obtaining multiple good individuals within the same population [57].

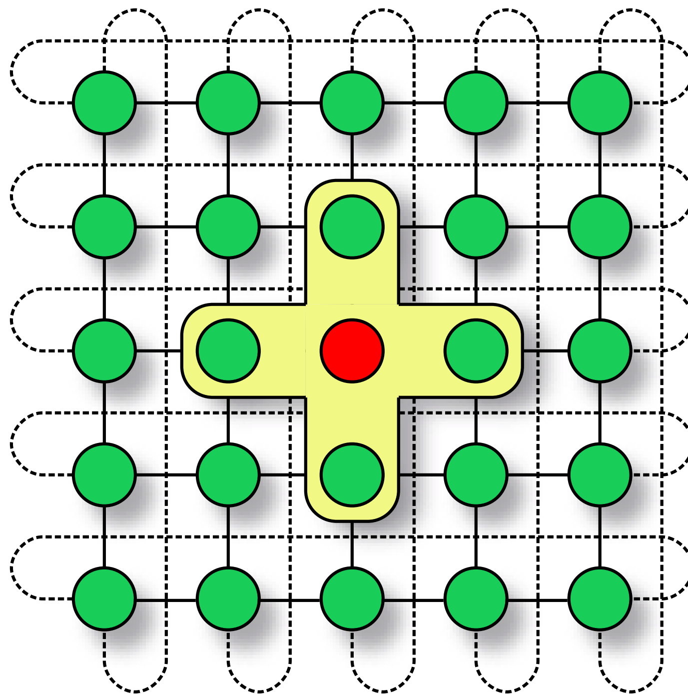
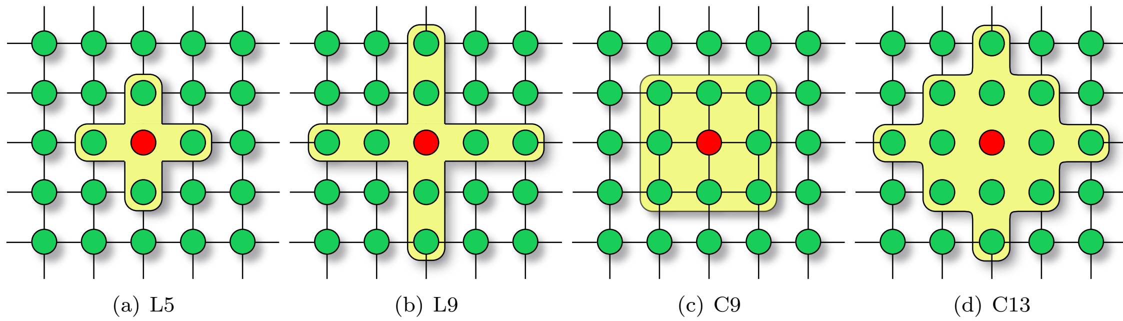


Figure 3.8: An example of cEA population, organized as a 2D torus, with a 4-neighbouring around the center individual.

If the parameters such as migration rates and topology of the islands are removed, the shape and size of the neighborhood seems to play an important role for cEA. Figure 3.9 presents four different neighbourhoods, used in a 2D grid. There are many others, for example used in populations that are distributed on a ring (1D-5, 1D-9).

The shape and size of the neighbourhood in a cEA algorithm appears to influence the selection pressure of the algorithm. Eklund [58] examines several neighbourhood shapes and sizes, on a genetic programming algorithm optimizing a symbolic regression problem. No generic configuration emerges as being the best, but the author concludes that there exists a link between population size and neighbourhood size and shape onto the selection pressure.

In Sarma and De Jong [59], the authors also analyze different neighbourhood sizes and shapes, this time by studying the spread of the best individuals. To do this, only the cloning operation is applied to the population (no mutation or crossover) and the authors use two different selection



(a) L5 (b) L9 (c) C9 (d) C13

Figure 3.9: Examples of neighbouring used in cEA.

operators (Fitness proportional selection and linear ranking). By using this measure which looks like a takeover time, the authors hope to observe selection intensity.

If the various neighbourhoods seem to relatively act as expected, *i.e.* a large neighbourhood shows a larger selection intensity than a small neighbourhood, they point out that C13 and L9 have a very similar behavior. They conclude that the neighborhood shape must also have an influence on the selection intensity. They define a measure, called *radius*, that can capture these two variables (size and shape), and which measures the dispersion of the points around the average center of the neighbourhood. Using this measure, they are able to classify the tested neighbourhood operators in the same order as in their experiments.

In Alba and Troya [60], the authors put forward the idea that the selection intensity is not only guided by the radius, but also by the size of the grid containing the population. They thus define a ratio between the radius of the neighbourhood and the radius of the grid. Using differently shaped grids (20×20 , 10×40 and 4×100), they show that this ratio has an influence on selection pressure. They also implement an algorithm that dynamically changes the shape of the grid, at the middle of the computing time of the square grid algorithm.

Chapter 4

GPGPU Hardware

Contents

4.1 History . . . 43
4.2 System overview . . . 44
4.3 Hardware part . . . 45
4.3.1 Computing elements organization . . . 45
4.3.2 Memory hierarchy . . . 48
4.3.3 Register . . . 48
4.3.4 Shared memory . . . 49
4.3.5 Global memory . . . 49
4.3.6 Considerations about memory spaces . . . 49
4.4 GPGPU Software . . . 50
4.4.1 Compilation chain . . . 51
4.4.2 Thread organization . . . 51
4.4.3 Threads scheduling . . . 52

4.1 History

Graphical rendering hardware appeared early, in the form of dedicated machines using micro-coded rendering algorithms [61, 62]. These kinds of architectures already allowed some programmability, even if their particular architectures make them very specific for 3D rendering computations.

If the first GPU (Graphical Processing Unit) in a personal computer appeared in the Commodore Amiga in 1985, it was the 90's that saw the explosion in sales of "3D cards" for personal computers. These years also saw the introduction of 2D/3D rendering libraries, such as OpenGL, DirectDraw, Direct3D, first computed in software by the central processor. As these APIs became more popular, more and more functions were directly hardcoded into the 2D/3D rendering chips.

From a research perspective, some 3D rendering machines began to be used for non-graphic processing. In Harris et al. [63], the authors cite Lengyel et al. [64] as the first "Generic Purpose" application of such hardware, where the authors use the rasterization process to schedule robot motions. In 97, Kedem and Ishihara [65] use PixelFlow, a graphic rendering SIMD machine, to crack the encryption of Unix passwords.

For graphics cards, parts of the rendering pipeline are hardcoded into the chip, but some units become programmable and allow the introduction of more and more advanced graphic processing

algorithms (lighting, shading, etc.). When such a customized treatment is applied before rasterization of the points, it is called a vertex shader, and if is is done afterwards, directly on the pixel, it is referred to as a fragment or pixel shader. The 2000's see the emergence of "shading-languages", one early implementation being the "RenderMan Shading Language" (RSL) developed by Pixar. It is developed within a communication protocol that allows interaction between modeling and 3D rendering softwares. It is then seen as a 3D-equivalent to PostScript.

Later, when the conventional 3D rendering pipeline of graphic cards includes programmable parts, shading languages are further developed, such as ARB (an assembly-type language associated to OpenGL) and high level languages such as: GLSL (OpenGL Shading Language), Cg (C for Graphics from Nvidia) and HSLS (High Level Shader Language from Microsoft). These languages and their implementations by manufacturers allow some researchers to use GPU cards for generic computing, some of them even being unrelated to 3D rendering [66]. This type of usage would lead to the emergence of the term GPGPU (Generic Purpose Graphical Processing Unit).

The latest phase of this development begins with the unification of vertex and pixel shader computing units. Such unified units thus exhibit more generic computing capacities than before and manufacturers rush in, with CUDA and CTM in 2007, OpenCL in 2008, which allow the programmer to write algorithms in non-3D related languages, to run on graphic processing units.

CUDA is the development framework created by NVIDIA for their graphics cards. The hardware continues to evolve, and if the programs had strong constraints when executed on early GPGPU architectures (such as on *G8x*), with for example the impossibility of making a function call, such constraints tend to be relaxed with the newer versions. We can note, for example, the availability of random generators, a dynamic memory allocation manager, 32 and 64 bits computing capabilities and the ability to call functions on most modern architectures (*gf1xx*).

Finally, the use of GPUs for generic computation is a relatively new concept. However, these architectures are largely based on older principles (SIMD, Simultaneous Multi-threading, etc.). This resemblance to older architectures may explain why these architectures are so appealing for researchers. Another explanation is obviously the *performance/cost* ratio and the theoretical peak computing power developed by these processors, which, at the price of a very particular architecture that we will detail below, reaches values that are inaccessible to conventional processors.

4.2 System overview

A GPGPU node includes a host part, consisting of one or more conventional processors (CPU) and their associated memory, to which is added one or more GPGPU cards (computing devices). Some motherboards allow to install up to 4 GPGPU cards into one host machine. In order to execute a program on such a node, one must code a CPU program, which embeds one or more GPGPU-compiled executables.

This kind of system is automatically exploited by 3D rendering softwares (video games, or other 3D programs) using 3D programming APIs such as OpenGL or DirectX. Unfortunately, because they are more generic, CUDA programs do not automatically parallelize over several cards and the parallelization has to be done by the programmer.

A CPU + GPGPUs system, as in Figure 4.1, can be seen as the integration of several quasi-autonomous systems around a communication bus. Indeed, each GPGPU has its own memory, its own processor and an interface to the host memory. This interface allows to exchange data with the GPGPU, to launch a program on it, and to signal the termination of GPGPU tasks, all from a host CPU program.

Apart from these exchanges, the two types of processors independently co-exist inside the same system. While the CPUs share the same memory space, each GPU card has its own. All exchanges between the memory are done through software DMA transfers controlled by the CPU.

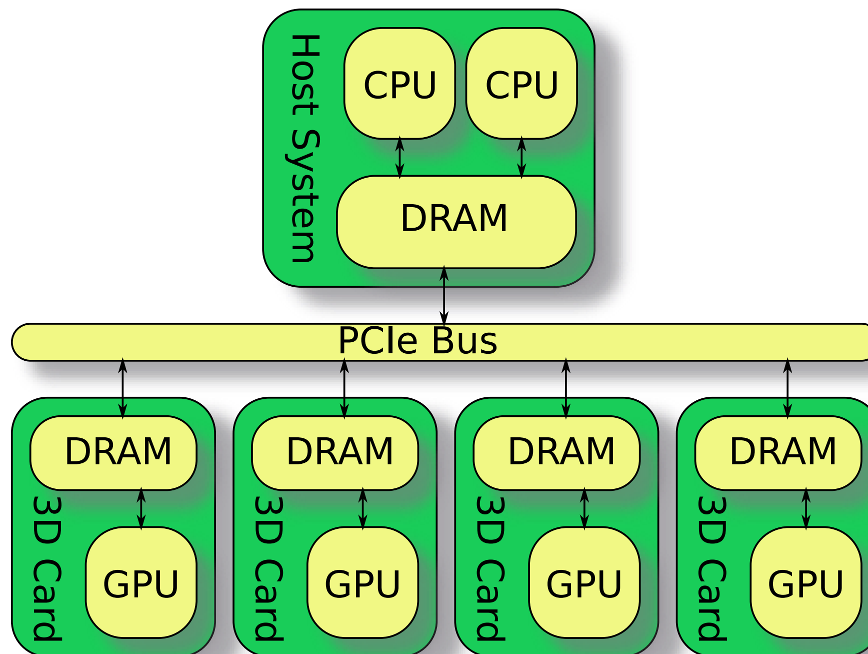


Figure 4.1: A schematic view of a CPU-GPGPU system.

4.3 Hardware part (valid in 2011)

4.3.1 Computing elements organization

Computing cores

NVIDIA GPGPUs contain several hundred cores. The chips are constrained in size and therefore in number of transistors, for problems of delay and manufacturing, as are other kinds of CPU chips. Knowing that they have a size and transistor thinness comparable to conventional processors (modern multi-cores), increasing the number of cores is done through the simplification of the whole processor.

An illustration of this simplification is the structuring of cores. Indeed, GPGPUs use the ancient parallelism principle of vector computation (vector processor, array processor). In array processors, units are grouped around a single instruction unit (Fetch and Dispatch or F&D), which decodes instructions that the group of cores will execute. The presence of one F&D unit implies that at most, one instruction is decoded at each cycle and is applied by all the cores together, typically on different data. This type of architecture is also known as SIMD architecture (Single Instruction Multiple Data) in Flynn's taxonomy.

This simplification cannot fully explain the space saved on GPGPU chips to add more cores, compared to standard processors, because the latter also implement vector registers, allowing SIMD computation (at a lower level). This type of CPU SIMD-parallelism is intended to be used by compilers through special instruction sets (SSE, Altivec,...).

Even if the paradigms seem different between the GPGPU and the SSE SIMD parallelisms, the principle remains the same. Indeed, in SIMD CPU functional units, registers are grouped into vectors and a single computing unit uses them, while for GPGPU, several computing units apply

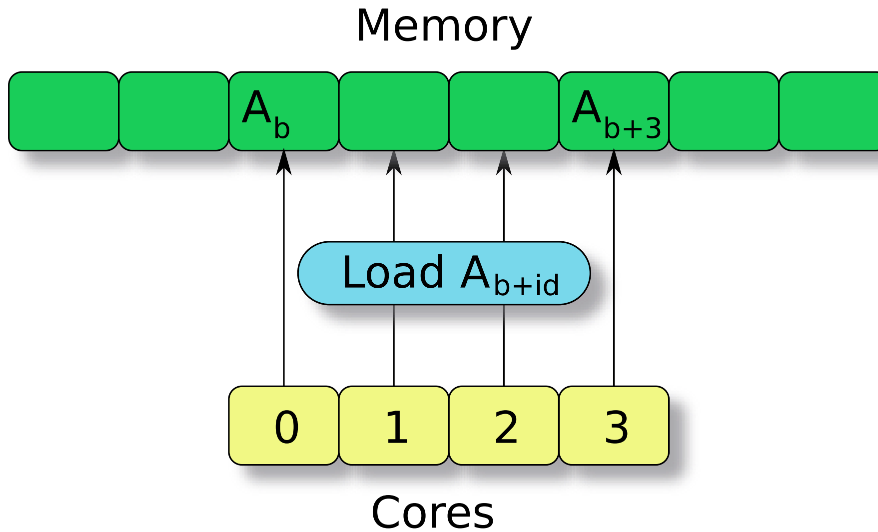


Figure 4.2: SIMD cores performing a memory load from a base address A_b and core id.

the same instruction on different private registers, containing possible vectorial data.

Another source of space-saving can be found at the memory level, as will be seen in section 4.3.2.

Multi-processors

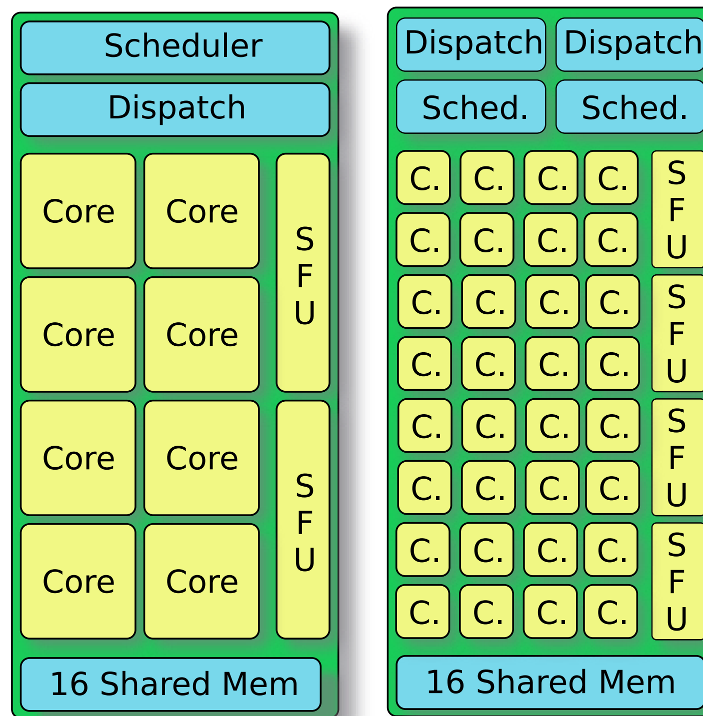
The sets of cores mentioned before are called multiprocessors (*MP*) and there can be several of them in a GPGPU card processor (currently 1 to 16). A schematic view of an *MP* is shown in Figure 4.3 for the two major NVIDIA GPGPU architectures.

All models of NVIDIA GPGPU cards atomically execute threads by groups of 32 and such a set is called a warp. On early models as presented in 4.3(a), a warp is executed every 4 cycles, by running 8 threads per cycle. The latest models, as shown in Figure 4.3(b), directly implement 32 real cores fed by one F&D unit. However, two F&D units each decode an instruction and the cores execute two different half-warps per cycle.

Due to the presence of these F&D units in each *MP*, GPGPU processors can execute different parts of the same code at the same time. This paradigm is sometimes denoted SPMD model (for Single Program Multiple Data) and imposes some limitations compared to a MIMD model. NVIDIA therefore implements a mixed model, because the cores work in SIMD-mode inside an *MP*, but the *MPs* work in SPMD-mode.

All of the *MPs* are connected to the same global main memory. No other communication mechanism between *MPs* is implemented. Finally, each *MP* contains a set of registers that will be introduced in Section 4.3.3 and a set of schedulers that will be presented in Section 4.4.3.

A last point to be noted here is the availability of SFUs (Special Function Units), which allow to compute fast approximations of heavy transcendental functions (EXP2, LOG2, SIN, COS, etc.). The accuracy of these fast implementations ranges from 22.5 to 24.0 bits, depending on the analysed function and the GPGPU.



(a) A G80 Multi-Processor.

(b) A gf100 Multi-Processor.

Figure 4.3: Schematic view of multi-processors of two major architecture designs.

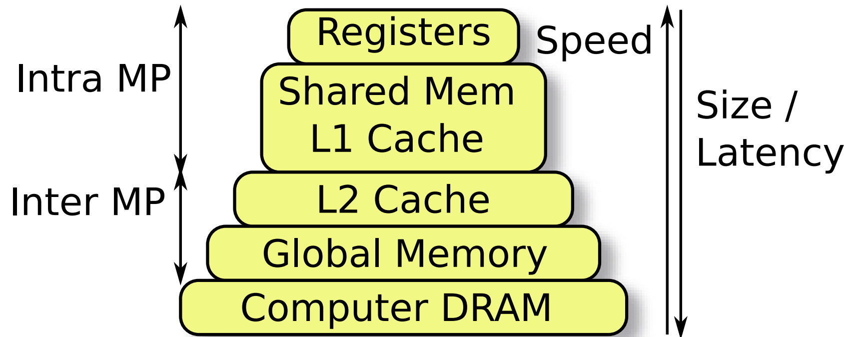


Figure 4.4: GPGPU memory hierarchy, organized by decreasing speed (bandwidth) and increasing size and latency.

4.3.2 Memory hierarchy

As in most computing systems, GPGPU cards embed several types of memories, organized according to access-speed and size.

L1-caches are not present on the early NVIDIA GPGPU cards, because they are not directly useful for 3D rendering. They are introduced later on, in order to be used for high performance computing.

Some of these memories are integrated directly in the chip (on-chip memory), then on the card and finally on the host system. Figure 4.4 shows the memory card hierarchy of modern architectures (*e.g.* a *GF100*). Different types of memories are shared, either between the cores of a same *MP* (Intra *MP*), or between all the cores (Inter *MPs*).

4.3.3 Register

Registers are on the top level of the memory hierarchy. They are obviously very fast, but what is more unusual is the large number of such registers (a total of 2MB on a GTX580). Time to access these registers is in the order of a cycle.

The registers are organized in a “Register File” and are assigned to a thread, rather than a core. It is therefore possible to organize them in different ways, for example by allocating more registers to a task, which reduces the number of tasks that can be loaded onto an *MP*. A register file is of course local to an *MP* and is shared between the threads running on it. More about the thread management process will be explained later on in Section 4.4.2. But, as the registers are allotted to a thread (a core), they cannot be used to communicate between them. The shared memory can be used to this end.

Taking into account the maximum number of tasks that can be executed on an *MP*, each task can use 32 registers of 32 bits. Compared to a standard processor, this number is relatively high, but it is important to note that storing the contents of a register in the main memory is costly, as discussed in section 4.3.5.

4.3.4 Shared memory

Shared memory is implemented into the chip (on-chip memory). Its size depends on the version of the chip and the latest (*gf1xx*) has up to 48KB of shared memory per *MP*. As for the registers, shared memory is attached to a given *MP*, but it can be used to share data between loaded threads.

Its organization includes separate banks and a fast interconnection switch which allows to create complex and fast communication patterns between cores and banks. A bank can issue a 32-bit word access and the switch supports fast accesses as long as two cores do not load two different words in the same bank. This allows to execute:

- a simple access, where each core loads into its own bank,
- a random access, where each core loads from a bank (with a 1 to 1 relation),
- a broadcast access, where all cores load the same data from the same bank.

Other kinds of accesses are possible, but they are not as efficient as the ones above. If several cores access different pieces of data from the same bank, these accesses are serially performed and in the worst case are achieved in 32 times the duration of a well-shaped access. Such an access is called a bank conflict.

The size of this on-chip memory has increased from 16KB for G8x-based cards to 64KB on more recent ones (*gf1xx*), but this memory is used to store an L1 cache and the shared memory. Two different configurations are possible: either 16KB to the L1 cache and 48KB to the shared memory, or 48KB to the L1 cache and 16KB to the shared memory.

4.3.5 Global memory

Global memory is a conventional large size DRAM (in the order of **few** GBs). All processing cores can access it. In addition, this memory can also be accessed by the processor of the host system that can perform data transfers in both directions. This mechanism is the only communication means between the card and the main processor.

Global memory transfer speed is very high ($\approx 200GB/s$) due to a large bus-width. This implies that any load is carried on a portion of the memory which is as wide as the bus itself. It is possible to maximize the use of the loaded data, if neighbouring threads (possibly in a warp) access neighbouring data, as described in Figure 4.5(a). As in the case of shared memory, the possible access patterns are relatively simple on the first GPGPU (G80) and more complex on the current generation. If this type of constraint is not met, the cores load unnecessary data, as in Figure 4.5(b), which overloads the cache and lowers the useful memory bandwidth.

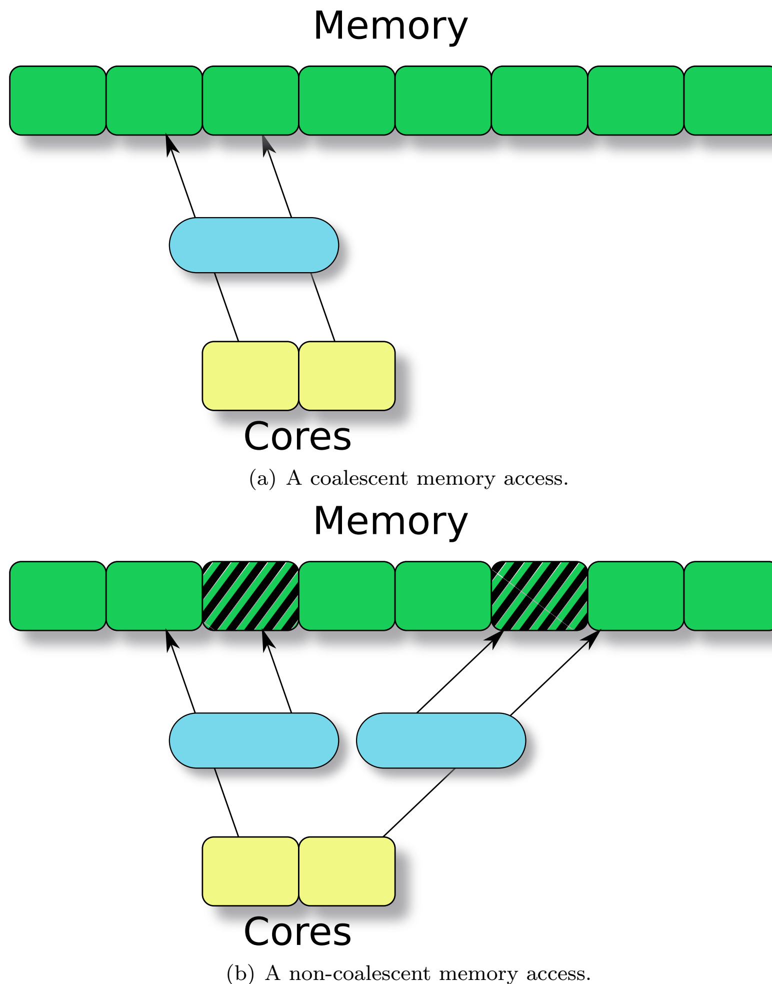
Such a memory access pattern is said to be coalescent, because the memory accesses it contains can be coalesced into fewer memory transactions than memory cell access. Applied to this case, coalesced memory accesses can be executed by less than 16 memory accesses (less than 1 per thread into an half-warp).

4.3.6 Considerations about memory spaces

The memory space of a program using GPGPUs is basically a set of memory spaces. The processor uses its memory (host DRAM), while each GPGPU processor has its own space and each *MP* owns its shared memory.

These three spaces are disjoint and it is necessary to execute explicit copies between them, in order to make available the data where it should be used.

The CPU has to handle addresses belonging to the GPGPU memory, because it is responsible for the exchanges to and from the GPGPU memory. Recent NVIDIA GPGPU software implements



(a) A coalescent memory access.

(b) A non-coalescent memory access.

Figure 4.5: Memory bus width and its importance in memory speed.

a unified virtual address space, by making automatic transfers between real spaces. However, if all the accesses are predictable, and clearly identified, the use of such an abstraction is not essential.

4.4 GPGPU Software

NVIDIA CUDA (Compute Unified Device Architecture) provides a unified view of different multi-core processors. Although only NVIDIA GPGPUs are taken into account, CUDA abstracts the real processor through different concepts. This abstraction ensures portability between different architectures (minor or major revision) produced by NVIDIA since 2007. Thus, the concepts included in this abstraction allow the programmer to ignore the number of cores and the *MP* structure.

CUDA allows GPGPU portability to the GPGPU programs, once compiled, with different types of cards. It allows to map the threads over the cores, according to their organization.

4.4.1 Compilation chain

CPU programs are usually compiled directly for the targeted architecture. While this approach was difficult to achieve and is the result of years of work to find a good consensus on the architecture (assembly language and operating system), binary compatibility is not always possible.

GPGPU architectures being recent and evolving, several of them coexist, even in the NVIDIA catalogue. The differences are sometimes minor (G80, G90) or major (G80 and GF100). To ensure the portability of a program, through these and the future architectures, compilation is performed in two steps.

First, the GPGPU program is compiled as an intermediate representation (*ptx* for Parallel Thread Execution), which is either embedded in the CPU program, or stored in an external file. Before executing a GPGPU program, the *ptx* object is loaded and compiled by the driver for the target architecture. This allows to not restrict a program to a GPGPU architecture and to use a diverse set of cards in one system.

However, these architectures have changed during time, for example with the introduction of new functionalities. Ascendant compatibility is maintained and made transparent through the 2nd compilation step, but is obviously not possible in the other direction.

It is still possible to force the compilation of the program into binary, doing both steps at compile-time. This mechanism can be helpful, when people are executing timing experiments, as the translation from *ptx* to binary could take significant time compared to simple computations.

4.4.2 Thread organization

As we have seen, due to the intrinsic SIMD structure of the cores, GPGPUs execute threads by bundles. These threads are organized spatially, first into a grid, then into several blocks.

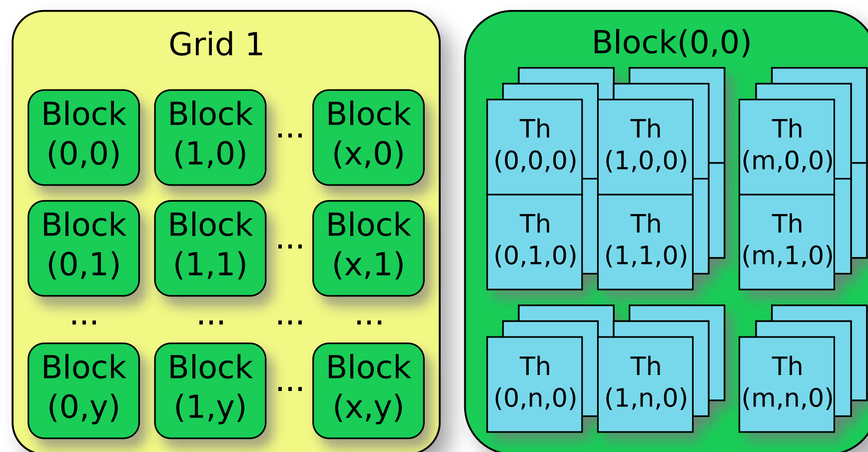
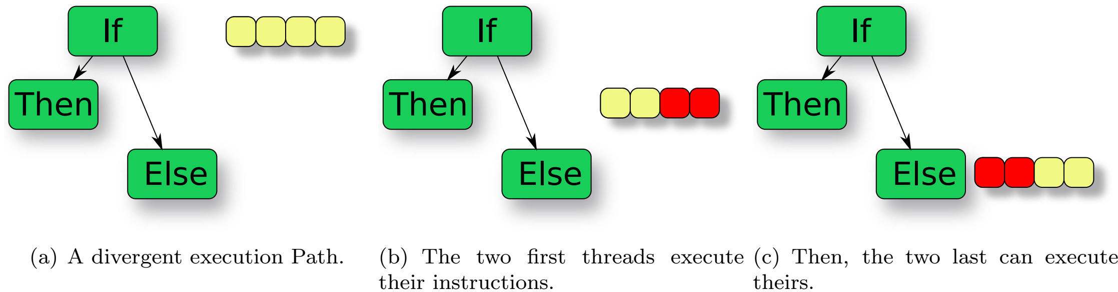


Figure 4.6: An example of threads software organization.

The blocks correspond to a 2D array, inside the grid, while threads are organized as a 3D array within a block. Figure 4.6 shows an example of a thread organization within a given grid.

Despite the abstraction proposed by CUDA, the logical organization matches the hardware organization. Indeed, a grid is run by a single GPGPU, a block is always executed by a single *MP* and of course, a thread by a core.

The warp, introduced in section 4.3.1, is one of the links between the hardware and software parts, even if it does not have a software counterpart. It is also one of the limits of the hardware abstraction exposed by CUDA. Indeed, the threads contained in a single warp run on a given *MP* at a given time (in a single or 4 cycles depending on the hardware version). Due to the SIMD nature of the cores, these threads cannot execute multiple instructions at a given time. Otherwise, they are executed sequentially, as in Figure 4.7. Thus, if the threads in a warp execute the code of Figure 4.7(a), cores 0 and 1 execute their instructions first, while the remaining 2 and 3 stay idle (4.7(b)). Then, inversely, cores 2 and 3 can execute their instruction, while the two first remain idle (4.7(c)).



(a) A divergent execution Path. (b) The two first threads execute their instructions. (c) Then, the two last can execute theirs.

Figure 4.7: SIMD threads executing different execution paths.

This behaviour is called a divergence between the threads. Of course, it causes a loss of performance, compared to the card peak computing power, as some cores remain inactive for a while.

But unlike a complete SIMD processor, only 32 threads (one warp) are executed together at a given time. A divergence therefore disrupts the execution of a code, only when it occurs between threads of a same warp. The divergence between different threads of different warps is therefore not a problem.

Thus, the warp notion can be important for performance. It allows to consider a minimum number of SIMD threads in the system and can allow to tackle problems where parallelism is not wide. Indeed, taking into account the warp size, it is possible to parallelize problems where there are *"only"* 32 SIMD parallel tasks. Up to now, the warp size remains constant through the different NVIDIA card generations.

4.4.3 Threads scheduling

The use of global memory is mandatory. Being the only interface between the two types of processors in the system (CPU/GPGPU), it must be accessed by the GPGPU at least once, to retrieve a value that can be stored in on-chip memories, explicitly in the shared memory or implicitly in the cache for future access. If a variable is accessed only once, the use of an on-chip memory brings no gain. Without data reuse, the cache only allows to accelerate spatially correlated data (whom a neighbour had already loaded).

The GPGPU implements an alternative mechanism capable of increasing the average rate of memory instruction execution. This mechanism is called "warp" scheduling and the idea behind it is to replace a thread waiting on a memory operation with a thread that is ready for execution. This principle has to be extended to the warp as a whole, because a thread is always executed together with the other threads of its warp. The scheduler therefore manages a set of warps and places it or not onto the computing unit.

This mechanism is different from the scheduling process found in operating systems, which stores the thread context into the main memory as soon as a context switch occurs (access to hard disk drive, waiting for a network message ...). Here, as the unit is already waiting for a memory operation, the warp has to remain within the registers in order to avoid other memory operations. This mechanism is therefore similar to SMT (Simultaneous MultiThreading), implemented under the name of HyperThreading™ on Intel® processors. However, Hyper-Threading only supports two competing threads, whereas on NVIDIA GPGPUs the scheduling set can contain up to 1024 threads (resp. 32 warps). As all the threads of a scheduling set stay in the "register file" of the MP all the time, the number of registers must be large enough to avoid storing registers into the global memory. For example, on modern architectures (such as *gf100* GPGPU processor), 32,768 registers are shared by the 1024 threads on each MP, allowing each thread to use up to 32 registers.

The set of threads to schedule comes from several blocks, as defined in Section 4.4.2. The block can be seen as a scheduling set, among which the scheduler can choose the next warp to execute in the case of a context switch.

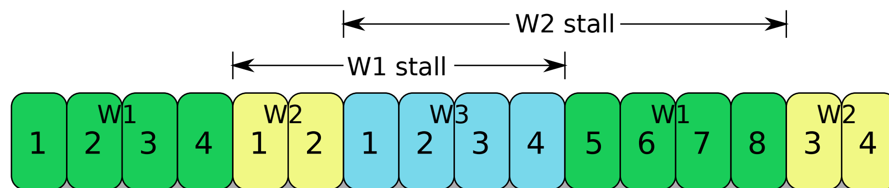


Figure 4.8: An example of scheduling.

Figure 4.8 presents a scheduling example for three warps running up to 8 instructions and causing three memory accesses. It is possible to see that the stalls induced by the memory latency are covered by the execution of other warps and that the warp causing this memory access was able to resume its execution once the scheduling process decided to.

Finally, scheduling appears to be a simple and automatic process, allowing to accelerate the average access time to the card global memory. Actually, this method is more generic than using the shared memory, provided that the number of warps present in the blocks is sufficient to overlap the memory latency. Warp scheduling being automatic, it is easier to use for the programmer.

Part II

Contributions

Chapter 5

Common principles

Contents

5.1 Analysis of EA parallelization 57
5.2 Load Balancing 60
5.2.1 Inter GPGPU load balancing 60
5.2.2 Intra GPGPU load balancing 60

5.1 Analysis of EA parallelization

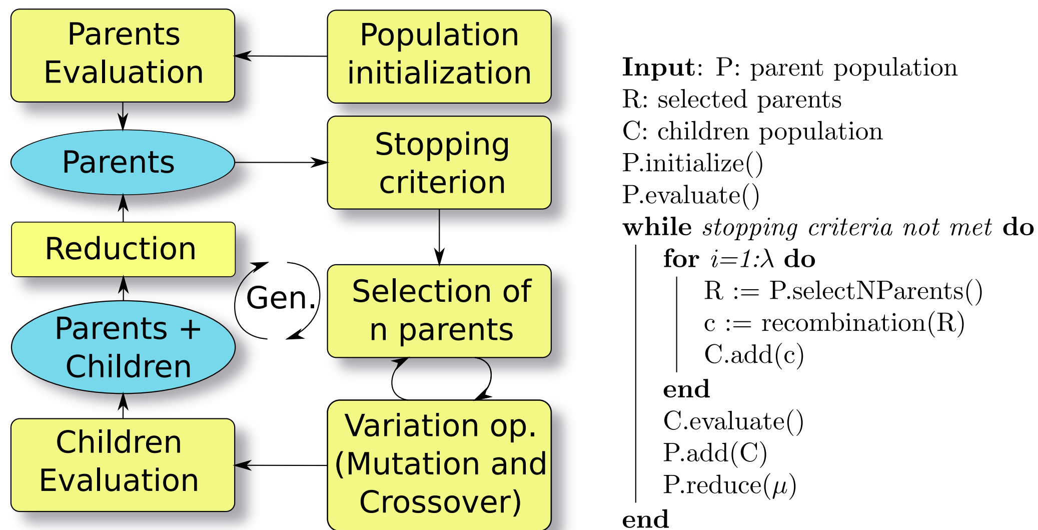


Figure 5.1: Recall from section 1.3.4: EA flow chart.

Input: P: parent population
R: selected parents
C: children population
P.initialize()
P.evaluate()
while *stopping criteria not met* **do**
  for i=1:λ **do**
    R := P.selectNParents()
    c := recombination(R)
    C.add(c)
  end
  C.evaluate()
  P.add(C)
  P.reduce(μ)
end

Algorithm 4: Recall from section 1.3.4: EA algorithm.

Evolutionary algorithms have an interesting characteristic, in that they are intrinsically parallel. Indeed, based on a standard algorithm, as the one described in Figure 5.1, it is possible to analyze the parallelization of all its steps.

Furthermore, it is possible to study the number of tasks that can be created from each step. This last point is particularly interesting, when one wants to use GPGPUs. Indeed, as we have seen, GPGPUs have many cores and even more virtual cores, and all these cores (virtual or real) must be kept busy during the whole algorithm for maximum efficiency.

In this section, μ will refer to the number of parents and λ will refer to the number of children, as in the ES terminology.

Initialization: It is a parallelisable step, in the general case. Indeed, the initialization of an individual usually does not depend on that of other individuals, as in evolutionary algorithms, genes of individuals are usually initialized randomly. Therefore, the initialisation of μ individuals can be performed by μ tasks, independently on n cores.

Evaluation: It is a parallelisable step, as in the general case, an individual is evaluated independently from the others. After the initialization phase, evaluation must be performed once per parent (μ times). In the evolutionary loop, evaluation must be performed once for every child (λ times).

In addition, for GP, the evaluation of a single individual can be parallelized, as the execution of the individual can be run in parallel on different training cases.

Stopping criterion: It is a fully parallelisable step:

- if the stopping criterion is a number of generations, each core can independently check whether this generation is reached,
- if the stopping criterion is to reach a specific fitness value, each core can independently check whether the individual just evaluated met this threshold or not. If it is, the core can raise a flag that will stop the evolutionary engine at the next generation, at which point each core will be polled and the best individual of all cores is returned.

Breeder Selection for child creation: It is necessary to select a number of parents for recombination. As it is possible (and even desirable) that the same individual be selected to create several children, n cores can independently select a parents in parallel for reproduction, if a is the arity of the recombination operator. The selection of parents is executed $\lambda \times a$ times independently, in order to produce λ children.

Recombination: After the parents are independently selected, each core can proceed to the creation of a child, by performing a recombination between the a selected parent's genes, that can be accessed simultaneously in a read-only manner. Recombination is therefore parallelisable.

Mutation: After a child has been created by a core through recombination, it can perform a mutation on it, independently from all other children mutations. Mutation is therefore parallelisable.

Population reduction: This step is *not* parallelisable in a straightforward way, unless a simplistic reduction method is applied (such as a generational replacement, where the new population is the children population).

If an ES replacement scheme is selected ($\mu + \lambda$-ES or μ, λ-ES), an efficient reduction of the population implies a selection of (μ) survivors without replacement. Indeed, if the same individual was selected several times to populate the new generation, this would generate clones that would reduce diversity and would cause premature convergence (when asked to choose a good individual, several cores could select the same individual, and this probability would increase with the goodness of the individual).

Making sure that all selected individuals are different would require cores to communicate, in order to make sure they did not select identical individuals. Selecting μ different individuals out of a $\mu + \lambda$ temporary population would imply numerous communications and synchronizations, which are both costly in a parallel environment.

In order to circumvent this problem, we propose in section 8.2 the DISPAR-Tournament operator, which works on disjoint sets of individuals.

As was just seen, all but one step of a generic evolutionary algorithm can be parallelized in a straightforward way on a shared-memory parallel architecture. The number of tasks is generally high, *i.e.* respectively μ, λ and $\lambda \times a$. Even if a sequential reduction phase is implemented at the end of each generation, it is possible to parallelize the contents of the evolutionary loop that are the most costly (mainly the evaluation step).

From a purely parallelization point of view, executing multiple iterations of the evolutionary loop is not possible simultaneously, because in synchronous evolutionary algorithms, the current population depends on the previous one, as the new generation uses the previous one as parents.

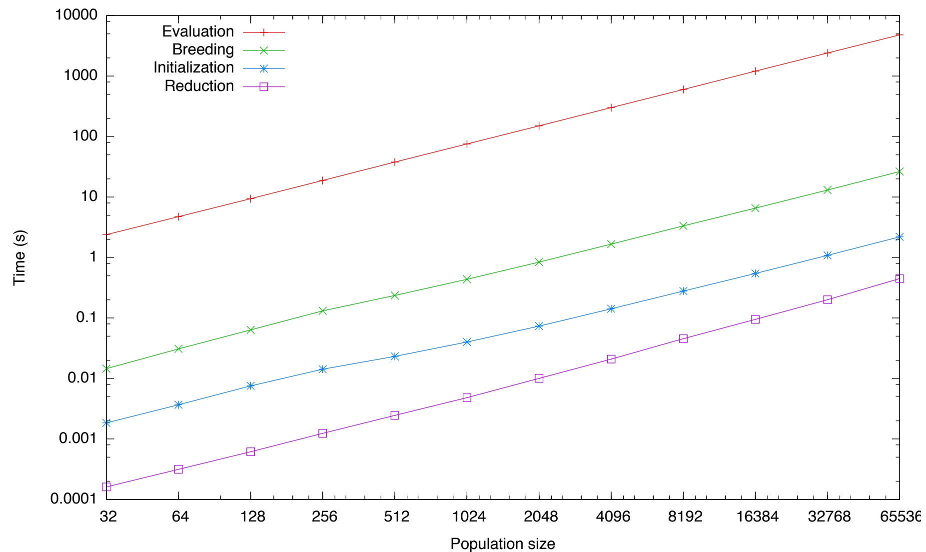


Figure 5.2: Amount of computing time for all the steps of an ES optimization of the Weierstrass function with 120 iterations on a CPU, depending on the population size. (Note the logarithmic scale on the Y axis.)

Executing several iterations of the evolutionary loop simultaneously would not make any sense. Generally speaking, parallelization is interesting in two cases:

- in order to improve the quality of the results for the same computing time,
- if optimization time is a hard constraint, as for real-time computing, for instance.

An EA is made of different steps, as described above. Genetic operators complexity depends on the genome size (initialization and variation operators), and evaluation of an individual depends on the problem. Finally, all the different steps depend on population sizes μ and λ. If the evaluation step is very calculation-intensive, parallelizing the evaluation step only may yield interesting

speedups. However, if evaluation is very short, then it may be necessary to parallelize all the steps of the evolutionary loop in order to obtain a good speedup.

Figure 5.2 shows the CPU time taken by all different steps of an ES optimizing the Weierstrass function on a CPU. On this problem, evaluation time is roughly 500 times more expensive than the rest of the algorithm. Suppressing evaluation time through infinite parallelization would approximately yield a 500× speedup, which is a good factor considering that the evaluation step only would have needed to be run in parallel.

5.2 Load Balancing

Having a system which embeds a host processor, a set of cards each with many cores on each card and with a set of tasks to perform makes it necessary to split the whole task into assigned subsets, to the cards, to the MPs and finally to the cores.

The main goal of a load balancing process is to reduce the overall execution time. On a synchronous EA, it is necessary to balance tasks between independent computing units, in order to allow them to finish their computation at the same time so as to start the next generation. Otherwise, if a unit finishes before the others, it will become idle while waiting for them, therefore wasting computing power.

Load balancing can be performed according to the tasks to be distributed at a certain level and to the devices related to it. However, distribution at a given level can take into account the structure of a lower level. Typically, MP distribution is also based on the structure of single cores.

GPGPU systems have three levels (cards, MP, single cores), but load balancing is done in two steps: first inter-GPGPUs and then intra-GPGPUs. Of course, the last two levels are highly correlated.

5.2.1 Inter GPGPU load balancing

If multiple GPGPU cards are to be used, it is necessary to assign a set of tasks to each GPGPU. This step matches the first level of the hierarchy described above.

The purpose of inter-GPGPUs load balancing is to assign processes to the cards in a balanced way, in order to allow the computing on each card to finish at the same time. Unfortunately, if different GPGPU types are used in the same machine, it is difficult to compare the power of the cards.

Indeed, it is necessary to take into account the characteristics related to the computing units, but also to the memory. An approximation was made, by dividing the number of tasks with respect to the number of MPs in the chips of the different cards.

Load balancing is much easier with identical cards and this was considered as the most common case for high performance computing.

5.2.2 Intra GPGPU load balancing

GPGPU chips comprise many cores, that are structured on several layers. Each layer has certain constraints, such as the number of tasks that can be scheduled, the number of registers available in an MP (its software-related notion being the block or the scheduling set).

The problem is: with a number of identical tasks and a number of cores (streaming processors, or SPs) grouped into a number of sets (MPs), how should these tasks be divided according to the following constraints ?

The proposed answer is to:

- Balance the load between the different MPs.

- Balance the load between the different cores of an MP, in order to not create artificial divergences, by assigning tasks to a part of a warp and nothing to the other part.
- Maximize the scheduling capability of each MP, while respecting the constraints on available registers per MP and the size of the scheduling pool.

The first point can be summarized as: the number of scheduling sets must be divisible by the number of MPs (otherwise, it is possible that an MP will become idle during the execution, while the others will still have work to do, especially towards the end of the computing phase).

Load balancing between the cores respects the warp size constraint. Indeed, the threads contained in a warp are always executed together. It is therefore necessary that the number of tasks distributed to a scheduling set be a multiple of the warp size (32). This is done even if it implies useless threads and it is up to the task itself to check its usefulness (and to do nothing in this case).

The last point is dependent on the card and on the tasks to perform on it. For an MP having a "register file" of a certain size (available registers) and a task using a number of registers, the maximum number of tasks that can be run on an MP is $n_{register}/n_{task_registers}$. This gives the maximum number of executable tasks which is e. The number of schedulable tasks is given by the hardware, as well as w and M (w being the size of a warp, M the number of MPs).

Finally, algorithm 5 is used to distribute the tasks into blocks and threads:

Input: N: number of tasks, w: WarpSize, M: Number of MPs, s: max number of schedulable tasks, e: max number of tasks.

Output: b: number of blocks, t: number of threads per block.

repeat

| $b := b + M$ $t := \lceil \text{Min}(s, e, N/(b \times M)))/w \rceil \times w$;

until $(b \times t > N) \wedge (e > t) \wedge (s > t)$;

Algorithm 5: Task Balancing on a GPGPU.

This algorithm will be used throughout the rest of this document.

Chapter 6

Parallelization of a GA/ES on a GPGPU card

Contents

6.1 Rationale **63**
6.2 Related works **63**
6.3 Proposed Master-Slave implementation **64**
6.3.1 Algorithm 64
6.3.2 Evaluation 66
6.4 Experiments **66**
6.5 Conclusion **70**

6.1 Rationale

Parallelization of evolutionary algorithms follows different models. One of them consists in parallelizing the evaluation step only (Master-Slave model). The advantage of this approach is that it keeps an *identical* behaviour compared to a sequential algorithm. Indeed, the evolutionary engine (the complete algorithm without the evaluation function) is the same as in the sequential algorithm.

Amdahl's law asserts that speedup is constrained by the sequential part of a parallel algorithm [67]. The achievable speedup using this principle is limited by the ratio T_c/T_e where T_c is the time of the complete algorithm and T_e the time needed to execute the evolutionary engine. Indeed, let us consider a parallelization over an infinite number of cores: the time of the population evaluation becomes virtually nil. Using this type of parallelism, the minimum execution time of the algorithm is the sequential part, *i.e.*, in a Master-Slave model, the evolutionary engine.

This kind of parallelization is useful for an algorithm whose evaluation time is predominant. This is the case in many problems.

6.2 Related works

There are relatively few studies addressing the use of GPGPUs to parallelize evolutionary algorithms.

To our knowledge, the first is [66], which implements a complete algorithm on a 3D rendering card. The GPGPU is used in a complete SIMD manner, which is then the only way to go, since

accessing the card for generic computation is done through fragment/pixel shader programs, which have a 3D rendering programming paradigm. Here, the authors use a fragment program, as it is adapted to their computations. The data required for the execution of the algorithm is stored in textures, such as the population, (stored as a 2D torus), an array of random numbers used by the algorithm during computation and a table of fitness values. In this implementation, four genes are stored into an RGBA pixel, allowing the application of genetic operators on four genes in parallel. Random numbers are generated by pre-executing a fragment program which executes an LCG algorithm (Linear Congruential Generator), in order to be used later. The implementation is similar to a cellular EA, as individuals are crossed according to their location in the population, with the best of their 4-neighbours. Finally, the authors evaluate their implementation on the Colville minimization problem by comparing the execution time between an Athlon 2500+ CPU and an NVIDIA 6800GT GPGPU card. The obtained speedups vary between 1.4$\times$ (resp. 0.3$\times$) for the genetic operators (resp. evaluation) for a population of 32^2 individuals and 20.1 $\times$ (resp. 17.1$\times$) for a torus of size 512^2. Using a 3D rendering paradigm imposes restrictions and issues regarding the implementation. In addition, this card model does not have unified computing units and as the authors use fragment programs, they can only exploit 16 cores on the card, leaving 6 inactive pixel shader units. All in all, this implementation seems to require a lot of work for a mitigated result, as a huge population is needed in order to obtain some reasonable speedups.

Then Fok et al. [68] published and proposed a similar technique, as they also port the complete algorithm onto the GPGPU. Fok *et al.* find that standard genetic algorithms are ill-suited to run on GPGPUs because of operators such as the crossover (that would slow down execution when executed on the GPGPU) and therefore choose to implement a crossover-less Evolutionary Programming algorithm [7] on the card. The obtained speedup of their parallel EP "ranges from 1.25 to 5.02 when the population size is large enough", on 5 different optimization problems, using a GeForce 6800 Ultra NVIDIA card, *v.s.* a 2.4GHz Pentium IV processor. As for the previous paper, using such an architecture imposes the use of the 16 fragment shader computing units, the 6 pixel shader units remaining idle.

Finally, in LI et al. [69], the authors implement a complete algorithm onto the GPGPU, in order to avoid the massive transfer of data between two memory spaces of the system. Strangely enough, the algorithm handles binary individuals, although the problems used for the experiments are continuous optimization problems (Schwefel, Shaffer and Camel functions). This implementation requires great efforts to implement operators as crossover and mutation without bit-wise operation capacities onto their GPU. They compare their algorithm with respect to a standard CPU algorithm, for convergence, optimal solution rate and speedup. The achieved speedups vary between 1.4 and 73.6, even if the two algorithms do not seem to be exactly comparable.

All these implementations were made on cards using non unified shaders. Also, programming was done using graphical paradigms. This explains the complexity of the chosen methods and the mixed results that were obtained.

6.3 Proposed Master-Slave (MS) implementation

6.3.1 Algorithm

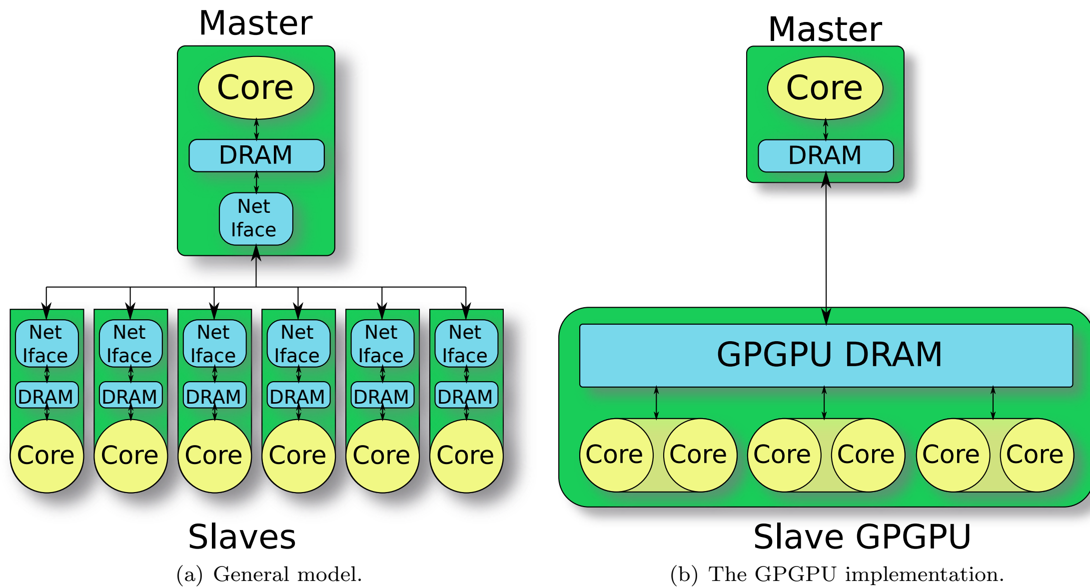
Parallelization can be achieved at several levels, one of them being the parallelization of the evaluation function using a master-slave model. The evolutionary loop is then executed on a master processor that performs the initialization of the population of μ parents, the analysis of the stopping criterion, the selection of parents for reproduction, the creation of λ children and the population reduction of the temporary population of $(\mu + \lambda)$ parents + children to the new population of μ parents.

The master processor also controls slave processors that are used for a parallel evaluation of the individuals. At each generation, the master processor distributes the population to the slaves, launches the evaluation, waiting for this evaluation to complete and receives back the fitness values just computed.

In a shared memory system, one of the processors performs the master role, the others being the slaves. In this case, no memory transfers are needed, as the master and the slaves share the same memory.

In a GPGPU system, the master and the slave processors operate as a distributed memory system: the master and the slaves are not using the same memory banks. The slaves do however use a shared memory architecture. Porting such a model on GPGPU imposes only a few changes compared to the standard algorithm, as the evolutionary loop is performed in the same way. It is quite possible to have, as in the standard model, several GPGPU cards sharing the evaluation of the same population. The master processor is then responsible for distributing this population among the cards.

Communication between the master and the slave processors is performed through direct transfers between the memories of those devices. This link can be seen as a communication network which is fast and has low latency (see Section 4.3.5) and a very large bandwidth (200GB/s), compared to using a physical network, as with parallel distributed memory architectures.



(a) General model.

(b) The GPGPU implementation.

Figure 6.1: Master-Slave model for EA parallelization.

Thus, the model comes from the one shown in Figure 6.1(a), becoming the one presented in 6.1(b). Going through a communication medium such as a network interface is no longer necessary, as the transfer is made directly between the different memory spaces.

However, because the transfer is done by DMA, it is more effective to add a data preparation phase that assembles all the individuals of the population in a single chunk of CPU memory, so as to transfer it to the GPGPU in one go, rather than to transfer one individual at a time. This way, the system suffers only one latency for the entire population.

If n cards are used, the population is divided into n subpopulations of individuals to be evaluated: one for each card. This makes for a quasi-linear speedup with the number of available cards. The

central processor launches the execution of the evaluation program on the cards and waits for the termination of all the involved cards. Finally, the fitness values are brought back to CPU memory and assigned to the individuals, as is the case in a standard master-slave parallelization model.

6.3.2 Evaluation

Evaluation is executed by the cores of the GPGPU card on the subpopulation that has been assigned by the host system processor. The population should be distributed to the different cores, to ensure load balancing. This is done using the Algorithm 5 presented in Section 5.2.2, considering an individual evaluation as a task.

Indeed, in a general case, in standard GA/ES, the evaluation time of an individual is considered as independent of the values contained in the individual's genome. So, having a set of x individuals (x being either μ for the initialization phase or λ for each iteration of the generational loop), each evaluation is considered as a part of an identical tasks set.

An individual is assigned to one processing core only and one core is preferably assigned several individuals, to be able to benefit from the very efficient scheduling system to circumvent the lack of cache memory, thus achieving temporal parallelization (pipe-lining). The core runs the evaluation code in a standard way by using its own internal variables, allocated in the global memory of the card.

As all the cores execute the same evaluation function on different data (different individuals) it is possible to use the cores in an SIMD manner with minimal divergence. But this SIMD behaviour concerns only the warps. It is therefore not possible (or not even desirable) to know if the warps are synchronized together. This global execution model can be viewed as an SIMD model, even if the hardware does not execute the same instruction on all the threads at the same time, as the number of threads exceeds the number of cores, in order to allow effective scheduling (SMT). Therefore, the considered SIMD model is more theoretical than practical.

The evaluation ends when all the individuals have been evaluated. The algorithm then returns to the host system processor. The algorithm has an implicit synchronization point to ensure that all the individuals are evaluated before resuming the execution of the evolutionary loop.

6.4 Experiments

This implementation was intended to be as generic as possible. However, using GPGPUs imposes a few constraints. For example, during execution, a GPGPU core must not perform any inputs / outputs other than memory accesses. Some constraints are recommended but not mandatory, such as allocating enough individuals per core for an efficient scheduling, as well as a fixed execution path in the evaluation function. Otherwise, performances are degraded, but the code still remains executable.

It is therefore possible to evaluate the implementation not only on well-known test functions (benchmarks), but also on real problems. This last kind of problems concerns the final user to a greater extent. The use of artificial problems allows to characterize the algorithm behaviour, while the implementation of real world problems demonstrates the applicability of these methods to industrial problems.

The metric used in this part is mainly the execution time of the algorithm on CPU and GPGPU, and a derived metric: the speedup. This is justified by the process of direct parallelization of the algorithm, which unlike the case of dEAs (Distributed Evolutionary Algorithms) does not imply algorithm behavioural modifications. As the evolution dynamics is not changed, only the gain in time needs to be analyzed.

Speedup is usually presented with respect to the number of processors involved in the computation. Here, it is difficult to apply such a technique. It is possible to reduce the number of cores or MPs involved on the evaluation of a given population but this does not make any sense. A GPGPU processor implements a number of cores and to use a subset of these cores primarily involves many disadvantages. Underusing an MP, that is to say use only a subset of cores, will not improve speed in a reasonable way, as all cores are available at no extra cost and will perform the same computation in SIMD, even if no threads are assigned.

However, it is possible to analyze the speedup by changing the number of cards used to evaluate the population, as a host PC will accept from 1 to 4 cards. It is also possible to analyze several models of cards with different numbers of cores, but other criteria could influence the results, such as the type of memory that is used and its associated bandwidth.

In addition, as CPU cores are symmetrical, it is possible to imagine that using all the processor cores could yield a linear speedup at best[1].

However, this is not the point in this document.

Throughout this document, it is a **parallel algorithm** *that is compared to a* **sequential algorithm**. *Therefore, whenever speedups are mentioned, they compare a* **parallel execution on all the cores of the GPGPU** *to a* **sequential execution on one core only of the CPU**.

Speedups are calculated with reference to the number of individuals evaluated on the card (μ on the first generation, λ on subsequent generations). They compare CPUs and GPGPU cards of similar eras. As in our model, the number of individuals is equal to the number of jobs created on the GPGPU and this method still allows an analysis of the behaviour of the algorithm on the hardware.

The Weierstrass benchmark

The first test conducted on the master-slave EA parallelization model is performed on a well-known problem, which is the optimization of a multi-dimensional Weierstrass fractal function, whose equation is:

$$W_{b,h}(x) = \sum_{j=1}^{\infty} b^{-jh} |sin(b^j x)| \quad \text{with } b > 1 \text{ and } 0 < h < 1$$

The multi-dimensional version used here sums the values of each dimension and samples the infinite sum with *iter* samples.

$$W_{b,h}(x) = \sum_{i=1}^{dim} \sum_{j=1}^{iter} b^{-jh} sin(b^j x_i) \quad \text{with } b > 1 \text{ and } 0 < h < 1$$

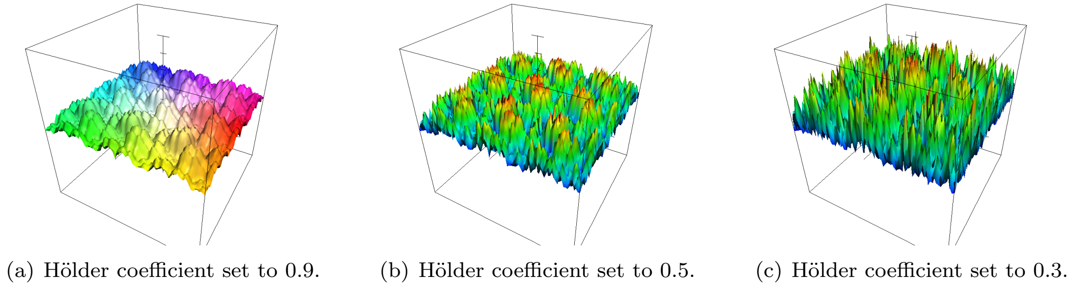
This function has many advantages in order to test the efficiency of an algorithm:

- The problem dimension can be changed (*dim*) therefore allowing to test the influence of the genome size on the behaviour of the algorithm (population management and transfer to the GPGPU card).
- The irregularity of the fitness landscape can be tuned thanks to the Hölder coefficient h, allowing to tune the difficulty of the search (used to evaluate the efficiency of the search).

[1] and possibly better than linear, as parallelization makes the caching system more efficient

- Finally, the computing load needed to evaluate an individual depends on a parameter (*iter*): the larger the *iter*, the longer it takes to evaluate an individual.

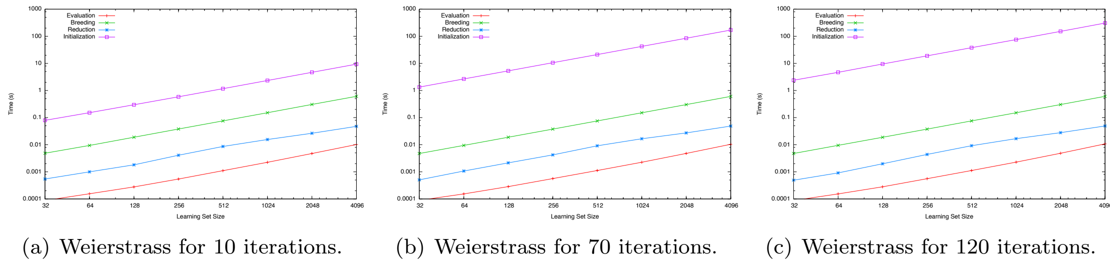
Figure 6.2 shows three fitness landscapes for a bidimensional evaluation function, for three different Hölder coefficients.



(a) Hölder coefficient set to 0.9. (b) Hölder coefficient set to 0.5. (c) Hölder coefficient set to 0.3.

Figure 6.2: The Weierstrass function using a 2D (2 real values) genome.

The ES optimization of this problem shows a predominance of the evaluation step compared to the rest of the algorithm, especially for a large number of iterations, as shown in Figure 6.3.



(a) Weierstrass for 10 iterations. (b) Weierstrass for 70 iterations. (c) Weierstrass for 120 iterations.

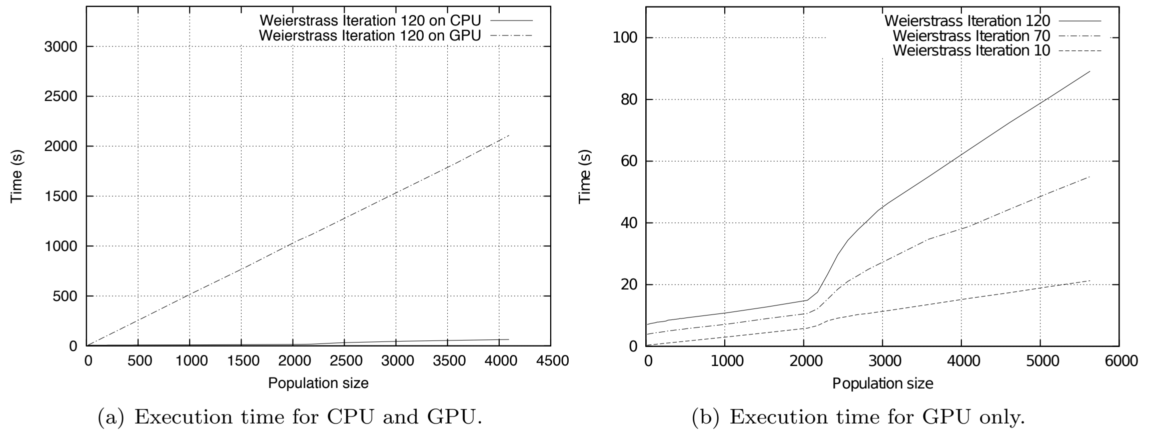
Figure 6.3: Execution time distribution of an ES algorithm onto CPU

Figure 6.4 presents the execution time for a standard algorithm, running on a CPU and an algorithm using a GPGPU as an evaluator slave. The machine used in these tests has an Intel Pentium IV 3.6GHz processor and the very first NVIDIA GPGPU card: the 8800GTX with 128 cores.

The complexity of the algorithm is linear *w.r.t.* the number of individuals in the population, for both versions of course. As shown in Figure 6.4(a), for a large number of iterations ($iter = 120$) the execution time of the sequential algorithm is considerably higher than for the parallel one (on the bottom of the graph) and it is so for any population size.

Figure 6.4(b) zooms on the y axis and shows the execution time of the parallel version of the algorithm for three different evaluation function complexities: 10 iterations, 70 iterations and 120 iterations. It is possible to see a change in the slope curve, at around 2048 individuals. Before this value, the evaluation time of the population is constant and only the time of the CPU part (sequential evolutionary engine) increases, which finally increases the total execution time of the algorithm.

This is due to two factors. A GPGPU processor is a chip containing p cores and the execution of a task takes a time t. As the population size n is smaller than p, each processor does not perform more than one task, taking a total time of t, so until the card is fully loaded (for 2048 individuals apparently), adding more individuals results in the same evaluation time.



(a) Execution time for CPU and GPU.

(b) Execution time for GPU only.

Figure 6.4: Execution time for the Weierstrass problem, for different numbers of iterations, *w.r.t.* the population size.

The knee occurs long after the n is over p (128 cores), because of the scheduling system. Indeed, the GPGPU processor can be seen as a processor with more virtual cores than real ones. A parallel can be drawn with operating systems, which consider the same amount of cores as the processors has, including SMT contexts. An I7 Intel IV processor is seen as an 8-core (4 SMT cores), a Pentium IV is seen as a dual-core processor (although it has only one). Thus, the processor of the 8800GTX can be seen as a processor with $128 \times 24 = 3072$ cores, 128 being the number of real cores, and 24 the number of scheduling slots by core. In Figure 6.4(b), the knee starts at about 2048 and the new slope stabilizes after around 3072 individuals.

Because the scheduling mechanism is a complex process, it is difficult to assert an exact number of threads to maximize it. Indeed, scheduling is only used to cover the memory access latencies. A code with just a few accesses saturates the computing cores with only few threads. The increase in the number of threads of the scheduling sets would no longer reduce the execution time, as computing power is the limiting factor and not the memory bandwidth. The size of an optimal scheduling set therefore depends on the hardware being used (number of cores, memory latency, scheduling capacity), but also on the code itself (ratio of memory access / computing).

However, it is possible to consider that the GPGPU computing task is not saturated below this value. Thus, all the above individual evaluations and up to this threshold, are virtually free. Indeed, they do not increase the population evaluation time.

It is interesting to note that for 10 iterations, the slope is roughly the same before and after the card is saturated. This means that on the GPGPU, 10 iterations in the evaluation function take practically no time. The slope is mainly due to the increase in population size that the evolutionary engine must deal with (more children to create using recombination and mutation and to choose from for reduction).

For 120 iterations, once the card is fully loaded (after 3072 individuals) more individuals mean more evaluations, that take much more time than population management by the evolutionary engine. The slope is therefore very different.

Figure 6.5 shows the speedup of the proposed master-slave implementation on the same problem using different types of NVIDIA cards. The load balancing algorithm presented in section 5.2.2 ensures that the hardware configuration of the different cards is dealt with correctly and transparently. All the speedups stagnate beyond a certain population size. This is the speedup counterpart for the change of the slope curve in Figure 6.4(b). Indeed, once the card and its scheduling system

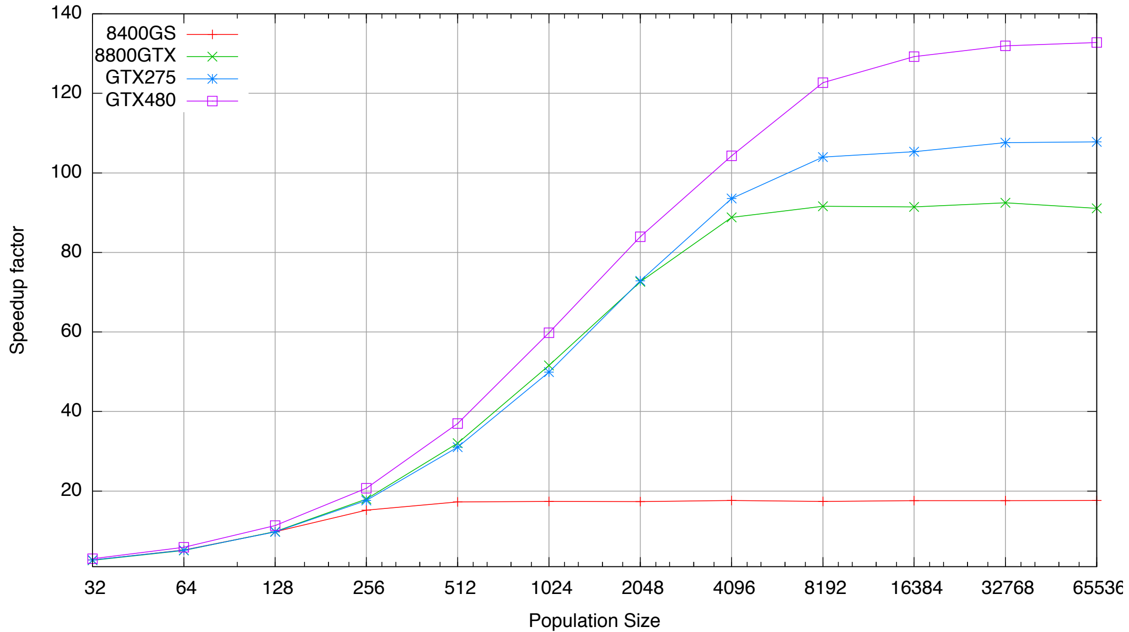


Figure 6.5: Speedup *w.r.t.* the population size, on different NVIDIA GPGPU cards.

is saturated, the execution time of the algorithm on the card becomes linear with respect to the population size.

Depending on the number of cores and the card type, these threshold values are different. Indeed, the G80-based cards (8400GS, 8800GTX) and G200 (275GTX) have a scheduling set size of 24 threads per core to be compared with 32 for the GF100-based card (480GTX), the number of cores also vary. In the case of the 275GTX with 240 cores *vs* 128 for the 8800GTX, we can observe that the speedup stagnates for the first card with a population size of twice the second. Finally, the 480GTX saturates later, because both the number of cores (480 cores) and the size of the scheduling sets are larger.

Note that all these speedup curves include *the complete sequential evolution engine*. The speedup factor on evaluation only would be much greater.

6.5 Conclusion

Even though the Master-Slave parallelization model that only parallelizes the evaluation of the individuals on the cores of one or several GPGPU cards using their shared global memory seems straightforward, it was apparently not described in any paper before our 2009 paper (Maitre et al. [70]).

The work referred to in this section was published in Maitre et al. [71] (international refereed journal paper) and Maitre et al. [70, 72] (international refereed conference paper).

Chapter 7

GPGPGPU (or Genetic Programming on GPGPU cards)

Contents

7.1 Introduction **71**
7.1.1 Option 1: SIMD parallelization of a single individual evaluation over the learning set 72
7.1.2 Option2: MIMD parallelization over the whole population 73
7.2 Related work **75**
7.3 Implementation **76**
7.3.1 Algorithm 76
7.3.2 Evaluation 77
7.4 Experiments **77**
7.4.1 Experimental process 77
7.4.2 Evaluation function 78
7.4.3 Experiments on the complete algorithm 82
7.4.4 Example on a real world problem 84

7.1 Introduction

Genetic Programming (GP) is generally known to be very computing intensive, because the search space is very large, as GP optimizes the values as well as the structure of the individuals.

Figure 7.1 shows computing time based with respect to the number of training cases, for different phases of the algorithm for a run of 10 generations and a population of 4096 individuals. The time spent in the evaluation function becomes as large as the rest of the algorithm above a relatively small number of training cases (between 64 and 128).

More generally, evaluation in GP consists in comparing the results produced by an individual with a set of learning points. Unlike GA / ES, the individual is executed in order to evaluate it and all individuals are different. During the evaluation phase, an individual is evaluated on a learning set that may contain many points and for each point, the individual must be executed.

This is a problem if one wants to parallelize evaluation on an SIMD parallel machine, as all threads will be divergent. However, GP evaluation reveals another source of SIMD parallelism, in

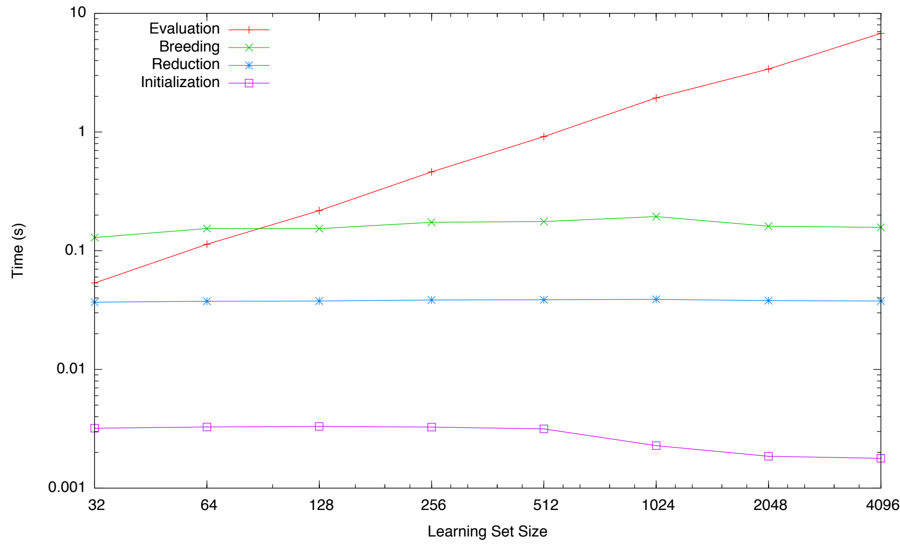


Figure 7.1: Time elapsed into different steps of a GP algorithm, *w.r.t.* the learning set size.

the evaluation of a single individual. Indeed, an evaluation implies to interpret / execute the same individual on different data (the learning set). The evaluation function (that can be seen as an interpreter) could run the same individual in parallel on a set of input values, in which case threads would not be divergent anymore as they would implement the same individual.

But is is also possible to use the same kind of parallelization as with GA / ES, that is: each individual can be evaluated by one core independently.

Based on these two approaches, there are two ways to parallelize the evaluation of a GP population.

7.1.1 Option 1: SIMD parallelization of a single individual evaluation over the learning set

First, the card can be seen as a completely SIMD machine. An individual is loaded on it and each core executes (or interprets) the same individual on a subset of training cases.

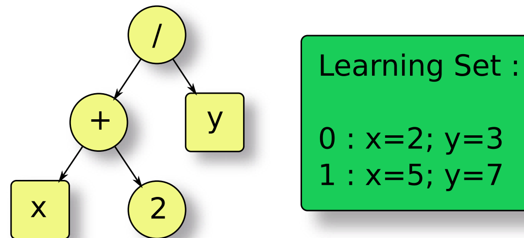
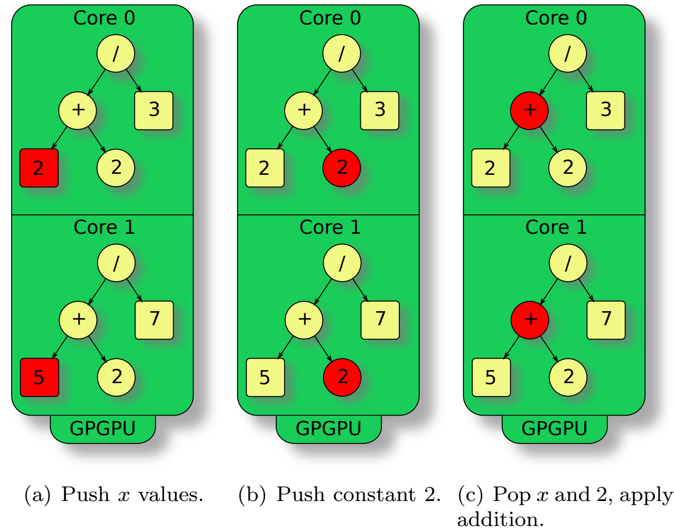


Figure 7.2: An example of a GP-tree individual and a problem-related learning set.

Figure 7.2 presents an example of a GP tree and a training set containing two couples of input

values for x and y, on which the tree must be evaluated. Using the technique described above, the evaluation of some of the nodes gives the sequence shown in Figure 7.3. Two cores execute the same instruction at the same time, possibly with different data, when the operator of a node is a training set variable (here x or y). The potential number of SIMD parallel tasks is then the number of elements in the training set (L_{ls}).



(a) Push x values. (b) Push constant 2. (c) Pop x and 2, apply addition.

Figure 7.3: Execution of the first learning set cases in SIMD-mode.

Unfortunately, in many real-world problems experimental data can be difficult (or expensive) to obtain, and GPGPU cards contain quite many cores (and even more if the scheduler is used to implement temporal parallelism, by overcoming the latency of global memory accesses). One has seen in the previous section that an old 128 cores 8800GTX needed 3,072 threads in order to be fully loaded. Current cards have 512 cores, which could mean that these cards would need more than 12,000 test cases to be used efficiently. However, in the manganese leaching inverse problem presented in chapter 10.1, much less than 100 training points were available.

Furthermore, this parallelization over the training set does not take into account the MIMD structure of GPGPU MPs.

7.1.2 Option2: MIMD parallelization over the whole population

On the other hand, it is possible to have an opposite approach. Each processor core can execute a different tree (individual), applying it to each and every learning case. This approach is suitable for an MIMD processor. Indeed, in this case, each processor executes a population subset. The number of parallel tasks is now N, where N is the size of the population to evaluate.

However, figure 7.5 shows how the two individuals of figure 7.4 would be executed on an SIMD processor:

1. The evaluation of the two individuals' first node causes the execution of the same instruction (load a variable). However, one loads the x variable, while the other loads the y one, meaning that memory access is not coalescent.

2. The second node implies the execution of the same instruction on both cores (read "2"). The memory access is coalescent.

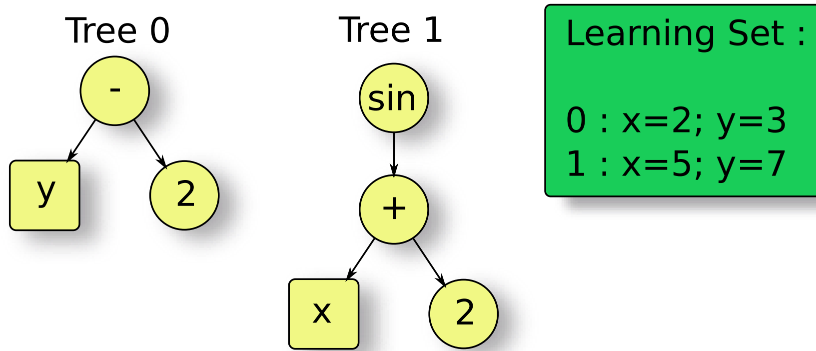
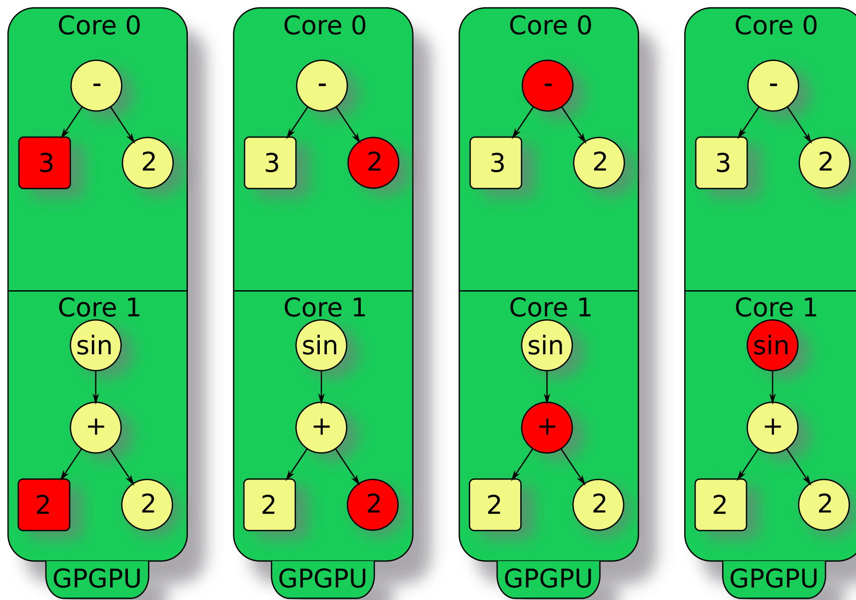


Figure 7.4: An example of a GP-tree population and a problem-related learning set.



(a) Single instruction, non-coalescent memory access. (b) Single instruction, coalescent memory access. (c) Divergence, two instructions. (d) One idling core.

Figure 7.5: Execution of the first three nodes of the example tree in an SIMD manner.

3. For the third node, the two cores should execute different instructions (+,−). With such an SIMD processor, these instructions are serialized.

4. The first tree is finished, so core 0 is idle until the end of the evaluation of the second tree by core 1.

MIMD parallelization on an SIMD machine shows some disadvantages, due to divergences between cores, the difficulty of predicting the memory access pattern and to efficiently use the bus width, and to balance the load between the different cores. However, if one takes into account that only a limited number of different operators will be used, it is possible to obtain some speedup: supposing that the set of operators is limited to $\{+, -, \times, /\}$ and there are 512 SIMD cores on the chip. If, during evaluation, all four operators must be executed, this will result in 4 time steps rather than one. But because 512 cores will be used, one could hope to obtain a 512/4=128× speedup.

In practice, this is unfortunately not true, because there are more different operations to perform than only 4 (there are usually more than 4 operators, and loading a variable on one core while another core wants to perform an operator is also a cause of divergence).

7.2 Related work

Genetic Programming, its huge search space and the time needed to evaluate a population, make it a good candidate algorithm to be executed on parallel architectures, so several studies have been done on parallelizing GP on GPGPUs.

A first test was attempted using a compiled-approach [73] and using option 1 presented above (parallelization of each individual over the learning set). The learning set is converted into a 2D texture, and is sent to the card. Then, each individual is written as Cg code (C for Graphics, a 3D rendering oriented language), compiled and sent to the card. Finally, the result is compared to the expected one in the learning set and the error is assigned to the fitness of the CPU individual. Despite the large overhead involved by the use of the Cg language, Chitty gets a speedup of 30× on an NVIDIA 6400GO when pitted against an Intel 1.7GHz, on the 11-way multiplexer with 2048 training cases, and a speedup of about 10× on a symbolic regression problem. The author says that the size of the training set must be large enough. Indeed, the overhead induced by using Cg and the compilation of the individuals is difficult to overcome. Apart from the overhead, evaluation of a compiled individual is fast. This idea is reinforced by an almost constant running time until it reaches a threshold number of training cases, that the author estimates to be between 10,000 and 15,000. It is not clear in the paper whether the compiling overhead is taken into account when calculating the execution time of an individual.

In the same year, Harding *et al.* apply a similar technique in [74], using *.Net* and *MS Accelerator*. For four different problems, the authors generate a random population of individuals of a fixed size and evaluate them on a GPU and a CPU. Then, they test the average speedup of the GPU implementation *vs.* the CPU on 100 different individuals, using a 7300GO card *v.s.* an Intel T2400 1.83GHz CPU. Using this mechanism, they investigate the obtained speedup *w.r.t.* different individual and training set sizes. They get very interesting speedups, in the order of 7000×, on the evaluation of a floating expression population, 800× for boolean expressions. For the complete algorithm, they obtain a maximum speedup of about 100× on a linear regression problem (which uses floating expressions) and 22× for a problem of spiral classification (which uses boolean expressions). However, the use of the *.Net* and *MS Accelerator* frameworks, the individual optimization during the compilation and the lazy evaluation, make it difficult to compare these results. Nevertheless, the use of *MS Accelerator*, which works on top of DirectX, allows the program to be ported on different types of devices (GPGPU, multi-core processor), if not portable to other operating

systems. In addition, this implementation uses the GPGPU as a complete SIMD processor, which implies using a large number of training cases, as GPGPU processors contain hundreds of cores (512 on the latest processors when this document is written).

Langdon *et al.* use option 2 in [75] (evaluating each individual on a different core). Their RapidMind implementation uses an interpreter that evaluates all the trees on all the cores in parallel. On one core, a tree is sequentially evaluated on all training cases. They obtain a speedup of about 12× using a 8800GTX card *vs* an AMD Athlon 64 3500+ clocked at 2.2GHz. Their RapidMind implementation evaluates all possible operators on a node, by keeping at the end the value corresponding to the good operator. For a set of 5 operators they evaluate the inefficiency of their implementation to 3×, due to using the SIMD architecture as if it was an MIMD.

Finally, in 2008 and 2009, Robilliard *et al.* propose an implementation based on a mixed approach between options 1 and 2 in [76, 77]. This approach allows to take into account the SIMD/MIMD character of the G80 NVIDIA architecture. In this implementation, the 32 SIMD cores of an MP are assigned to an individual. They are used to compute in parallel, as in option 1. Then, different MPs get different individuals to evaluate, as in option 2. The authors get an acceleration of 80× on an 11-multiplexer problem and a sextic regression, with 2,500 training cases. This speedup is obtained on the evaluation function only, with a 8800GTX card, against an Intel 2.6GHz processor. As this approach is the basis for ours, we will see it in more detail later.

7.3 Implementation

7.3.1 Algorithm

One of the big problems with option 1 is that it needs many training cases in order to efficiently exploit the many-core architecture of the GPGPU chips. Unfortunately, when test cases come from real world problems, it is often the case that fewer than 100 learning cases are available. On a work with Biswas *et al* [78], cf. section 10.1, only 77 test cases were available to learn from, in order to optimize the leaching of manganese ore extracted from polymetallic sea nodules. It is therefore very important that GP parallelization methods be efficient with the smallest possible numbers of fitness cases, if one wants the method to be applicable to real-world problems.

The implementations by Robilliard *et al* [76] was efficient, GPU-wise, with as few as 128 learning cases. We extended their work so that GP parallelization over GPGPUs could be efficient with as few as 32 learning cases, which is important for real-world applications with few learning cases (if less than 32 learning cases are available, one can reasonably question whether the data set is large enough to learn anything interesting from it).

As the implementation presented in Robilliard et al. [76], our implementation uses tree-shaped individuals for the evolutionary engine. These trees are implemented using objects for nodes connected to their sons through pointers.

This individual representation is not interesting for the evaluation. Robilliard *et al* use a RPN (Reverse Polish Notation) format for the evaluation and apply a translation routine to flatten the trees before exporting the population to the card. The translation is obvious and is done by recursively travelling the trees. The RPN notation is more compact and allows to transfer the population in a single block, as in the case of GA/ES.

In Robilliard et al. [77], the authors directly use the RPN representation at the evolutionary loop level. This solution of course accelerates the transfer, because it allows to avoid the translation phase, but more complicated variation operators (crossover and mutation) are needed to take into account the implicit structure of the individuals. This solution was not selected here, for simplicity and genericity reasons.

If the authors of [76, 77] use the ECJ library and an interface to the C to access the graphic

card, we chose to directly use C++ at the evolutionary loop level for easy integration into the EASEA platform (*cf.* chapter 9).

7.3.2 Evaluation

In [76, 77] the authors use only 32 threads for the evaluation of an individual. This corresponds to the size of a warp, and therefore to the least number of threads executed by an MP in one cycle. Thus, the evaluation of an individual is done by one MP on 32 training cases in parallel.

The NVIDIA hardware can schedule up to 4 blocks on a single MP, at least if the computing resources allow it (number of registers, the shared memory occupancy). In the best conditions, an MP will schedule between 128 threads, coming from four different individuals. The scheduling set is not full, because it may contain up to 768 threads on the card being used. This implies that in Robilliard's implementation, the scheduling capacity of the card is not completely used and that more memory latencies could be avoided by using more threads.

NVIDIA documentation states that a divergence occurs *between the threads from a same warp only.* As an MP can load four blocks and schedule between them, the divergences between threads from these four different individuals do not affect their execution. This behaviour makes sense because these threads are executed at virtually the same time. However, as there is no interaction between the different threads of two warps, then two different warps can execute different instructions without degrading the MP SIMD behaviour. Evaluating multiple individuals on a single MP allows to maximize spatio-temporal parallelization on the mixed SIMD/MIMD architecture of GPGPU processors.

In [76, 77], the authors implement a GPGPU RPN interpreter, which evaluates an individual on a subset of the training cases in parallel, then on the following subset sequentially.

Two solutions exist to increase the number of threads loaded onto an MP. Using a larger training case subset in parallel is possible. This number can be increased up to 192, which allows to maximize the size of the set, if the card uses 4 interpreters.

An interpreter managing several individuals over several training cases in parallel is also possible, if no individuals are shared between two warps. In this case, even in a symbolic regression on 32 points, all scheduling sets will be fully loaded, as long as six individuals are assigned to each MP.

However, in [76, 77], the authors use shared memory to store the interpreter stacks. There is a stack per learning case and per individual (actually, a stack per thread). By increasing the number of threads per MP, shared memory can no longer store enough large stacks. To avoid this problem in our implementation, the stacks are stored in the global memory, without observed side-effects.

Another notable effect of increasing the number of threads in a scheduling set is the increased load on instruction cache pressure, but as with the previous remark, no adverse side-effects were observed.

7.4 Experiments

7.4.1 Experimental process

The experiments have been performed on a GTX295 NVidia card *vs* a Intel Quad core Q8200 (2.33GHz) processor and 4GB RAM of the same vintage, under GNU/Linux 2.6.27 64 bits with NVidia driver 190.18 and CUDA 2.3.

The GTX295 is a dual GPU card, but to simplify things, we only used one GPU to evaluate GP individuals, the second card being used for the standard display.

A multi-GPU implementation has been tested by distributing the population on the two GPUs of the GTX295 card. No drawback has been observed other than the need for the population size

to be large enough to fully load the two GPUs, meaning that the speedup in terms of number of GPU cards is linear.

The CPU code has been compiled with gcc 4.3.2 using the -O2 optimization option and run on one core of the CPU only, so as to a compare the parallel GPGPU algorithm to a sequential one on CPU. By not implementing a multi-core CPU GP, one still has the possibility to imagine that in the best case scenario, a perfect multi-core CPU implementation would show a linear speedup, with respect to the number of cores of the CPU.

7.4.2 Evaluation function

The evaluation function is applied to a symbolic regression problem. The population is randomly generated, by using the *grow*, *full*, or ramped half and half initialization method described by Koza in [10]. The evolutionary loop is not implemented for these tests, in order to avoid a bias in the size or the shape of the trees selected by the evolution. Furthermore, the same trees are used for both the CPU and the GPGPU algorithm.

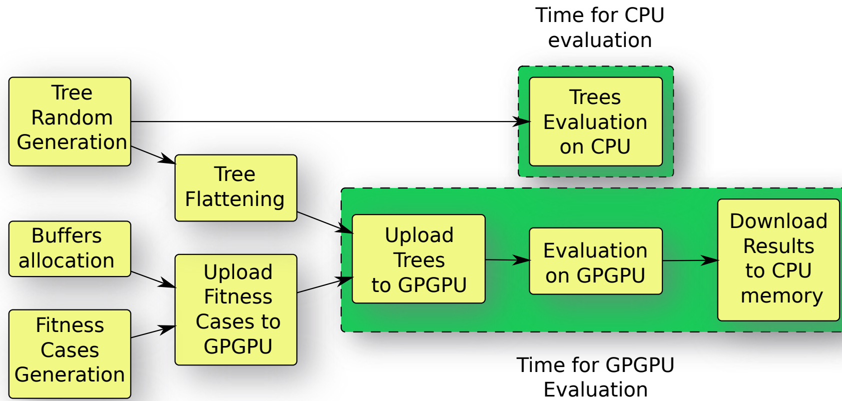


Figure 7.6: Global scheme for GP simulation and timing process.

Figure 7.6 presents the methodology used in these tests. Only the evaluation phase is timed on the CPU side, while for the GPGPU this timing includes the transfer of the trees to the card, the population evaluation and the transfer of the fitness values back to the CPU memory. The algorithm includes an initial step, which involves the allocation of different buffers and the transfer of training cases to the card memory. In a real algorithm, this step is performed only once at the beginning of the algorithm. This is why it has not been included in the timing.

Evaluation time *vs* number of fitness cases

Figure 7.7 presents the evaluation time obtained with four different implementations, using a population of 60,000 individuals of depth 7 initialised with Koza's *full* method [10]. The *BlockGP32* implementation shows the evaluation on 32 training cases in parallel. *BlockGP128* uses a larger number of training cases, evaluating an individual on 128 training cases in parallel. Then, *BMGP* makes the evaluation of several individuals (respectively 2 and 4) in a single interpreter, on 32 learning cases in parallel.

The single-individual implementation (*BlockGP*) implicitly evaluates four individuals per MP. This brings the number of threads to 128 (respectively 512) for *BlockGP32* (respectively *BlockGP128*).

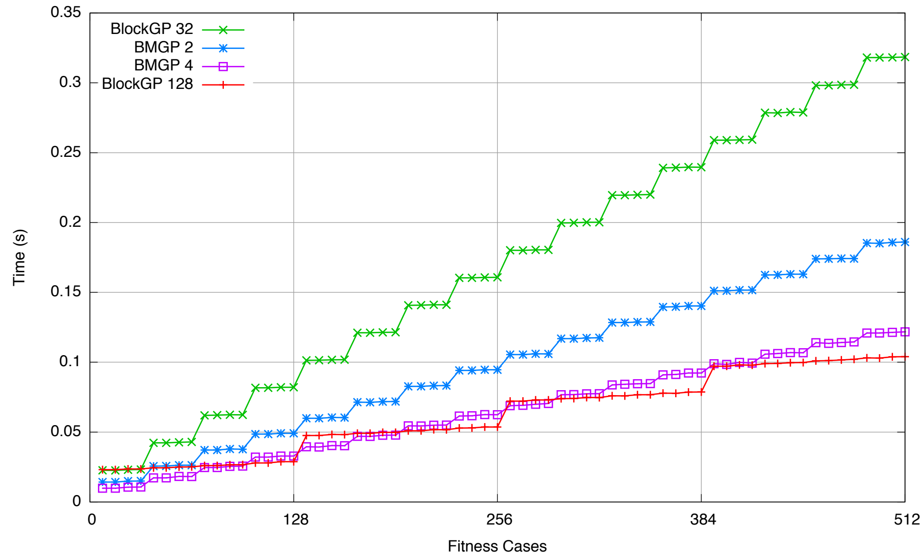


Figure 7.7: Evaluation time for depth 7 trees for different implementations with a sampling size of 8 fitness cases.

Multi-individual implementations also implicitly use 4 individuals, which brings the number of threads to 256 (respectively 512) threads for *BMGP2* (respectively *BMGP4*).

The curves are shaped like stairs, whose sizes are equal to the number of training cases computed in parallel. The *BlockGP128* implementation (corresponding to our modified version based on the one described in [76, 77], a re-implementation of this algorithm being BlockGP32) is very effective, but with steps that are 128 fitness cases wide. Our improvement (*BMGP4*) allows a finer granularity with similar performance, *i.e.* with steps that are 32 fitness cases wide.

One can note that the amount of threads loaded on an MP has a real impact on the population evaluation performance. Memory behaviour is improved, even if shared memory is no longer used to store the interpreter stack.

Influence of tree depth on speedup

Using the same timing technique, Figure 7.8 shows a speedup trend with respect to the tree size and to the number of training cases, for the *BMGP4* algorithm. The learning function set used here is $\{+, -, \oplus, \times, \cos, \sin\}$ (where $\oplus$ is the protected division and the population size is 4096). Trees are created using the *full* method, although when using this learning function set the trees are not guaranteed to be full. Indeed, the use of unary functions does not allow to predict the number of nodes of a tree, only the depth of all leaves.

The curve shows that the size of trees has a marginal influence on the obtained speedup, but the size of the training set seems to be more important in this test. Indeed, the surface reaches a plateau above 32 training cases. Above this value, the evaluation function speedup increases only a little, showing that the speedup of this implementation can be maximized with several learning cases.

The plateau represents a speedup close to 250×, once scheduling sets are correctly loaded.

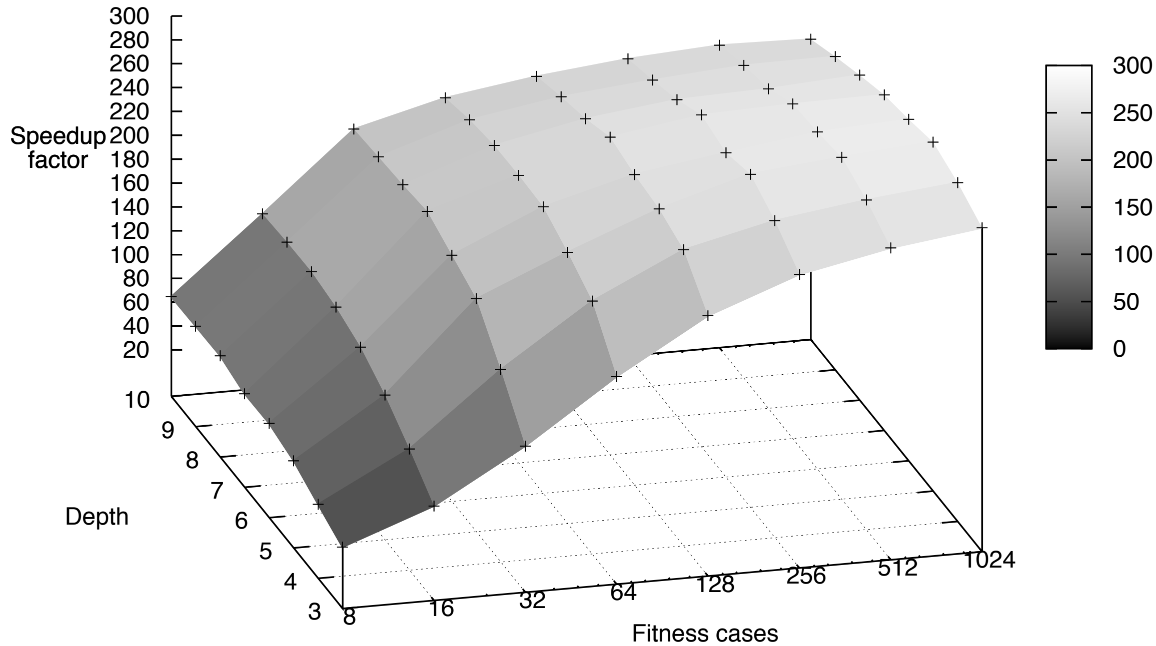


Figure 7.8: Resulting speedup between GPGPU and CPU.

Influence of the function set

Learning function sets may have different computational intensities. Table 7.1 shows two learning function sets where one (*FS1*) has a low intensity and the other contains heavy functions such as cos, sin, log and exp (*FS2*). Figure 7.9 presents the speedup obtained with these two learning function sets, and the use of Special Function Units (SFU) presented in Section 4.3.1. These experiments use a population of 4096 7-deep individuals, initialized using the *full* method.

Function set 1 has a relatively low computational intensity, using only fast-to-compute operators. The number of threads loaded per MP does not seem to influence the obtained speedup, as *BMGP2* and *BMGP4* have substantially the same values. The better memory performance accelerates only slightly this type of calculation.

Function Set 2 having a higher computational intensity, the speedup reaches the one presented in the previous section. The curves show teeth shapes, which coincide with a line divided by the staircase shape of the previous section. Indeed, the evaluation time on the processor is linear with the number of training cases.

Teeth repeat every 32 cases; this is where the cores are all busy and warps do not contain any useless threads. Also, this is the first point of the stairs in the previous figure. Depending on the number of individuals per MP, the speedups range between 180 and 230×.

The last couple of curves (FS2 SFU) use the fast approximation operators, for functions like sin, cos, log, exp. The acceleration is greater than the one obtained using the standard implementation of these functions, but at the cost of lower accuracy. This behaviour can be a problem if the result of a run is intended to be executed on computing units that have no comparable units. However, if the obtained function is used on the GPGPU, the symbolic regression will work as expected, as it will use the very same approximation as the one that was used to evolve the function.

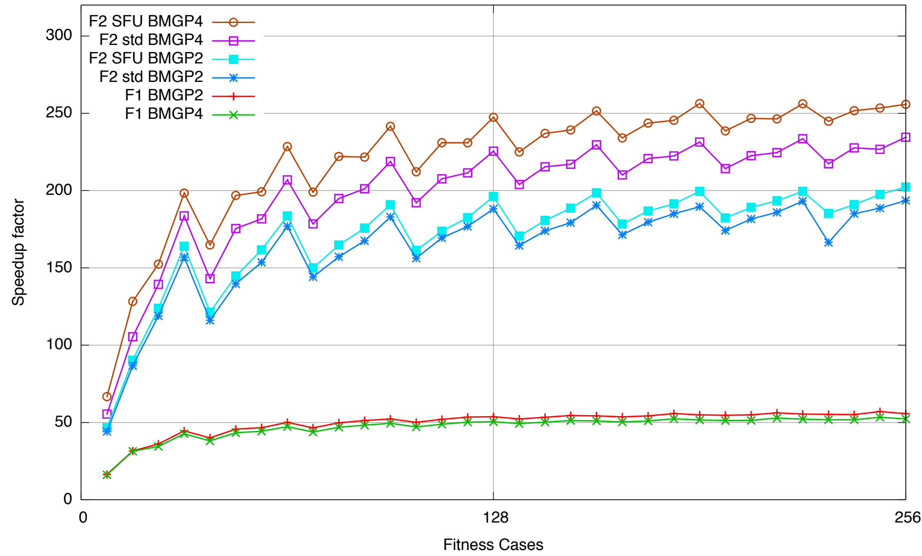


Figure 7.9: Resulting speedup for depth 7 trees on GPGPU against CPU for different function sets.

F1	F2
+, -, $\oplus$, $\times$	+, -, $\oplus$, $\times$, cos, sin, log, exp

Table 7.1: Two different learning function sets.

Influence of tree shapes on speedup

All the previous section experiments used the *full* construction method type. This method creates trees whose sizes are on average comparable, even with the use of unary functions (like sin or exp). It is an optimistic situation, because the evaluation of a set of 2 or 4 individuals on the same MP (*BMGP2* or *BMGP4*) ends at the same time for all individuals on average. In the case of a group containing short and long individuals, an individual can complete its evaluation before the others, leaving free scheduling slots and therefore degrading the memory behaviour.

In addition, the standard ramped half and half method introduced by Koza in [10] constructs trees of different sizes, ranging between minimum and maximum depth, using respectively the *full* and *grow* methods for each half of the population. The *full* method constructs trees of size t using only non-terminal nodes (of $arity \geq 1$) up to a level $(t-1)$, then uses terminal nodes (of $arity = 0$, leaves) for the last level.

The *grow* method randomly selects the type of nodes in the complete set of operators (terminals and nonterminals); at the last level, it requires the selection of nodes among the terminals. It is therefore possible that a branch ends well before the end. Fortunately, Koza excludes extreme possibilities by not allowing trees to be shallower than a predefined value, otherwise if the root node came to be a terminal, the resulting tree would have a single node.

These different methods are used to introduce genetic diversity at the beginning of the run, especially when they are used together (ramped half and half). As the evolution is taking place, the sizes of the individuals tend to get homogeneous, but there is no evidence that the trees should take a full shape.

The experiment presented in Figure 7.10 shows the speedups reached by the evaluation function

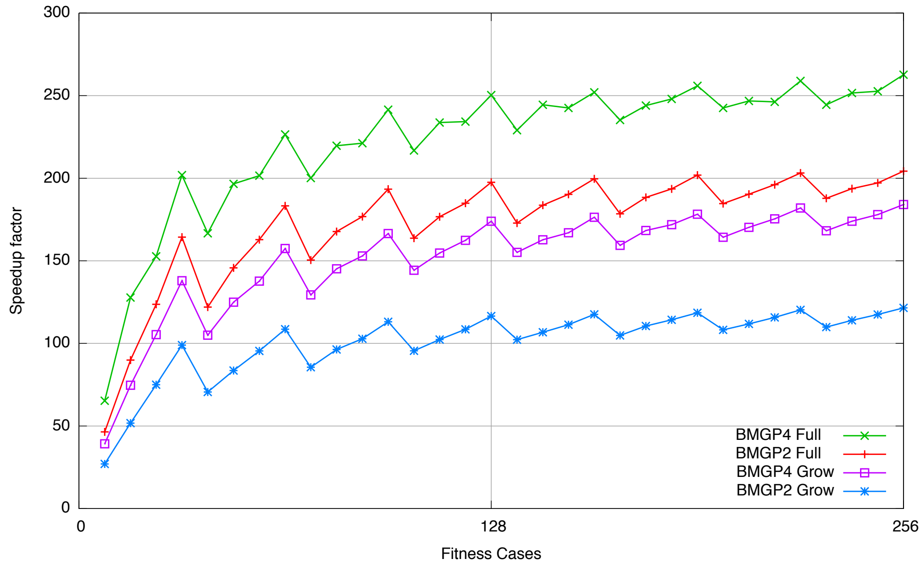


Figure 7.10: Resulting speedup for depth 7 trees on GPGPU against CPU for different tree construction methods.

on a 4096 individual population of a maximum depth of 7, using these two construction methods. The speedups drop from 250 to 170 using *grow* rather than *full*, confirming the advantage to evaluate similar sized individuals (at least in a matter of number of nodes).

Influence of population size

The population size is a factor which influences the performance. As we have seen, it is necessary to load the card with enough threads to keep it busy. A larger population means a larger number of blocks (scheduling sets), helping the cores load balancing. Using a multi-individual interpreter, as in the case of *BMGP*, leads to a division of the number of blocks loaded onto the card, because individuals now share blocks.

These effects are visible on Figure 7.11 where the *BMGP2* and *BMGP4* implementations reach their maximum speedup with a larger population than *blockGP32* or *blockGP128*. However, if the speedup stagnates with a smaller population, the overall speedup is more interesting for *BMGP2* and *BMGP4*.

7.4.3 Experiments on the complete algorithm

The analysis of the evaluation function allows to observe the behaviour of the parallel part of the algorithm. As it is the only part which is different from the sequential one, this analysis is important, in order to characterize the changes introduced by the use of GPGPU for genetic programming. Nevertheless, the study of the complete algorithm allows to take the final user point of view, for whom the overall speedup is the one that matters.

Theoretical speedups

Amdahl's law asserts that the speedup is limited by the sequential part of the algorithm. Indeed, imagining an infinite number of processors executing the parallel part, the execution time of this

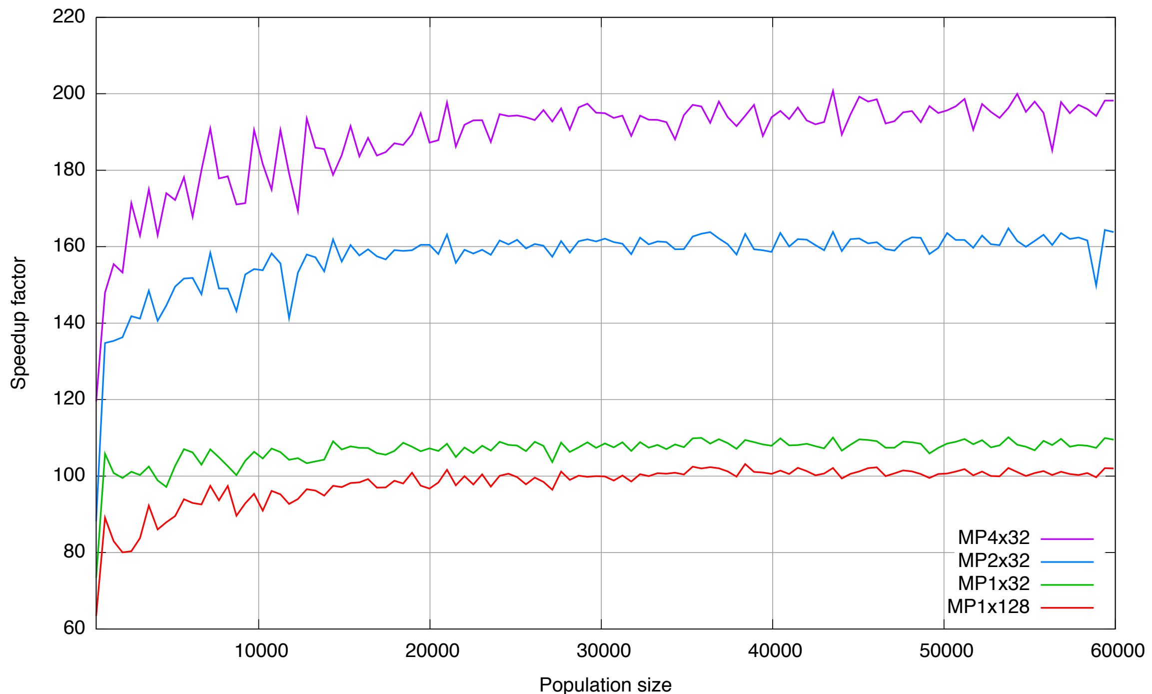


Figure 7.11: Resulting speedup for depth 7 trees, function set FS2SFU and 32 fitness cases on GPGPU against CPU *wrt* the population size for different implementations of GP evaluations.

part becomes nil. Finally, the only part that is left is the sequential part.

For our experiment, we use an algorithm whose fitness values do not depend on the individuals. Fitness values are fixed regardless of the evaluation method, so the evolution follows a deterministic path. This was introduced, as when using two different processors, CPU and GPGPU, executions give slightly different results for the same evaluation (the same individual, on the same training set). This makes the comparison between the two implementations difficult, because the differences in evolution imply structural differences in the individuals, which results in differences in execution time.

Using a fixed fitness value allows to remove the evaluation part. We simulate here an infinite number of processors, and thus calculate the maximum speedup that our implementation can obtain. These results are shown in Figure 7.12 for different population and training set sizes.

The maximum speedup is limited if the number of training cases is too small. This surface is related to our implementation, thus it is not possible to extrapolate these results to Genetic Programming in general.

Figure 7.13 presents the speedup obtained with the complete algorithm, including the multi-individual evaluation function. The maximum obtained speedup (134) is smaller than the speedup of the evaluation function only, and only a consequent increase in the number of training cases allows to find comparable values.

Nevertheless, the population size seems to matter more than in the time spent in the evaluation function only. Considering the population size used by Koza in his books (up to several million individuals), it seems that this constraint is not a really important one.

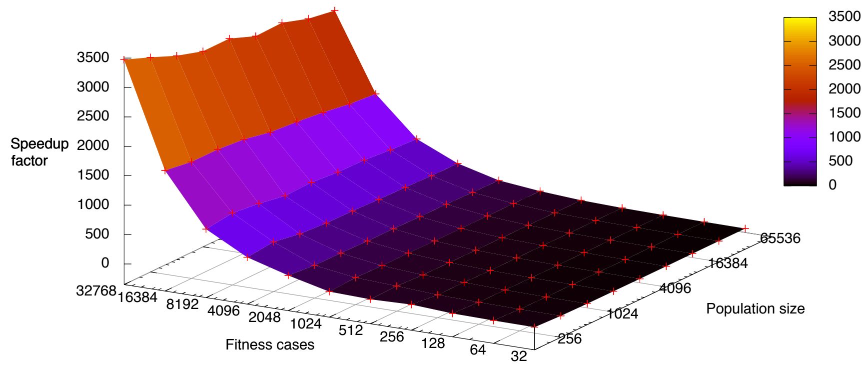


Figure 7.12: Maximal speedup with the tree-based GP implementation on the cos $2x$ problem.

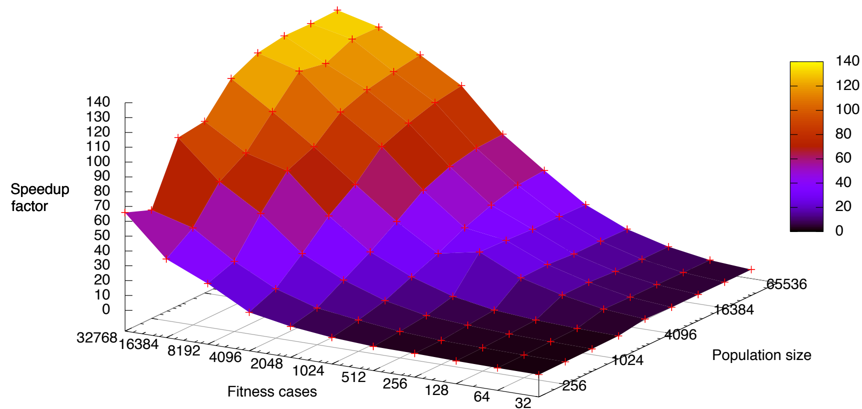


Figure 7.13: Speedup with respect to the population size and the number of fitness cases.

7.4.4 Example on a real world problem

Principle of an aircraft model

It is mathematically shown that in the real world, any dynamic system can be modeled through a nonlinear state space representation [79].

This consists in a mathematical model of the physical system, defined as a set of inputs, outputs and state variables related by first-order differential equations, such as:

$$\begin{cases} \dot{x}(t) = f(t, x(t), u(t)) \\ \dot{y}(t) = h(t, x(t), u(t)) \end{cases}$$

where x(t) is the state vector, u(t) the control vector and y(t) the output vector. The first equation is the “state equation” and the second one is the “output equation.”

The internal state variables (the state vector) are the smallest possible subset of system variables that can represent the entire state of the system at any given time. The minimum number of state variables required to represent a given system, n, is usually equal to the order of the defining differential equation of the system.

The control variables (the control vector) are the subset of variables which are used to drive the system.

If we consider the following state vector X: $x^T = [x_1\ x_2\ x_3\ \ldots\ x_n]$
and the following control vector U: $u^T = [u_1\ u_2\ u_3\ \ldots\ u_m]$
Then, the nonlinear state space representation is:

$$\begin{cases} \dot{x}_1(t) = f_1(t, x_1(t), \ldots, x_n(t), u_1(t), \ldots, u_m(t)) \\ \dot{x}_2(t) = f_2(t, x_1(t), \ldots, x_n(t), u_1(t), \ldots, u_m(t)) \\ \ldots \\ \dot{x}_n(t) = f_n(t, x_1(t), \ldots, x_n(t), u_1(t), \ldots, u_m(t)) \end{cases}$$

Aircraft nonlinear state space

In the aeronautical field, a nonlinear state space representation is often used to model aircrafts, in order to create autopilots. Thanks to control engineering, the linearization of a system around its equilibrium point allows to use all the useful tools provided by the science of automatics.

In this paper, we choose to model a small F3A airplane through its nonlinear state space representation, which is often flown in radio-controlled aerobatic airplane competitions, to have the capability to follow various trajectories, without major structural constraints.

The choice of the state vector is:

$$x^T = [V, \alpha, \beta, p, q, r, q_1, q_2, q_3, q_4, N, E, h, T]$$

where V is the airspeed, α the angle of attack, β the heeling angle, p the x-axis rotation rate, q the y-axis rotation rate, r the z-axis rotation rate, q_1 q_2 q_3 q_4 the attitude quaternions, N the latitude, E the longitude, h the altitude, T the real thrust.

The choice of the control vector is: $u^T = [T_c\ \delta_e\ \delta_a\ \delta_r]$

where T_c is the commanded throttle, δ_e the commanded elevators, δ_a the commanded ailerons, δ_r the commanded rudders, as in figure 7.14.

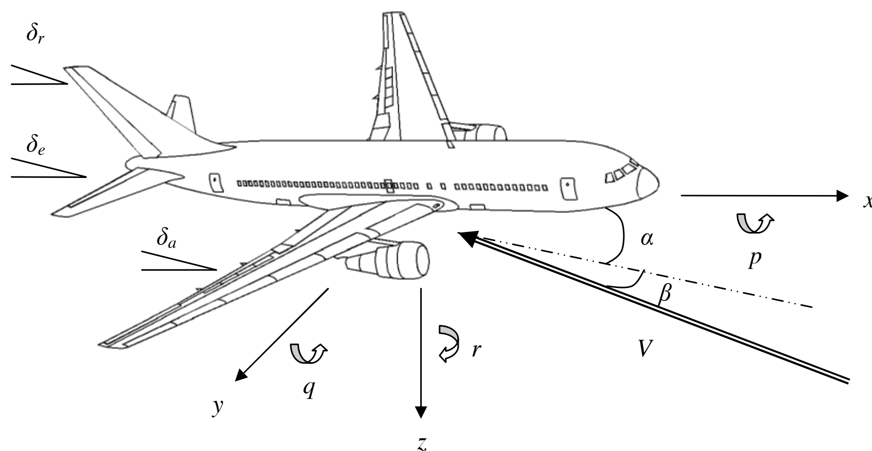


Figure 7.14: State and control variables of an airplane.

As described earlier, the nonlinear state space representation is: $\dot{x}(t) = f(t, x(t), u(t))$

And more precisely:

$$\begin{cases} \dot{V}(t) = f_1(t, V(t), \alpha(t), \dots, T(t), T_c(t), \dots, \delta_r(t)) \\ \dot{\alpha}(t) = f_2(t, V(t), \alpha(t), \dots, T(t), T_c(t), \dots, \delta_r(t)) \\ \dots \\ \dot{q}_1 = f_7(t, V(t), \alpha(t), \dots, T(t), T_c(t), \dots, \delta_r(t)) \\ \dot{q}_2 = f_8(t, V(t), \alpha(t), \dots, T(t), T_c(t), \dots, \delta_r(t)) \\ \dot{q}_3 = f_9(t, V(t), \alpha(t), \dots, T(t), T_c(t), \dots, \delta_r(t)) \\ \dot{q}_4 = f_{10}(t, V(t), \alpha(t), \dots, T(t), T_c(t), \dots, \delta_r(t)) \\ \dots \\ \dot{h}(t) = f_{13}(t, V(t), \alpha(t), \dots, T(t), T_c(t), \dots, \delta_r(t)) \\ \dot{T}(t) = f_{14}(t, V(t), \alpha(t), \dots, T(t), T_c(t), \dots, \delta_r(t)) \end{cases}$$

Considered functions

In this experiment, only the quaternion functions have been regressed using genetic programming. The generic quaternion functions in the state space are:

$$\begin{cases} \dot{q}_1 = f_7 = 0.5(q_4 p - q_3 q + q_2 r) \\ \dot{q}_2 = f_8 = 0.5(q_3 p + q_4 q - q_1 r) \\ \dot{q}_3 = f_9 = 0.5(-q_2 p + q_1 q + q_4 r) \\ \dot{q}_4 = f_{10} = 0.5(-q_1 p - q_2 q - q_3 r) \end{cases}$$

Table 7.2: Parameters used to regress a part of the airplane model.

Population size	40 960
Number of generation	100
Function set	+, -, *, /
Terminal set	x[1]...,x[17], ERC {0,1}
Learning set	51000 values

An artificial model (an F3A simulator) is used to generate a few minutes of flight. Flight telemetry data is saved into a file, which will serve as a training set for genetic programming.

All the state variables $[V, \alpha, \beta, p, q, r, q_1, q_2, q_3, q_4, N, E, h, T]$ have been recorded, as well as the control variables $[T_c, \delta_e, \delta_a, \delta_r]$ and the time. The learning set contains 51,000 points, *i.e.* around 8 minutes of flight. Run parameters are summarized in table 7.2.

Figure 7.15 shows speedups obtained with our implementation, with the given parameters. In this case, the serial part of the problem is negligible compared to the evaluation time. Given this implementation, the maximum theoretical speedup ranges between more than 100 and 500. Those maximal speedups are computed, as in the previous section 7.4.3, by removing the evaluation function during a run, therefore simulating an infinite speedup for the evaluation function.

Because of the simple function set (that does not contain complex functions such as sine, cosine, exponential, that are approximated by SFUs in a very efficient way), the speedup is lower than that shown in section 7.4 for more complex function sets. In the current experiment, the terminal set is larger (ERC, 17 variables), *i.e.* the GPU interpreter has to perform more memory accesses than the one presented in the benchmark work, where only variables can be stored in the registers

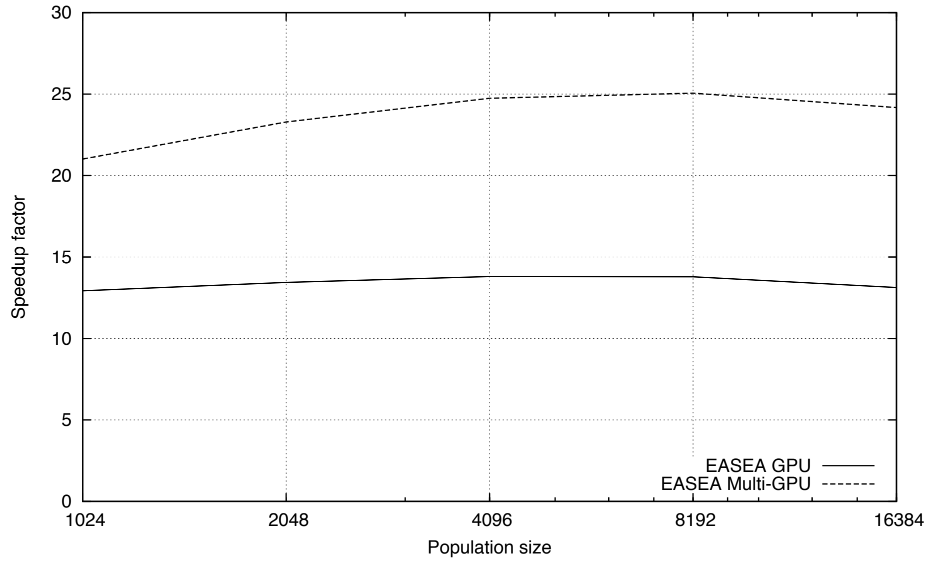


Figure 7.15: Speedup obtained on the airplane problem with EASEA tree-base GP implementation.

by the optimizer. These drawbacks as well as the evolutionary process can be blamed for the drop in speedup.

Speedup factors still remain satisfactory given the size of the problem: a real run takes hours of computation on CPU, but just several minutes on GPU. This is very important, because this problem needs to regress 14 different functions.

Figure 7.16 shows an example of a trajectory obtained with the evolved model as well as the trajectory resulting from the trained model.

Functions that have been evolved with the EASEA GP implementation have been used instead of the real ones. It has to be noted that quaternion functions that have been evolved through symbolic regression impact the plane trajectory in a very strong manner, meaning that errors in this part of the model will have noticeable consequences. In this example, the difference in trajectories between the reference model and the best trajectory obtained in generation 250 is so minimal that it is not possible to distinguish between them. The best trajectories obtained by the best individual of generation 1, 10, 20 and 50, show that wrong quaternion equations have a real influence on the resulting trajectory.

Theses results were presented in conference papers, in Maitre et al. [80], which details the parallelisation of the evaluation part, and in Ogier Maitre [81], where the whole algorithm has been detailed. The complete work was part of the ones published in Maitre et al. [71].

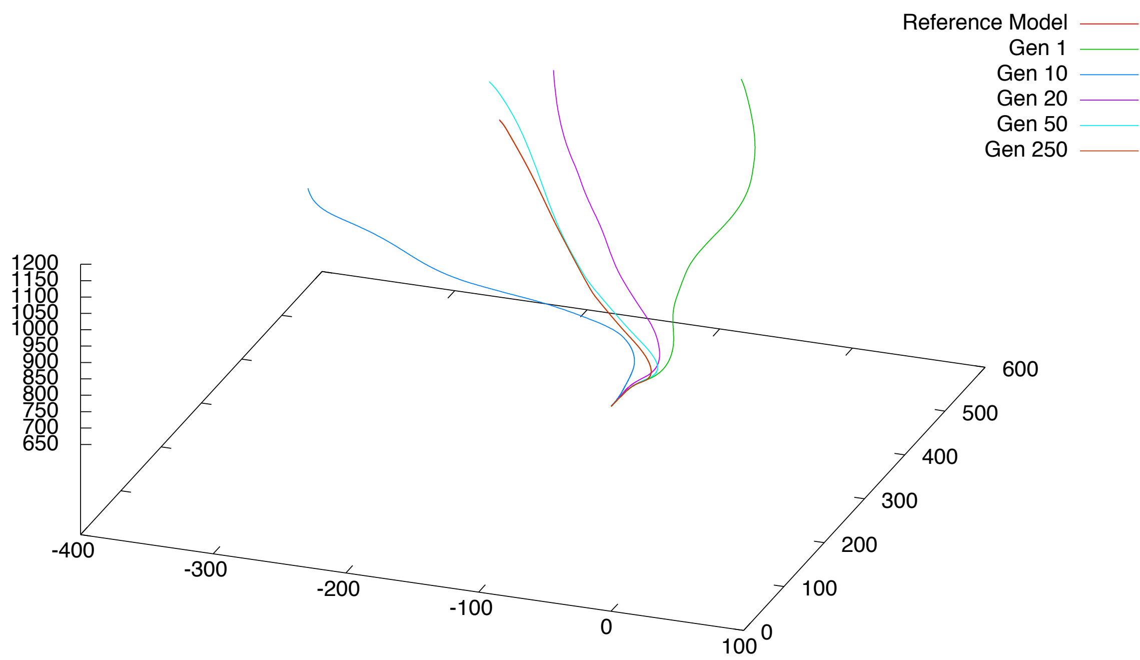


Figure 7.16: Simulated trajectories, from the original model and the evolved one.

Chapter 8

A complete synchronous EA running in parallel on GPGPU

Contents

8.1 Tournament study **90**
8.1.1 Analysis of the population distribution 91
8.1.2 Reproduction rate 92
8.1.3 Selection intensity 92
8.1.4 Loss of diversity 93
8.1.5 Experimental measurements on tournament for reduction 95
8.2 DISPAR-tournament **95**
8.2.1 Principles 95
8.2.2 Experiments with DISPAR 96
8.3 DISPAR-tournament on GPGPU **97**
8.4 Complete generational algorithm on a GPGPU card **99**
8.5 Experiments **101**
8.5.1 Speedup 101
8.5.2 Timing distribution analysis 103
8.5.3 Qualitative behaviour on the Weierstrass function 104

As detailed in section 6, the parallelization of the complete algorithm faces the problem of the reduction function. This function uses a selection operator (tournament, roulette-wheel,...) without replacement to select μ individuals out of $(\mu + \lambda)$. Once an individual is selected to populate the next generation, it must not be selected again, so that there are no clones in the new generation.

Of course the creation of clones still is possible in the individuals' creation, if for instance, a child is created without using the crossover operator and undergoes no subsequent mutation, but this is controlled by the crossover probability.

If the possibility is given to the reduction operator to select the same individual several times, then good individuals will likely appear many times in the new generation, if some selection pressure is applied when selecting individuals to populate the new generation. This selection pressure is one of the fundamentals of Darwinism.

Avoiding clones creation is easy to achieve in a sequential algorithm: select an individual from the (*parents+children*) population, *move* it to the next generation population, repeat μ times. It can therefore not be selected again.

This approach cannot be applied to the parallel case, as even if a selected individual is removed from the *(parents+children)* set, it is not possible to guarantee that the same individual has not been simultaneously selected by several different cores. Furthermore, because of selection pressure, good individuals have a greater than normal probability to be selected. As the number of cores is high on GPGPU chips, population reduction is problematic.

It is therefore necessary to design a new operator to this specific use, that should be able to satisfy the following constraints:

- be massively parallel,
- avoid selecting identical individuals,
- behave as closely as possible to a standard selection operator, so that all the conclusions on EA literature are still valid when this new operator is used.

We chose to mimic a tournament operator, described by algorithm 6, as it is the selection operator that is used in $(\lambda + \mu)$-ES and the most suitable for an easy parallelization. In algorithm 6, the difference between a parent selector for reproduction (= selection with replacement) and a population reductor (= selection without replacement) appears on the last line of the algorithm. This (and the possibility of selecting the same individuals several times in parallel) is what makes it impossible to parallelize a standard tournament (or any other standard selection without replacement method, by the way).

Input: μ: the size of the parent population, λ: the size of the children population,
I: A population of $(\mu + \lambda)$ individuals, t: the tournament size
Output: I': A population of μ individuls
while *size(I') < μ* **do**
  BestCompetitor := uniformly select an individual from I
  for *t times* **do**
    CurrentCompetitor := uniformly select an individual from I
    if *CurrentCompetitor→fitness is better than BestCompetitor→fitness* **then**
      BestCompetitor := CurrentCompetitor
    end
    I'.push(BestCompetitor)
    //The line below is for a tournament without replacement:
    `I:= I - BestCompetitor`
  end
end

Algorithm 6: An implementation of a reduction method using Tournament Selection, with or without replacement.

8.1 Tournament study

As mentioned before, the tournament operator has been chosen as a basis because it is widely used in artificial evolution and above all, it has a low complexity, which is not dependent on the population size ($O(t)$, where t is the size of the tournament). Finally, this operator has been studied in many papers [22, 82, 83].

However, the only studies found on tournament selection were for selecting parents in order to produce offspring (breeders selection step), *i.e.* with replacement of the selected individual in the original population, so that good individuals can breed several times. Intuitively, a selection operator would work differently when the selected individual is removed from the original population.

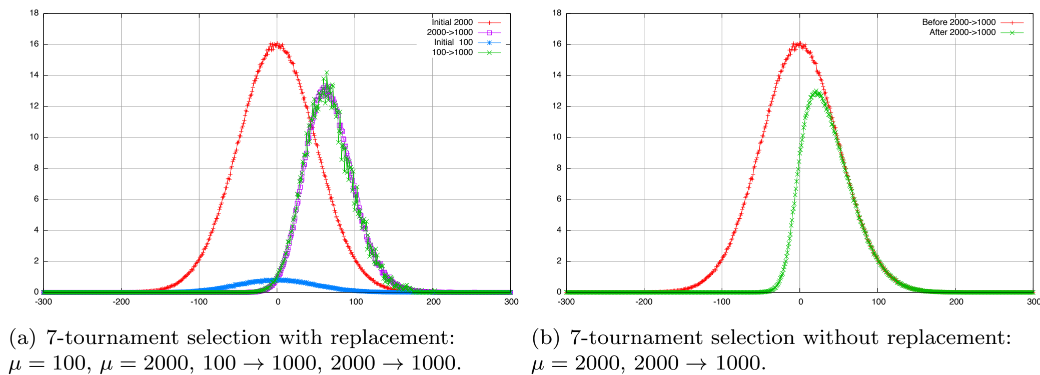
It is therefore necessary to conduct a study, at least experimentally, on a tournament operator without replacement for population reduction.

Starting with a thorough paper on selectors with replacement, [22], we decided to analyze this operator by measuring the loss of diversity (LoD) of the operator, and its selection intensity (SI) .

8.1.1 Analysis of the population distribution

In this section, we will study a population before and after applying multiple selections. The tournament algorithm used is presented in Algorithm 6. The population is simply a uniform set of random fitness integer values. It is generated using a normal distribution centered on 0 and with a standard deviation of 50 ($G(0, 50)$).

Tournament selection



(a) 7-tournament selection with replacement: $\mu = 100$, $\mu = 2000$, $100 \to 1000$, $2000 \to 1000$.

(b) 7-tournament selection without replacement: $\mu = 2000$, $2000 \to 1000$.

Figure 8.1: Population distribution by fitness value for 1000 7-tournament selections "with" and "without" replacement, from a population of 100 and 1000. (number of individual *w.r.t.* rounded fitness values.)

Figure 8.1(a) shows two populations of 100 and 2000 individuals, before and after application of a tournament selection operator. In order to get smooth curves, these values have been averaged on 1000 runs.

The two initial populations are centered on 0. Out of these two initial populations, 1000 individuals are selected with replacement. The result is two curves with similar distributions, shifted to the right, with more noise on the population that was created from the smaller population of 100 individuals. It is interesting to see that the same selection pressure (7-tournament) on two populations of radically different sizes returns a population with a similar distribution.

The selection pressure "pushes" the population to the right, *i.e.* to higher fitness values. With a 7-tournament, virtually no individuals with a fitness value under 0 are selected (virtually all selected individuals are part of the better half of the initial population).

Figure 8.1(b) shows what happens in the case of a selection without replacement. The modification to the algorithm is very limited (removing the selected individual from the pool), but the influence on the resulting distribution is very important. As it is impossible to select the same individuals several times, there is no way to go beyond the right part of the original curve. The distribution of the population has an area that is half that of the original distribution area. The larger the tournament size (and therefore the selection pressure), the more the target distribution is collapsed to the right. If the pressure is very high, this type of selection becomes similar to a "truncation selection," *i.e.* only the best are selected.

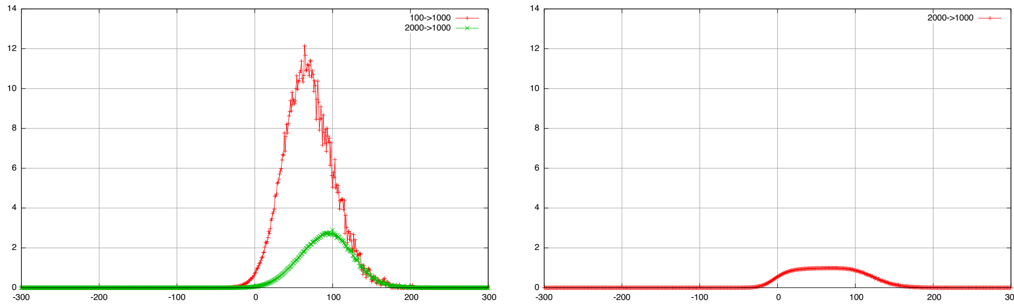
8.1.2 Reproduction rate

The reproduction rate gives, for a type of individuals, a multiplicative factor between the initial and target distributions. It is defined in [22] as equation 8.1:

$$R(f) = \begin{cases} \dfrac{s^*(f)}{s(f)} & \text{if } s(f) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{8.1}$$

where $s(f)$ and $s^*(f)$ denote the fitness distribution before and after selection, *i.e.* the number of individuals having fitness f.

Thus, this metric gives an idea of the cloning factor of an individual type in the target population.



(a) 100 → 1000 and 2000 → 1000 7-tournament with replacement selection.

(b) 2000 → 1000 7-tournament without selection.

Figure 8.2: Mean fitness distribution and repredocution rate over 1000 experiments for with and without replacement selection. (*w.r.t.* rounded fitness values.)

For a selection with replacement, there is a difference between the cloning factor of 100 → 1000 and 2000 → 1000 selections. When selecting 1000 individuals out of 100 it is necessary to clone some individuals, but individuals below average show a very low reproduction rate. When selecting 1000 out of 2000 with replacement, cloning will happen, but to a lesser extent.

Finally with a tournament selection without replacement, cloning is impossible. Therefore, the maximum selection of a fitness value for the next generation is one.

8.1.3 Selection intensity

Selection intensity is defined in [22] by equation 8.2 where M is the mean fitness of the population *before* selection, M^* the mean fitness of the population *after* selection and σ the standard deviation of the first population :

$$I = \frac{M^* - M}{\sigma} \tag{8.2}$$

In the same paper, the authors also define a theoretical value for selection intensity based on the size of the applied tournaments for a Gaussian distribution $G(0,1)$. This value is found using the following approximation:

$$I_T(t) \approx \sqrt{2ln(t) - ln(\sqrt{4.14ln(t)})} \tag{8.3}$$

This formula gives an approximated value with less than 2.4% error for $t \in [2, 5]$ and an error of less than 1% for $t > 5$.

However, as usual, these values are only applicable to tournaments *with* replacement and nothing is said about tournament *without* replacement.

Intuitively, a tournament without replacement has a lower selection intensity than a tournament with replacement. Indeed, the tournament without replacement cannot create clones and as the selection is biased towards good individuals, in the beginning of the process, selection with or without tournament has the same probability of choosing the very good individuals. However, where tournament with replacement keeps the same probability to pick up these very good individuals, this probability decreases in tournament without replacement, because every time a good individual has been chosen, it is removed from the pool of individuals out of which they are selected, so it is not possible to pick good individuals several times.

This effect can be seen experimentally in Figure 8.3, which shows the selection intensity *w.r.t.* to tournament size, from equation 8.3 and experimental curves for tournament selection *with* and *without* replacement.

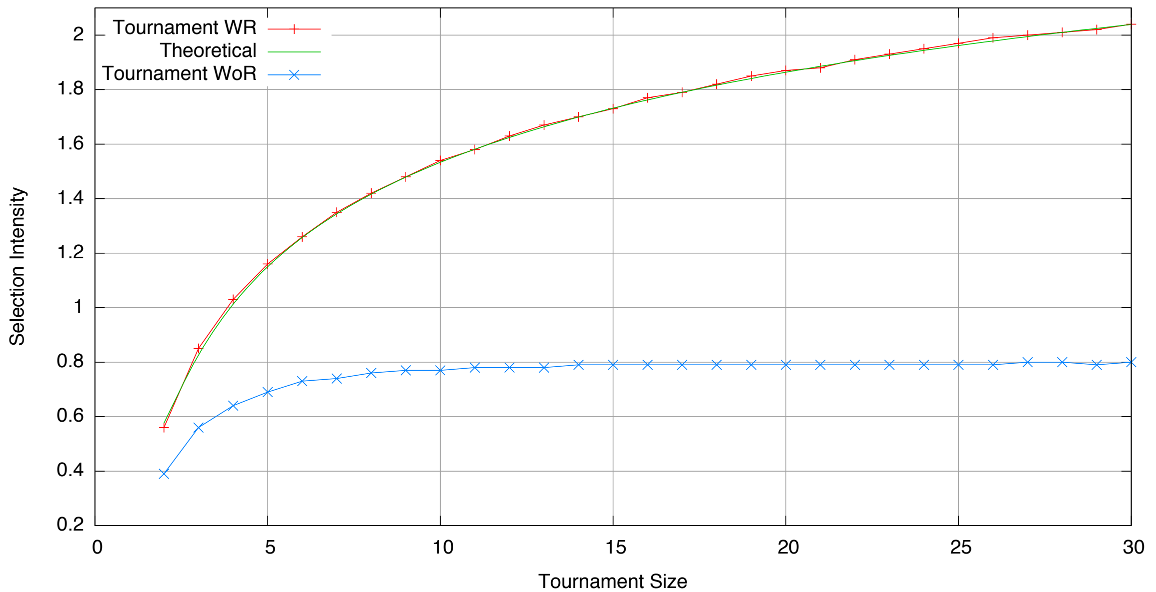


Figure 8.3: Selection intensity for a tournament selector with and without replacement for 2000 $\rightarrow$ 1000 reduction. The theoretical curve (top curve) plots the equation found in Blickle and Thiele [22]. It is superimposed with the experimental tournament with replacement curve.

The tournament with replacement experimental and theoretical curves are nearly identical. It is interesting to see that where selection pressure keeps increasing in the tournament with selection, it becomes nearly flat beyond tournament size 10 for tournaments without replacement.

8.1.4 Loss of diversity

In Blickle and Thiele [22], loss of diversity is described as "the proportion of individuals of a population that are not selected during the selection phase." This is detailed in equations 8.4 and 8.5. One can see this measurement as the number of individuals which disappear from the population.

Summing $L(f)$ for every fitness f of the population gives the experimental loss of diversity, noted D_T in Equation 8.5.

$$L(f) = \begin{cases} s(f) - s^*(f) & \text{if } s^*(f) < s(f) \\ 0 & \text{otherwise} \end{cases} \tag{8.4}$$

$$D_T = \sum_{f \in s} (L(f)) \tag{8.5}$$

In [22], Blickle and Thiele give equation 8.6 to calculate the theoretical loss of diversity for the tournament operator, *w.r.t.* the tournament size (t). Motoki in [82] suggests equation 8.7, for a better accuracy, as it takes into account the population size N:

$$D_T(t) = t^{-\frac{1}{t-1}} - t^{-\frac{t}{t-1}} \tag{8.6}$$

$$D_T(t, N) = \sum_{k=1}^{N} \frac{1}{N} (1 - \frac{k^t - (k-1)^t}{N^t})^N \tag{8.7}$$

Here again, both papers consider selection with replacement, which does not concern the reduction step, where the population is reduced from $(\mu + \lambda)$ individuals to μ.

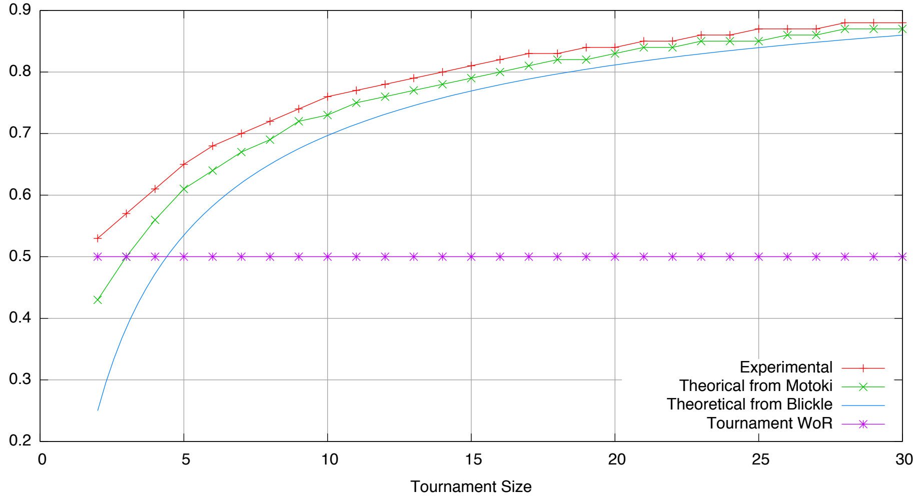


Figure 8.4: Loss of Diversity for (1000+1000)-ES selection using tournament selection.

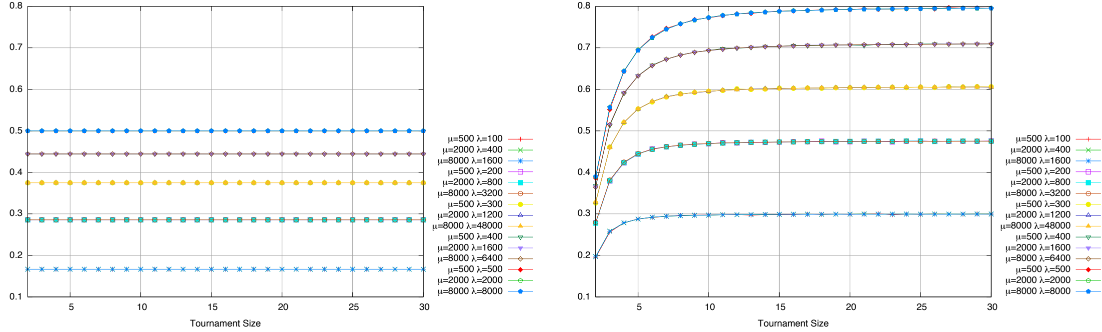
Figure 8.4 presents loss of diversity with respect to tournament size, observed on a population of size 2000 reduced to 1000. The theoretical curves according to Blickle *et al.* and Motoki are also represented. The error between our experimental values and those given by Motoki's formula ranges between 19% for $t = 2$ and 1% for $t = 30$ and the error falls rapidly to less than 3% for $t = 10$. Our experimental data shows a higher error rate in the case of Blickle's formula, starting at 65% for $t = 22$, but is then less than 8% for $t = 10$.

These curves were meant for a tournament with replacement. For a tournament without replacement, loss of diversity from 2000 individuals to 1000 individuals is obviously 0.5, whatever the tournament size.

8.1.5 Experimental measurements on tournament for reduction

We have seen that tournaments without replacement behave in a quite different way from tournaments with replacement. One identified factor is the $\mu/(\mu+\lambda)$ ratio, and so we seek to further observe the behaviour of tournament without replacement with different population sizes.

Figures 8.5(a) and 8.5(b) present these experiments on different initial and target population sizes.



(a) Loss of Diversity for Different Population Sizes (curves are superimposed three by three).

(b) Selection Intensity for Different Population Sizes (curves are superimposed three by three).

As previously, the $\mu/(\mu+\lambda)$ ratio determines the loss of diversity, whatever the size of the tournament. This ratio affects the selection intensity, even if it does not remain constant as in the case of a loss of diversity.

This section was written to show that selection operators without replacement (that have not been studied yet) behave quite differently from selection operators with replacement. Selection operators without replacement would deserve a more thorough analysis which is out of the scope of this PhD thesis. It is important to remember that they are highly influenced by the children/parents population size ratio.

8.2 DISPAR-tournament

8.2.1 Principles

In this section, our main goal is to propose DISPAR-Tournament (Disjoint Set Parallel Tournament), a parallel selection operator *without* replacement, that could be executed efficiently on a multi-core architecture for the reduction phase of a $(\mu+\lambda)$-ES algorithm, for instance.

DISPAR-Tournament is an operator which assigns a subset $t = (\mu+\lambda)/T$ to T parallel threads, where T is preferably greater than the number of physical cores c of the system.

Each thread will have the task to preserve the k best individuals of t, with the constraint that $k \leq t$ (be aware however that if $k = t$, no population reduction will be performed). Because these tournaments will return more than one individual, they will be referred to as multi-tournaments in the rest of this document.

By assigning T multi-tournaments to c cores, the initial population is reduced to a target population of size $T \times k$ without any clones and with no communication between the cores.

Morever, because each thread selects the k best individuals of all the t subsets, DISPAR is also elitist (the best individual *will* be preserved) still without any communication between the cores.

For memory access efficiency reasons, the t individuals affected to a thread T are contiguous in memory. If the population were organized into two blocks (a children block and a parent block, for

instance), all the first multi-tournaments would be performed between children only and the last ones between parents only.

This could introduce some (desirable or not) biases in the selection of the next generation as for instance, the p multi-tournaments made of parents only would select only parents, meaning that the $p \times k$ parents selected would make it to the next generation, even if all individuals in the children population were better than their parents. Another side-effect would be that both best individuals of the children and parent population would be elected to be in the new generation, which again, depending on the problems, might be desirable or not.

Even though such features could be useful for diversity preservation, for instance, we have not investigated their effects yet, so we prefer to implement a behaviour that would be similar to what anyone would expect from a standard $(\mu + \lambda) \rightarrow \mu$ tournament reduction operator.

To this effect, it is recommended that any implementation of DISPAR-tournament makes sure that individuals are randomly distributed in the population without any parental relationship between neighbour individuals.

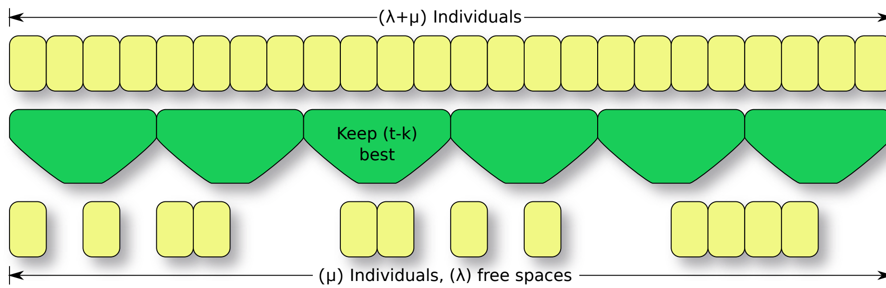


Figure 8.5: DISPAR-Tournament principles.

Figure 8.5 illustrates this principle, where a population of size $(\mu + \lambda)$ is uniformly distributed (between parents and children) and multi-tournaments are performed on $(\mu + \lambda)/t$ threads with $t = 4$ and $k = 2$.

DISPAR-Tournament works on different subsets, meaning that no individual can be selected twice, by two different threads, as each one participates to only one tournament. The stochastic aspect is implemented by the uniform distribution of the individuals in the population. The μ and λ population sizes are constrained.

Note that selecting the k best individuals of all T multi-tournaments is not equivalent to selecting the best $k \times T$ individuals of the whole population. As for standard tournament, it is possible that all the worst individuals of the population appear in a single t subset, in which case the k best of the t worst individuals will be selected.

8.2.2 Experiments with DISPAR

DISPAR is a reduction operator which is parallelizable on shared-memory parallel architectures.

Figure 8.6 presents experiments on selection intensity, calculated as above. DISPAR-Tournament is not easily adaptable to any pressure for any population size. For instance, for $\mu = 1000$ and $\lambda = 1000$, it is not possible to obtain a $\mu + \lambda \rightarrow \mu$ reduction with a 3-multi-tournament. Dividing the 2000 individuals in sets of 3 individuals means roughly 666 multi-tournaments. If these return the best of the three individuals, then, the new generation will only contain 666 individuals (where 1000 would have been needed). If k is set to 2, the 666 multi-tournaments will return 1333 individuals rather than the 1000 that were needed.

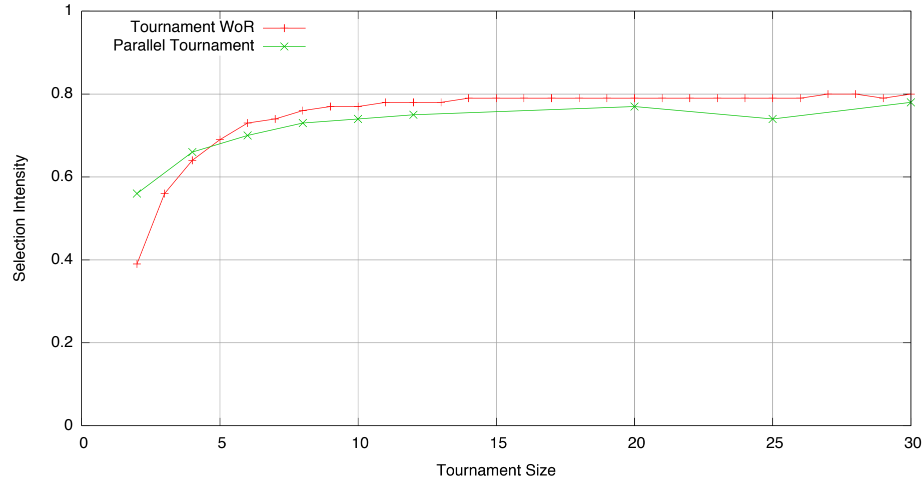


Figure 8.6: Selection intensity for (1000,1000)-ES reduction, with standard tournament without replacement and DISPAR-tournament. Tournament size is $t \to 1$ for tournament without replacement, but for DISPAR-tournament, the value of k is calculated depending on the number of individuals that are needed for the next generation. Because 2000 individuals must be reduced to 1000, multi-tournament parameters chosen for DISPAR-tournament are respectively $2 \to 1$, $4 \to 2$, $6 \to 3$, $8 \to 4$, $10 \to 5$, $\ldots$

However, it is possible to obtain 1000 individuals with a 3-multi-tournament if $\mu = 1000$ and $\lambda = 2000$.

So population sizes and tournament sizes must be adapted in order to be able to implement DISPAR-tournament on a parallel machine. However, as the selection intensity varies only slightly above a certain threshold, these are not very hard constraints to match, as seen in Figure 8.6. The only real discrepancy in selection intensity is for 2-tournament and 2-DISPAR-tournament.

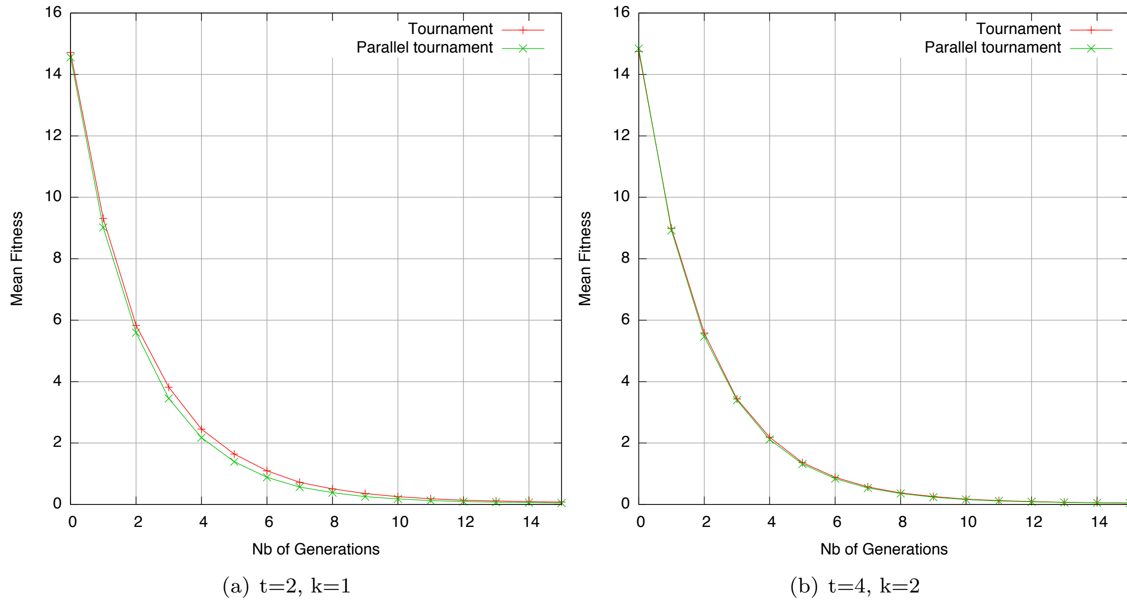
Figure 8.7 shows the evolution of the mean fitness (on an average of 50 runs) of the population on a “sphere” problem with a (1000+1000)-ES. This problem was chosen because it is a simple convex problem, where selection intensity has a strong influence on results. In figure 8.7(a) one can see that the stronger selection intensity implemented by the $2 \to 1$-DISPAR tournament over a standard 2-tournament without replacement results in a faster convergence towards 0.

If a lower selection intensity were needed, it would be possible to simulate it using a stochastic binary tournament method, where out of the two individuals, the best is returned with a probability $p < 1$.

Figure 8.7(b) presents the same experiment, but for a size 4 tournament without replacement and a $4 \to 2$ DISPAR multi-tournament. One can see on Figure 8.6 that the selection intensities are very similar for these two implementations. This is confirmed in Figure 8.7(b) where the two curves are nearly superimposed.

8.3 Using DISPAR-tournament to implement a complete EA on GPGPUs

With DISPAR, we now have a parallel operator suited to shared-memory parallel architectures that is able to reduce a population. It is now possible to implement a complete evolutionary algorithm on GPGPU.



(a) t=2, k=1

(b) t=4, k=2

Figure 8.7: Mean fitness on the sphere problem.

We noted in the section presenting the parallel panmictic GA/ES from Chapter 6, that this implementation is not efficient in the case of short evaluation functions, due to both the Amdahl law and overhead due to CPU $\longleftrightarrow$ GPU memory transfers.

All these problems disappear if the complete algorithm runs on a GPGPU card. For maximum efficiency, such an implementation should use as many threads as possible (allowing the card scheduling hardware to overcome memory latencies) and exploit coalescent memory accesses in order to improve memory bandwidth.

Finally, the algorithm should be as comparable as possible with a standard sequential algorithm. To this end, variation operators need to be as close as possible to what is used in the standard version.

Figure 8.8 presents the main principles of an algorithm completely implemented on GPGPU. The population is initialized by the GPGPU in its own memory space. It is composed of $\mu + \lambda$ slots, where λ is the number of children and μ the number of parents.

To start with, parents and children slots are evenly distributed in memory, with one child every p parents if there are fewer children than parents, or conversely, one parent every c children if there are more children than parents. A table stores the addresses of the children (this part is not shown in the figure).

During initialization, only the parents slots are populated, instantiated and evaluated in parallel.

The CPU then launches in parallel all λ threads running the algorithm. To each thread is associated a free slot to receive a child. Each thread selects two parents, using a standard tournament, with replacement, in the global population. Then, it executes a crossover on parents to create a child and mutates it. This new individual will be stored into the open slot associated with the current thread. Finally, each thread performs the evaluation of the produced child.

Then the required number of threads execute an instance of DISPAR-tournament on a population subset. Because the population is divided into disjoint sets, fewer threads will be necessary than were needed to create all the children.

The threads running the DISPAR tournament will assign the slots of individuals to be removed

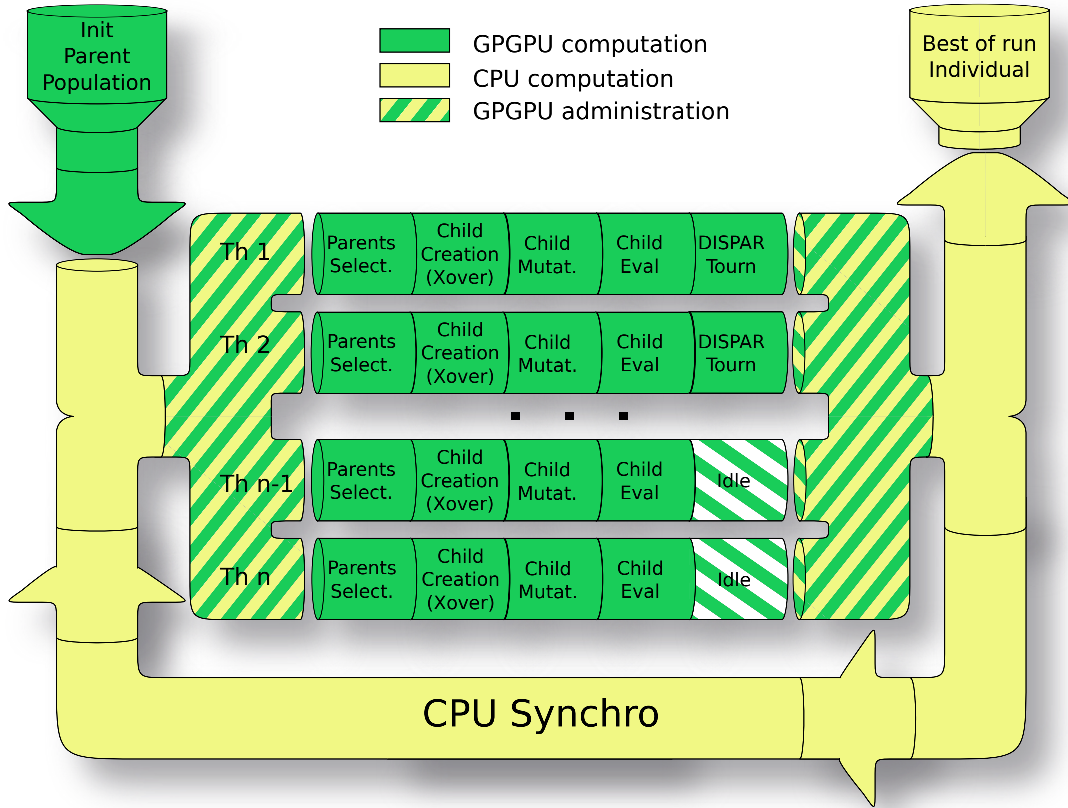


Figure 8.8: A schematic view of a full GPGPU DISPAR-EA.

from the population to all the λ threads. The threads will use these slots to store the child they will create.

The algorithm returns briefly to the CPU, in order to synchronize all threads. If the number of generations reaches its limit, the population is transferred back to the CPU memory, and the best individual is returned. Otherwise, the processor executes another generation similar to the previous one.

The synchronization of all the threads in the GPGPU can only be done by terminating the GPGPU program and then by waiting on the threads. Data stays in the GPGPU memory, so that the code executed by the processor for synchronization is minimized.

The inclusion of children within the global population is intended to maintain its randomness. Thus, there is no clear separation between these children and parents so as to avoid biases in the DISPAR-tournament reduction phase.

8.4 Complete generational algorithm on a GPGPU card

In order to create the new generation, Generational Genetic Algorithms (GGA) simply replace all the μ parents with the newly created λ children. In this paradigm, λ is typically equal to μ. No reduction phase is required, and no clones can be created.

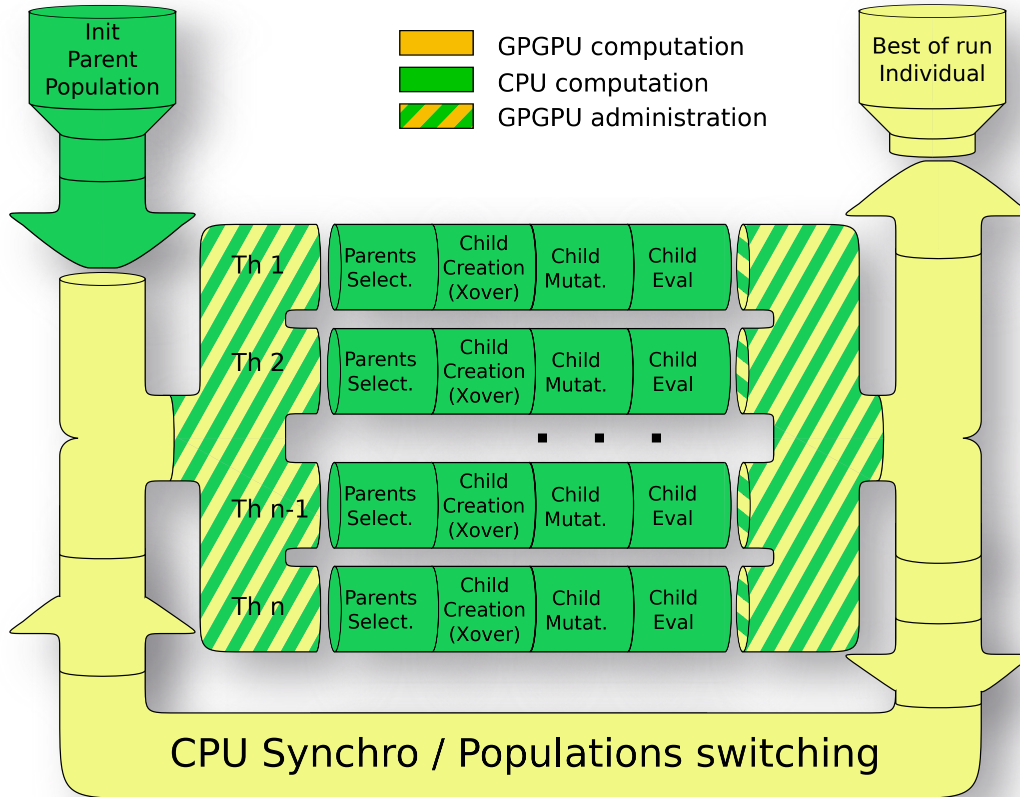


Figure 8.9: A schematic view of a full GPGPU generational EA.

In addition, we shall see that GGAs have some implementation advantages in terms of predictability, which allows to use the wide memory bus of the GPGPU processor. Figure 8.9 presents a GGA running completely on a GPGPU. Compared with the DISPAR-GPU algorithm presented in the previous section (8.3), this algorithm does not implement any reduction phase at all.

In this variant, the GPGPU stores two different populations that are respectively the parents and children population. Individuals are stored in order to group the same genes of different individuals for a coalescent access, as shown in Figure 8.10. So when a group of neighbouring threads accesses the same gene in neighbour individuals, it is possible to group memory accesses in accesses as wide as the memory bus, without loading unnecessary data. During synchronization, the CPU exchanges the addresses of these two buffers, the former parent buffer becomes the children population and children become the parents of the current generation.

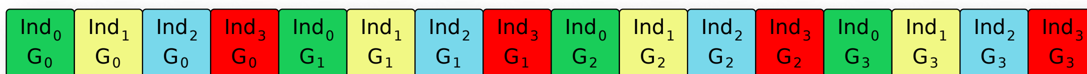


Figure 8.10: Individuals organization for genGPU algorithm.

Removing the reduction phase accelerates the execution of the algorithm. It also allows to keep parents and children in two groups in memory (to the contrary of DISPAR-tournament where parents and children needed to be interleaved). The parent selection step does not take advantage of coalescent accesses, as it needs the fitness values of randomly selected individuals.

The child creation step does not benefit from this coalescent memory while using the parents, as they were chosen in the previous phase. It is not possible (or even desirable) to predict which parents will be selected. Nevertheless, accesses to the children are coalescent, as neighbour threads create children which are neighbour in memory.

Mutation uses the produced child, and possibly a self-adaptive mutation rate. This data is accessed in a coalescent manner as this function is acting on the output of the previous step.

Similarly, evaluation uses the child produced by the two previous functions, so the same behaviour occurs again.

In general, memory accesses targeting the parents are not coalescent. They cause high memory latency and they load many pieces of useless data, which will probably be removed from the cache memory before being useful.

All operations involving children can benefit from the bus width and then from the high memory bandwidth offered by a GPGPU.

8.5 Experiments

The implementation using GPGPUs for the evaluation step presented in section 6 obtains mixed results when the evaluation function is not heavy enough.

First, the computation time of the evaluation function is too short to justify the transfer of individuals to the GPGPU memory. On the other hand, the overall time to evaluate the population is too small to obtain some gain in the parallelization. Again, according to Amdahl's law, if the evaluation function takes a time equal to $1/N$ of the total algorithm, it is not possible to obtain a speedup greater than N.

The two presented algorithms have the double advantage of avoiding the transfer between the memory spaces of the individuals and the fitness values, but also to parallelize all phases of the algorithm. Thus, the transfer time is of course no longer a problem, and similarly the parallel part of the algorithm is almost the entire running time.

The following tests have been performed on a Linux 2.6.32, 10.04.2 Ubuntu with NVIDIA driver 260.19.26. The CPU used is an Intel$^{(R)}$ Core$^{(TM)}$ CPU i7 950 clocked at 3.07GHz and the GPGPU card is a GTX480. Only one core has been used for CPU implementations because, as was already written, the idea is to compare a massively parallel algorithm to a standard sequential algorithm as is usually found in the literature (and not a massively parallel algorithm to another parallel algorithm).

8.5.1 Speedup

The Rosenbrock function (*cf.* Figure 8.11) is a typical problem in which the use of a mixed CPU/GPGPU approach brings many problems. Indeed, the Rosenbrock function (*cf.* equation 8.8) is so fast to compute that parallelization using the approach presented in section 6 brings a slowdown rather than a speedup (*cf.* Figure 8.12(a) that shows speedups lower than one).

$$R(x) = \sum_{i=1}^{dim} [(1 - x_i)^2 + 100(x_{x+1} - x_i^2)^2] \tag{8.8}$$

Figure 8.12(b) shows the speedup obtained by the DISPAR-GPU algorithm on the same problem. Here, the speedup reaches values as high as 200x using a relatively large population and a

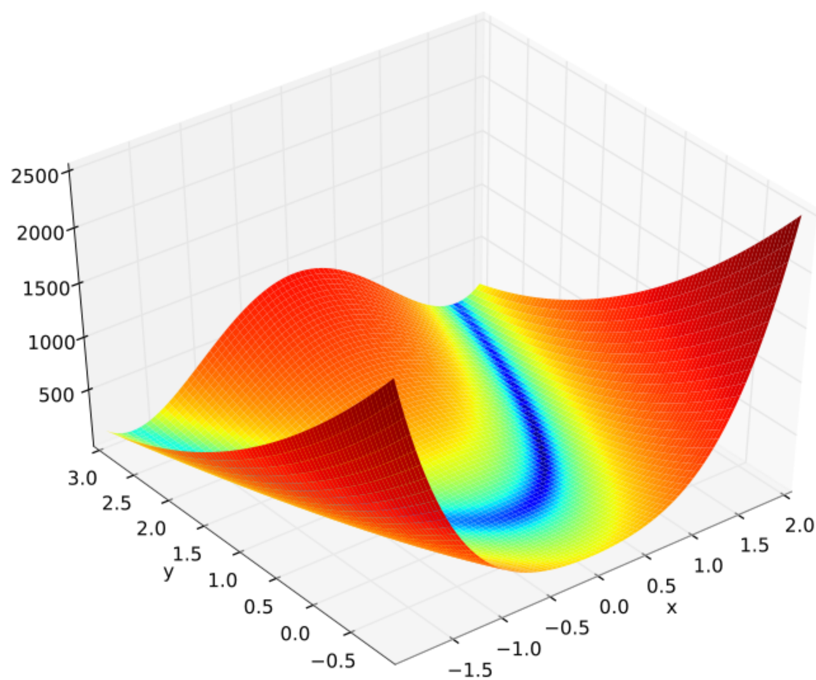
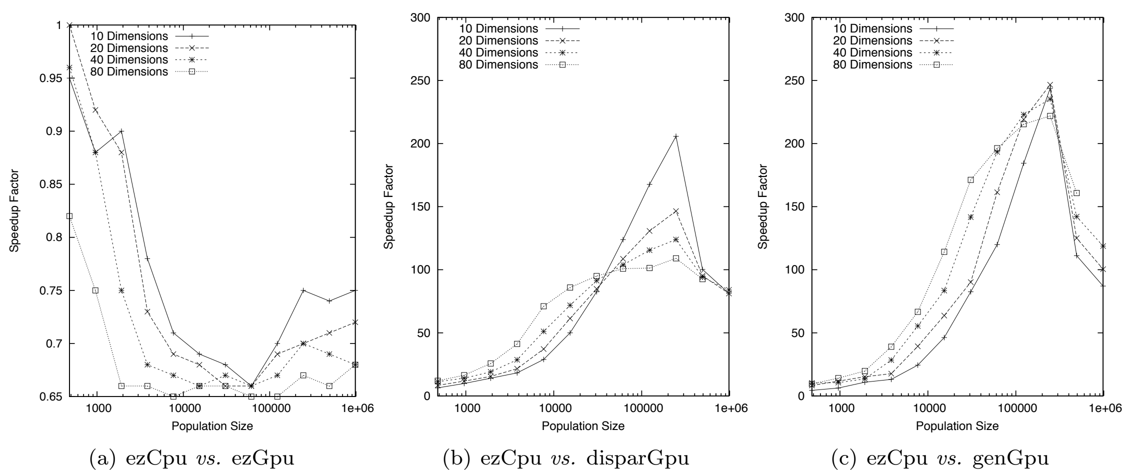


Figure 8.11: Rosenbrock function.



(a) ezCpu *vs.* ezGpu (b) ezCpu *vs.* disparGpu (c) ezCpu *vs.* genGpu

Figure 8.12: Speedup factor for Rosenbrock problem.

small size problem (genome size 10, and 245.000 individuals). The speedup is still very interesting for reasonable population sizes, for all problem sizes.

Then, according to what could be expected, the genGPU implementation (*cf.* Figure 8.12(b)) is faster than the previous two. Even the large size problems get results close to the best. The speedup reaches 250× with this implementation, for the same population size as previously, on the same problem.

Above these sizes, the speedups drop, for both implementations, probably because of conflicting accesses to the MPs caches.

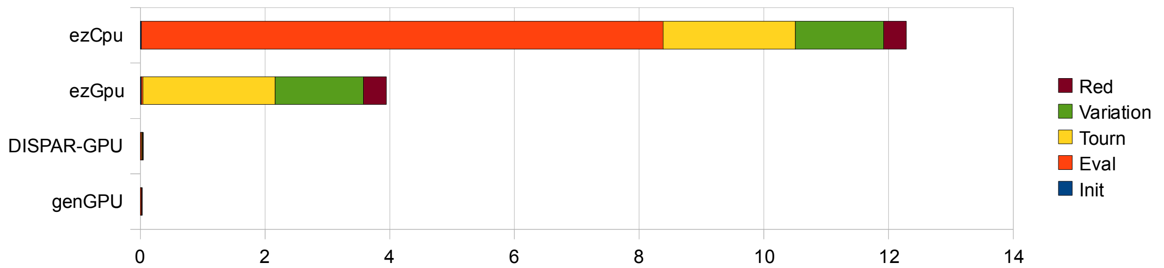


Figure 8.13: Computing time distribution in presented algorithms (s).

8.5.2 Timing distribution analysis

We have seen that the speedup of the parallelization of the evaluation step only was limited by the sequential part of the algorithm and that both complete-GPGPGU implementations show interesting speedups even on very light problems such as Rosenbrock. Figures 8.13 and 8.14 show the distribution of computing time between the different phases of the classical evolutionary algorithm for different implementations, in seconds and percentage of the total computing time. The problem used here is the determination of a minimal multidimensional Weierstrass function, using a number of iterations of 10, *i.e.* a relatively lightweight evaluation function, although much more computation intensive than Rosenbrock.

Figure 8.13 shows that compared to ezCPU, parallelization of the evaluation function by ezGPU virtually suppresses evaluation time. Only the sequential parts of the algorithm remain. Because in ezCPU, parallelizable evaluation time was about 60% of the total time, the obtained speedup is finally of only about 3.

However, one sees that DISPAR-GPU and genGPU manage to parallelize all the steps of the algorithm, virtually nullifying the total execution time in this example (respectively 0.05s and 0.03s, *i.e.* speedups of ×259 and ×380).

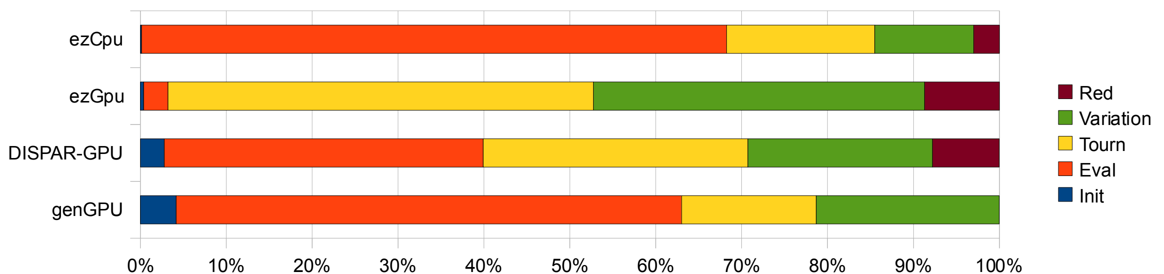


Figure 8.14: Computing time distribution in presented algorithms (% of execution time).

Because it is very difficult to see on the graphs the time used by the respective parts of the algorithm on the fully parallelized algorithms, Figure 8.14 shows all the parts of the algorithm in percentage of the total time.

On this figure, one can see that ezGPU reduces to less than 5% the evaluation time of this lightweight Weierstrass function (therefore increasing the relative percentage of the other parts of the algorithm).

On DISPAR-GPU, evaluation percentage is lower than on the original ezCPU line, meaning that parallelization of the evolution engine is not as efficient as the parallelization of the evaluation function, even if it resulted in a $\times 259$ speedup.

However, on genGPU, it is very nice to see that the proportions of the different parts are roughly identical to those of ezCPU, meaning that all parts of the algorithm have been equally well parallelized (except for population reduction, that disappears in genGPU).

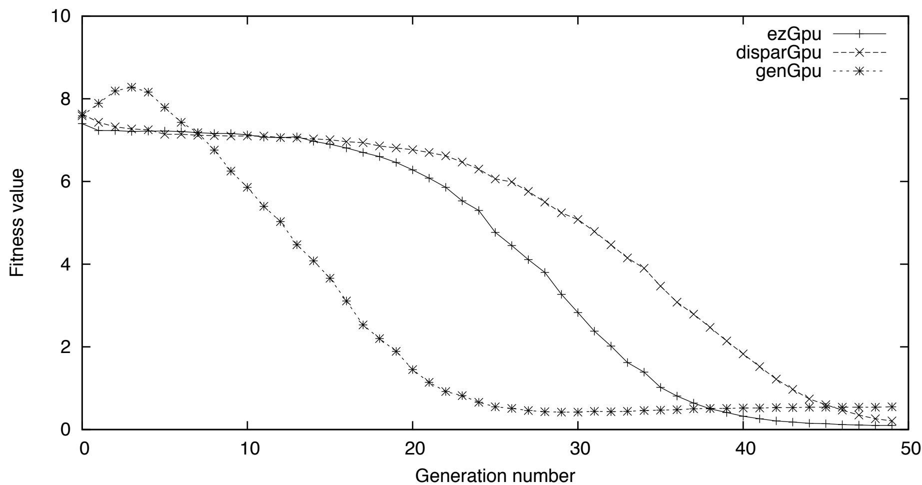


Figure 8.15: Evolution of the best individual for the three GPGPU parallel algorithms (average on 20 runs).

8.5.3 Qualitative behaviour of the proposed parallel algorithms on the Weierstrass function

Completely parallelized algorithms (compared to a parallelization of the evaluation function only) provide advantages in terms of computation time, but also modify the general behaviour of the algorithm because they impose algorithmic changes, related to the underlying hardware constraints.

Nevertheless, DISPAR-Tournament manages to roughly mimic a typical ES tournament reduction operator and GenGPU implements a classical generational algorithm. If the differences with standard algorithms are only slight, this means that the published qualitative results obtained with the standard algorithms should transfer to the GPU parallel implementation of these same algorithms, but using only a fraction of the time of the sequential algorithm.

Figure 8.15 shows the evolution of the best individual over generations for the three considered algorithms on the 10 dimension Weierstrass problem, using 10 iterations only, for a (1000,1000) population with generational replacement for genGPU, 4-tournament for ezGPU, and a $4 \rightarrow 2$ multi-DISPAR tournament for DISPAR-GPU.

Obviously, genGPU shows a widely different behaviour compared to the two others. The lack of elitism sometimes creates a negative evolution of the best individual in the beginning of the run,

but then, a much better convergence towards good results than the two other algorithms. However, towards the end, genGPU fails to find really good solutions, probably again due to the lack of elitism.

ezGPU and DISPAR-GPU show a comparable evolution of the best individual fitness (the difference in convergence rate coming from the fact that the two reduction operators are not exactly qualitatively identical).

Chapter 9

Modern version of EASEA

Contents

9.1 Diffusion of research results using EASEA 108

9.2 Internal structure of EASEA 109

9.2.1 Code generator and templates 109

9.2.2 LibEASEA 111

9.3 Island Model 112

The old EASEA platform developed by Pierre Collet presented in section 2 was chosen as a medium for disseminating the algorithms elaborated in this PhD thesis. As such, a thorough rewrite of this platform was therefore performed so that it could produce parallel code to exploit GPGPU cards.

The effort required to porting the developed algorithms into this new EASEA platform so that other researchers and scientists could use them in a transparent way on possibly already existing *.ez* programs was an important part of this PhD thesis.

This new version includes many new features and removed some others that were considered as obsolete. All these changes were directed by the wish to turn EASEA into an evolutionary algorithm massive parallelization platform.

The version of EASEA developed for this PhD makes it possible to efficiently use massively parallel machines equipped with one or several GPGPU cards for the execution of parallel EAs.

EASEA is used to receive all the new evolutionary algorithms developed by the students and collaborators of the BFO team, hence the importance of the time spent in porting all developments into the EASEA platform. As a result, the island model currently developed by another PhD student (Frederic Krüger) can immediately benefit from the parallelisation of EA algorithms on GPGPU. This work (shortly described in section 9.3 as it extends the presented work) now allows EASEA to exploit:

- the different cores or processors of a single machine,
- different GPGPU cards as different islands,
- clusters of sequential machines,
- clusters of multi-cores machines,
- clusters of GPGPU machines.

Combining the island model to GPGPU parallelization has lead to the discovery of 50 possible zeolite structures among which one corresponded to a real one (cf. section 10.3.2).

9.1 Diffusion of research results using EASEA

Evolutionary algorithms are not new. They are around since the very beginning of Computer Science with papers referring to an implementation of artificial evolution processes back in the late 1950's. Since then, thousands of papers have been published on the topic.

Nowadays, many algorithms are developed, that are adapted to current hardware or that use recent theory to better exploit knowledge and hardware developments to optimize more and more difficult problems. However, most potential users of artificial evolution (*i.e.* applied researchers from other disciplines) do not seem to be aware about them and keep implementing older EAs, as described by Holland *et al.* in the 1970s. A reason may be that these early algorithms are simple to understand (Simple GA) and therefore much simpler to implement than later developments such as CMA-ES or distributed EAs with an island model even on a single processor architecture.

As a consequence of these awkward experimentations by potential users on really interesting and challenging problems, EAs do not have a good reputation and are thought to behave poorly.

However, when new algorithms work well, they are immediately accepted by users, and NSGA-II [84] is a very good example. It is a modern algorithm, that is used outside the EA community from which it originates and produces interesting results on real-world problems. This creates a virtuous circle as many satisfied users will disseminate their satisfaction, by citing this algorithm in their own publications. The availability of a public, documented and easily downloadable code is probably a key point in this situation, as well as obviously the intrinsic quality of the algorithm.

This should certainly convince researchers to make their work available in a format that is as easy to use as possible.

As this work tries to show, EAs can be very efficiently ported on very common GPGPU cards (nearly all modern computers now have a graphics card), but an efficient implementation is very difficult to achieve due to the complexity of programming these cards.

As a consequence, there is a major risk that the same happens here as with original GAs: people trying to implement EAs on graphics cards will find it very difficult, and will very likely be disappointed by the results, unless they spend several years to understand finely how these cards work, as was the case in this PhD.

Many papers currently describe how to parallelize EAs on GPGPUs, but only very few authors make their code freely available online and as the implementation can be tricky and complex, it is very likely that applied scientists that are not expert programmers or who do not know how parallel hardware works will not understand how to exploit the published code to solve their problems.

The EASEA platform aims at allowing applied scientists to use parallel evolutionary computation to solve their real-world problems. As the source code produced is human-readable, it can also serve as a basis for other researchers to understand and exploit what has been developed in this PhD as a deep reading of this thesis and related papers should make it possible for other researchers to reproduce the presented results and implement the presented algorithms to optimise their problems.

One of the outcomes of this effort to make available the developed algorithm into an open platform is that a 389K Euro Emergence ANR project has been funded by the French ministry of Research to develop an industrial grade version of EASEA to run on grids and clouds of computers (EASEA-CLOUD project). Another outcome is that the EASEA platform starts to be recognized as a massively parallel problem solver that is now participating in much larger research projects applications for grants (RAPSODY 12M Euros Equipex project).

If the EASEA and future EASEA-CLOUD platforms are successful, they could participate in the diffusion of evolutionary computation technique to solve very large inverse problems and serve to optimise complex systems, for instance (this is one of the primary aims of the RAPSODY project).

All these considerations make it important to diffuse research on evolutionary algorithms through a platform such as EASEA, that was an already existing project in our laboratory (the first publi-

cation on EASEA dates back to 2000 [24]. It was therefore natural to use it for this purpose.

A good part of this thesis was spent in modernising, and re-orienting the platform towards the creation of parallel code for GPGPUs, as all developed algorithms had to be implemented in this framework. Some of these implementations had more impact on the global EASEA architecture than others.

Last, but not least, diffusion of source codes allows for reproductibility of research. Using the automatic adaptation mechanism included into the framework, a *.ez* file can be exchanged between EA researchers even if they work on different hardware platforms.

9.2 Internal structure of EASEA

EASEA provides a unified view of evolutionary algorithms. In addition, it allows the use of GPGPU cards for the parallelization of such algorithms. Finally, this software includes a code generation layer, which allows to reduce the amount of code that the user should write in order to implement his own EA.

These features are reflected in its structure, which is similar by many point to what it was originally. Indeed, the parsing layer still exists, as well as the code generation one. This part has remained very similar, especially for backward compatibility with previous users' code. Yet, both parts have been adapted to new needs, particularly to the generation of GPGPU code.

The library part has undergone significant changes. The original EASEA (before 2008), produced code for two external libraries, Galib (a *C++* Library of Genetic Algorithm, presented in Wall [85]) and EO (Evolving Object, introduced more recently in Keijzer et al. [86]). The use of these libraries in EASEA was abandoned, thanks to the development of an internal library: LIBEASEA.

Some of these changes were carried out to help software maintenance. The development of new modules also needed simplifications. The use of one single library serves this purpose.

Nevertheless, the underlying principles of this program have remained the same. The generated code is still *C++* and human-readable and writable. However, since a lot of the code has been outsourced into the library, the generated code is smaller. The library is distributed with the project, it remains editable and searchable. Internal documentation is becoming available in the library code. This means that new EASEA supports *.ez* source files from older versions (example file such as onemax.ez and listsort.ez are virtually unchanged since years 2000).

The following sections detail the different components of EASEA, highlighting the changes and upgrades that have been applied to it in the course of this thesis.

9.2.1 Code generator and templates

As stated in the previous section, the code generation has remained close to what it was before the beginning of this thesis. This workflow allows the generation of code to support the old user files, developed for older versions of EASEA.

The main changes to code generation have been applied to support GPGPUs, that impose some additional constraints for writing the code of a user *.ez* file targeted at GPGPU parallelization. The burden of parallelizing the EA on the GPGPU card is left to EASEA and its code generation part. The code of the evaluation function is modified to use the global population that is exported by EASEA to the GPGPU card.

The new version of EASEA generates an evaluator GPGPU program for the population. This evaluator is written into a CUDA *.cu* file which can be compiled by *nvcc* (NVidia C Compiler), to generate a *.ptx* file containing the code transformed in a portable assembly code, file that will loaded on the card.

These operations are guided, as in the original version of EASEA, by a human-written template file (*.tpl*). This file contains tags that indicate the actions to be run by the generator.

Different actions are possible: the first action is to choose an output file, then the code generator copies the contents of the *.tpl* file to the output file, until the appearance of a new tag, which allows for example to jump in the user's specification file *.ez*, whence problem specific parts are sourced. According to the tag, the text copied can be processed and modified on-the-fly, before being written to the output file.

The main output file contains the code of the individual. This individual is an implementation of the abstract individual that libEASEA manipulates. The concrete individual can be created with the help of the description file of the user, which determines the specifics of an individual designed to solve the problem under consideration.

Two other files are generated, one contains the other elements specific to the user's algorithm (settings). These parameters include those present in the former version of EASEA, but some new have appeared, such as those for setting up the distributed EA island model. The last file contains the user's main function, which is reduced to the essential part, but is still present to allow a customized launch. For example it is possible to build an EA produced by EASEA into another program.

The original EASEA had three main template files, two of them for the *C++* libraries (EO and Galib), and one for generating *Java* code designed for the European DREAM (Distributed Resource Evolutionary Algorithm Machine). Because this project is not supported anymore, its template has been abandoned, as well as templates for the other two libraries. The modern version supports only its internal library, but uses yet more template files. Each one of them is related to an EA variant (for CPU, GPGPU, for different types GA/ES, for CMA-ES, for memetic and genetic programming algorithms). Some template files are also developed through external collaborations, as for example a template file for Differential Evolution and Parallel Differential Evolution (developed with Prof. Jaroslaw Arabas of Warsaw University), a template for the Predator-Prey paradigm (developped with Prof. Jim Smith of the University of West of England). In the latest beta version, the two templates describing CPU and GPGPU algorithms have been merged into a single one, meaning that the choice of the architecture is made at compile time and does not add any overhead at runtime.

This merge was carried out with several goals. First is to have only one template, which allows for an easier maintenance and future extension of EASEA, as EASEA will soon need to support GRID computing and Cloud parallelization (the EASEA-CLOUD project will start in April 2011). In the other hand, Evolutionary algorithms produced for CPU or GPGPU are now created out of an identical code. The algorithms produced are similar in all aspects, except for the evaluation phase, which is realised using a master-slave model on GPGPU. This last point is interesting as it allows to better compare between the two implementations.

The produced code is identical for CPU or GPGPU. A makefile is automatically generated. If the `EZ_GPGPU` environment variable is set to 1, `make` will automatically generate files with parallel CUDA code, and compile them with *nvcc*.

Code generation automatically generates code from the user's specifications for some problem-specific functions of the algorithm. In particular, analysis of the genome allows the creation of serialization functions, as the genome declaration is fully parsed and analyzed. Thus, the functions for the transfer of individuals between islands will be automatically created, as well as functions to dump the whole population and reload it later on for future restarts.

Following this same model, it is possible to generate default crossover and mutation functions, if the user has not provided any of them (the CMA-ES template even ignores whatever crossover or mutation functions that are present in the *.ez* file, and forces a CMA-ES mutation). These variation operators use the default functions defined in the LibEASEA.

9.2.2 LibEASEA

The new version of EASEA therefore uses an internal library, abandoning the use of the external libraries. It is distributed with the EASEA software and is statically compiled with the produced executable.

LibEASEA uses a highly customizable evolutionary loop. shown in Figure 9.1.

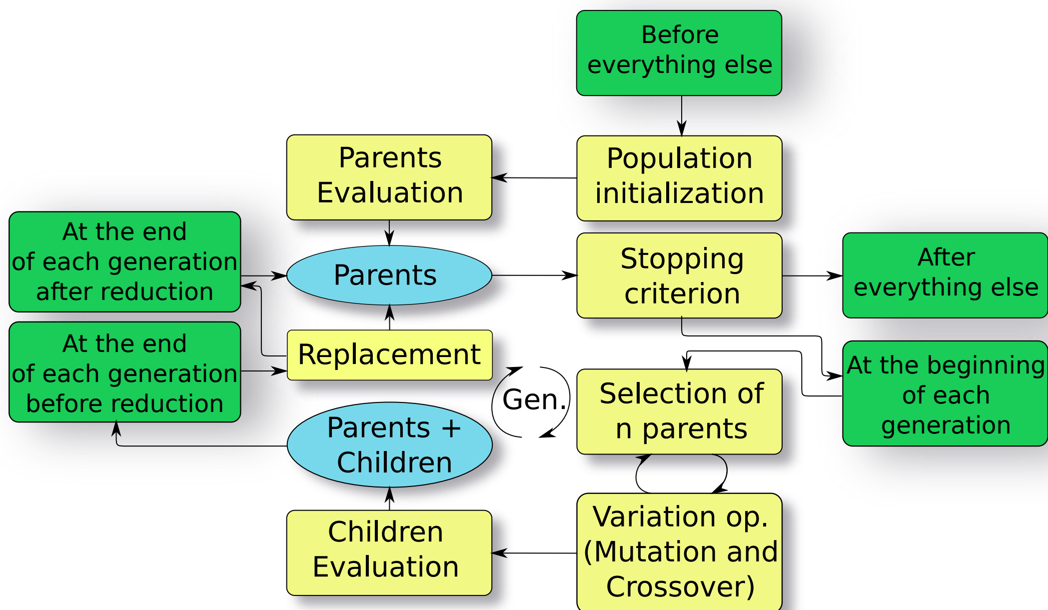


Figure 9.1: Overview of the complete LibEASEA internal loop.

During evolution, this loop allows to call 5 different functions written by the user. Each of these functions is related to a section in the EASEA specification file, where the user can manipulate the genetic algorithm, for example by changing its parameters.

The latest version of libEASEA allows the instantiation of an evolutionary algorithm by extending the abstract individual manipulated by the algorithm. This step is allowed by the moderate use of *C++* templates, especially for the evolutionary algorithm. Writing a main function is also needed, which allows the implementation of a standalone EA or one which is included within a broader application. Finally, a parameter file must be provided to describe the specificity of the user implementation.

The libEASEA manages a set of GPGPUs for the population evaluation. This management is introduced in another *C++* template, which generates a concrete class at compile time. GPGPU card management class allows to automate the management of the GPGPU cards, the load distribution of the binary evaluation of the population individuals on the GPGPUs, as well as the to and fro migration of the population between CPU and GPU memory (two-way for the implementation of memetic algorithms).

In addition, the library implements a multi-threaded loop, for sending the population, launching the evaluation on the GPU cards and retrieval of results (see Figure 9.2).

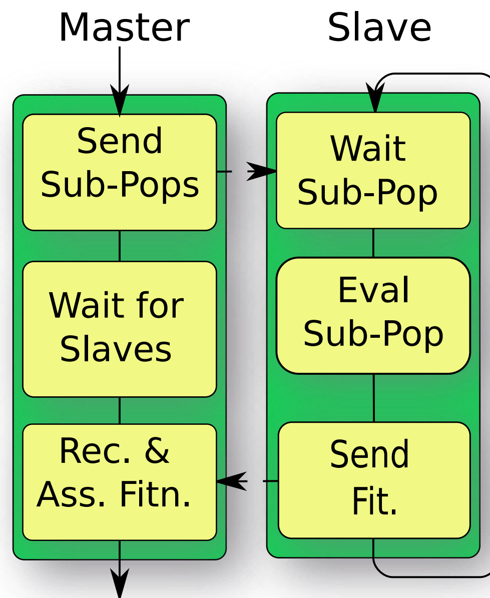


Figure 9.2: Recall from Figure 3.3 in section 3.3.1: evaluation in master-slave model.

9.3 Island Model

This short section describes very briefly the island model implemented by Frederic Krüger (and Pascal Comte) on top of the new EASEA platform that was used to obtain the results presented in section 10.3.2. It allows to couple n machines in a scalable way to get them to work on a single problem and was published in Krüger et al. [43].

We saw in section 3.4 that many studies illustrate the principles and uses of distributed evolutionary algorithm. However, only the 2003 DREAM version of EASEA did implement such functionalities (the aim of the Distributed Resource Evolutionary Algorithm Machine [26] was to run a distributed evolutionary algorithm on many (up to 100 000) machines on the internet) but this functionality was not integrated back into the 2003 C++ version of EASEA. The island algorithm possesses the great advantage that it is a loosely coupled model. This feature allows to scale a run on a large number of possibly geographically distant computing nodes, connected over the internet.

Using one or several GPGPUs for the evaluation of the local population, it makes it possible to mix a distributed EA and a master-slave model to create a system which allows the automatic generation of individual migration functions while being portable to several architectures (Windows, Mac, Linux, with or without GPU cards) in order to create clusters of heterogeneous machines, or multiple clusters of homogeneous or heterogeneous machines.

The loosely coupled communication allows an implementation based on asynchronous communication methods using UDP (User Datagram Protocol) rather than TCP. This protocol does not provide service guarantees, so it is possible for packets (= migrated individuals) to get lost, damaged or duplicated, but the beauty of a stochastic algorithm like EA is that it is totally robust to these kind of random inputs.

The asynchrony allows independence of the various islands. The failure of a node does not stop the overall algorithm and the difference in execution speed between several nodes does not

penalize the performance of the fastest. Also the lack of synchronization, (an expensive feature on distributed architectures), improves the scaling ability and accelerates the execution of the global algorithm.

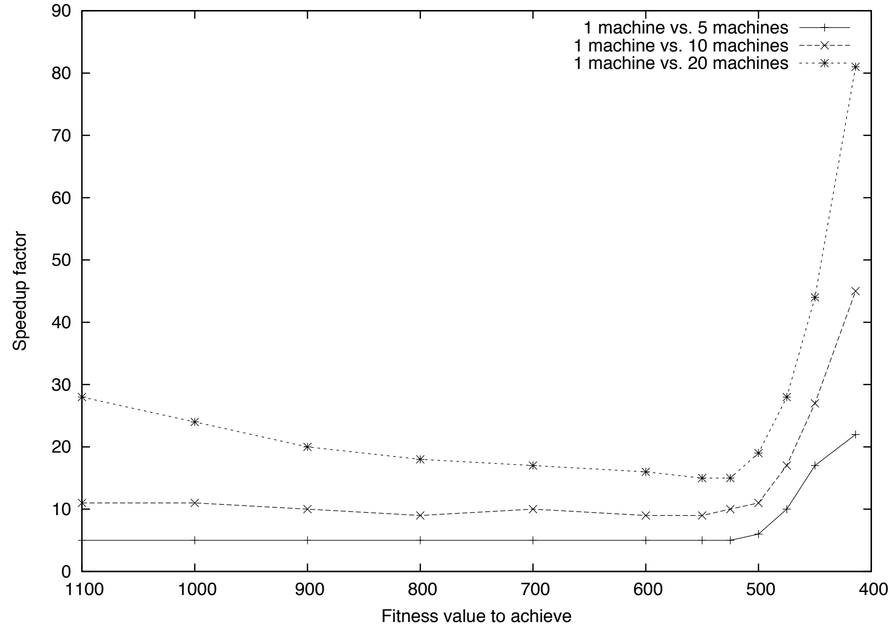


Figure 9.3: Speedup factor achieved with dEA on weierstrass benchmark.

Using this particular implementation, it is possible to obtain supra-linear speedup compared to a panmictic implementation [87]. In order to evaluate the EASEA implementation of distributed EAs, the weierstrass benchmark function is tested on clusters of 1, 5, 10 and 20 GPGPU machines.

A goal-fitness speedup measure is used, by comparing the needed time for each configuration to attain particular values (1100, 1000, down to 400) of a 1000 dimensions and 120 iterations Weierstrass benchmark with an Hölder coefficient set to 0.35. The implementations use respectively 81920, 16384, 8192 and 4096 individuals, for 1, 5 ,10 and 20 nodes so that all in all the global number of individuals is identical for each configuration.

Figure 9.3 shows the relative speedups *w.r.t.* the goal fitness, knowing that each node use a GTX275 (including the reference one), that already show an approximate speedup of 150 compared to the CPU implementation, as shown in Figure 11.2.

On this figure, it is interesting to note that until the problem gets really difficult for panmictic populations of 81920 individuals around value 500, the obtained speedup is roughly linear with the number of machines. Then, beyond 500, speedup rises until value 430, which is the lowest common value found over 20 single machine runs.

This value is reached slightly above 80× faster for 20 machines, slightly above 40× faster for 10 machines, and slightly above 20× for 5 machines. When a supra-linear speedup is observed, the linear speedup between the n machines of the three different island models is respected.

Chapter 10

EA applications

Contents

10.1 Data-driven multi-objective analysis of manganese leaching 115
10.1.1 Presentation of the problem 115
10.1.2 Results 117
10.2 Propositionalisation using EA 119
10.2.1 Introduction 119
10.2.2 Problem presentation 119
10.2.3 Evolutionary Algorithm description 120
10.2.4 Experiments 122
10.2.5 Conclusion 124
10.3 Zeolite structure optimisation 125
10.3.1 Problem description 125
10.3.2 EAs for zeolite structure optimization 126
10.3.3 Speedup on zeolite problem 126
10.3.4 The zeolite search continues 127

This chapter contains sections referring to published papers on different problems on which I worked to gain experience on EAs and get a feeling on how to use them to solve real-world problems.

10.1 Data-driven multi-objective analysis of manganese leaching from low grade sources using genetic algorithms, genetic programming, and other allied strategies

10.1.1 Presentation of the problem

This work uses different nature-inspired strategies to come up with a manganese leaching model from low grade sources. The used algorithms are neural networks and genetic programming. For both paradigms, a commercial solution and tailored algorithms are compared, in order to produce data-driven models that are subsequently subjected to bi-objective Pareto optimization algorithms, which means that four regression algorithms are used to construct models from data:

EVO_NN is a multi-objective evolutionary algorithm that evolves a neural network topology and internal weights, by optimizing two conflicting objectives, that are the RMSE (Root Mean

Square Error) and the complexity of the network. This software is presented in Pettersson et al. [88]:

MD_ANN is a neural network module implemented into the modeFrontier™software suite.
ECJ_GP is an home-made genetic programming algorithm implemented with ecj library [89].
MD_GP is a genetic programming module implemented into the modeFrontier™software suite.

All the nature-inspired algorithms optimise RMSE between a set of documented experiments, where different parameters are strictly controlled. These parameters are: the initial concentration of reactants, (acid and glucose or lactose or sucrose) temperature of leaching and particle size as presented in Pettersson et al. [90].

Then multi-objective algorithms try, using the data-driven model as an evaluation function, to maximize the percentage of manganese recovery while minimizing quantity of acid needed in the leaching process. Several optimizing multi-objective optimization (MOO) algorithms are used, that are NSGA-II (Non-dominated Sorting Genetic Algorithm-II, [84]), MOGA-II (Multi-Objective Genetic Algorithm-II [91]), MOPSO (multiple objective particle swarm optimization, [92]) and PPGA (Predator-Prey Genetic Algorithm, [88]).

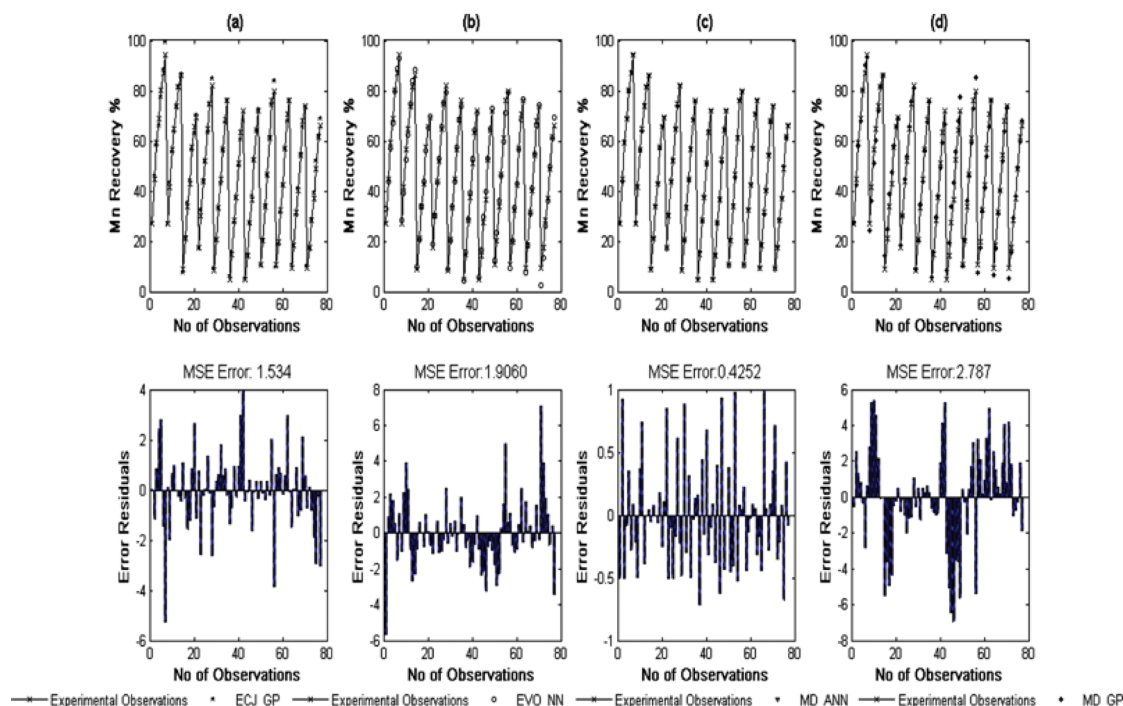


Figure 10.1: Efficacy of carbonate leaching models using various strategies.

They are used in various combinations with the models obtained by the data-driven model algorithms such as :

MD_GP_NSGA2 Genetic programming with NSGAII MOO modules in modeFrontier™.
MD_GP_MOGA2 Genetic programming with MOGAII MOO modules in modeFrontier™.
MD_GP_MOPSO Genetic programming with particule swarm multi-objective optimization.
MD_ANN_NSGA2 Neural Network with NSGAII MOO modules in modeFrontier™.
MD_ANN_MOGA2 Neural Network with MOGAII MOO modules in modeFrontier™.

MD_ANN_MOPSO Neural Network with with particule swarm multi-objective optimization.

ECJ_GP_NSGA2 Genetic programming using ecj and NSGAII MOO algorithm.

ECJ_GP_MOGA2 Genetic programming using ecj and MOGAII MOO algorithm.

ECJ_GP_MOPSO Genetic programming using ecj and MOPSO MOO algorithm.

ECJ_GP_PPGA Genetic programming using ecj and PPGA MOO algorithm.

EVONN_PPGA EVO_NN and MOO PPGA.

10.1.2 Results

Using these different algorithms, the resulting models presented in Figure 10.1 have been found, for carbonate related data. The ascending curves present recovery of manganese *w.r.t* time, one complete experiment after another (one tooth corresponds to one experiment). Vertical lines simply connect the last experiment to the next.

	5 min	15 min	20 min	60 min
MD_GP_NSGA2	Good	Good	Good	Good
MD_GP_MOGA2	Good	Good	Good	Good
MD_GP_MOPSO	Sparse	Sparse	Sparse	Sparse
MD_ANN_NSGA2	Good	Good	Good	Good
MD_ANN_MOGA2	Sparse	Sparse	Sparse	Sparse
MD_ANN_MOPSO	Sparse	Sparse	Good	Good
ECJ_GP_NSGA2	Good	Good	Good	Good
ECJ_GP_MOGA2	Good	Good	Good	Good
ECJ_GP_MOPSO	Poor	Poor	Poor	Poor
ECJ_GP_PPGA	Poor	Poor	Poor	Poor
EVONN_PPGA	Poor	Poor	Sparse	Sparse

Table 10.1: Gradation of various strategies on the basis of density and spread of carbonate leaching Pareto-frontiers

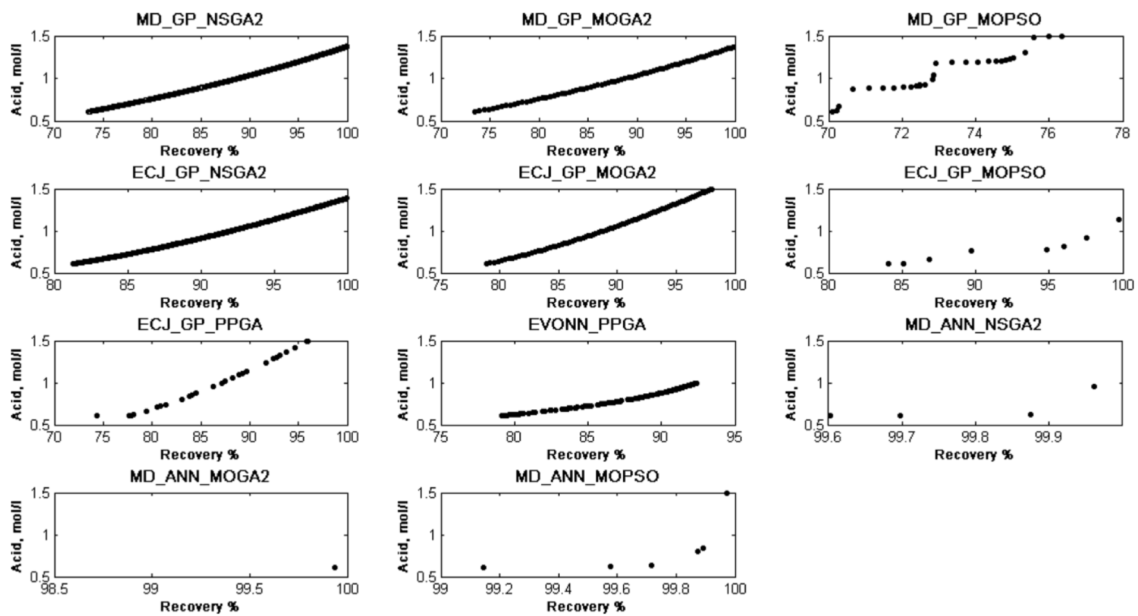
The two ModeFrontier^{TM} modules (neural network and GP) obtain the best and the worst results in RMSE. The first probably shows signs of overfitting, while the latter underfits the data. Finally, the two other strategies remain between these two extremes.

Using the model that produced the data in Figure 10.1, several multi-objective optimization algorithms have been applied. Figure 10.2(a) shows the Pareto-frontiers for carbonate 60 minutes leaching time. In his PhD thesis, Biswas [93] presents the results for the other leaching times. Finally Table 10.1 helps to analyse these results, classified as “Good” “Sparse” or “Poor.”

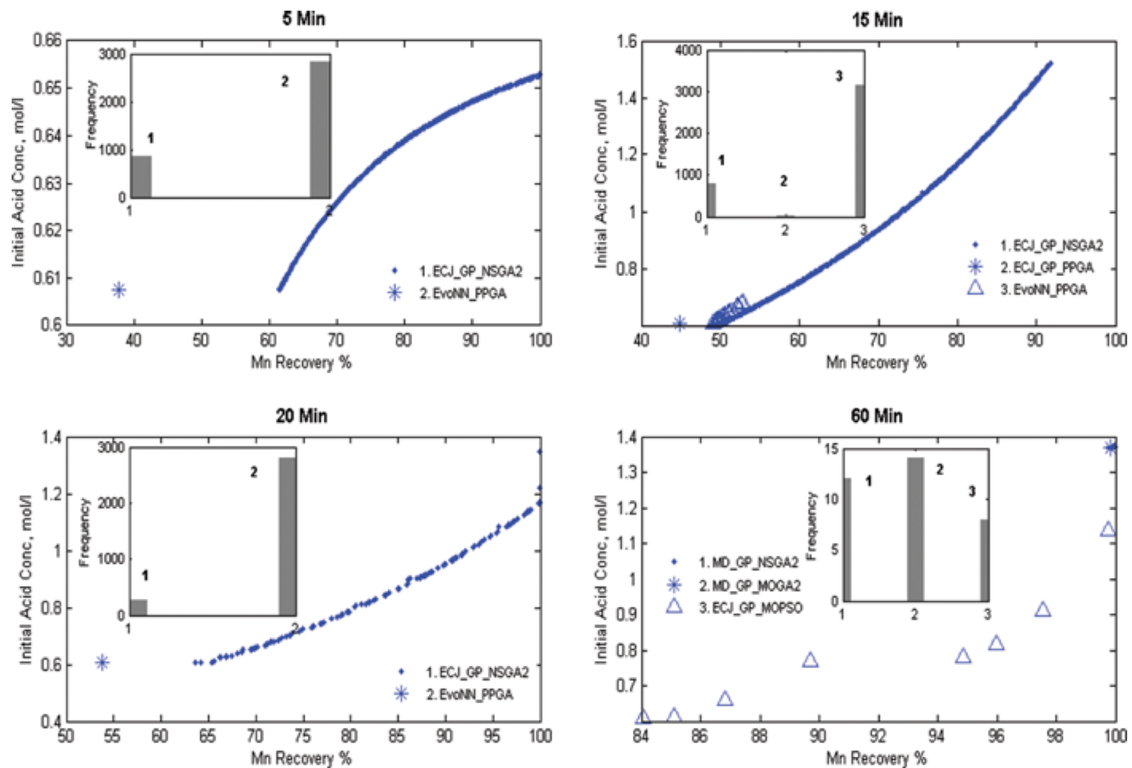
Several interesting combinations of modelling algorithms and multi-objective optimizing algorithms exhibit good performances, as for exemple ECJ_GP with NSGAII or Predator-Prey algorithm Genetic Algorithm.

Using a Fonseca ranking [94], the results from “Good” algorithms are mixed together into ultimate Pareto frontiers, in Figure 10.2(b). The relative contribution of each algorithm to these frontiers are depicted as barcharts, in the same Figure. For this particular leaching, most contributions come from ECJ_GP and EVO_NN. Indeed, this genetic programming algorithm shows a good spread, while EVO_NN brings the majority of the points.

These results are published in Biswas et al. [78], as well as leaching with other additional reactant, as glucose, lactose and sucrose.



(a) Pareto frontiers for carbonate leaching computed for 60 minutes of leaching time.



(b) Pareto frontiers o btained for carbonate leaching by ranking the best solutions of all strategies. Realtive contributions of various strategies are shown as insets

Figure 10.2: Pareto-frontiers for carbonate leaching.

10.2 Propositionalisation as complex aggregates thanks to an evolutionary algorithm

10.2.1 Introduction

A relational database describes data in the form of several tables. Three such examples are molecules, customers and trains. A database of molecules can have a main table describing the molecules, annex tables for the atoms, and bonds between the atoms. A database of customers can have a main table describing the customers, and annex tables for products to purchase. A database of trains can have a main table describing trains and an annex table for cars. Relational data-mining looks for properties of individuals, such as the properties of molecules or customers. Traditional attribute-value data-mining uses only informations from the main table. The specific difficulty of relational data-mining is to use some information from the other tables that have a one to many relationship with the main table [95]. These properties take the form of aggregates such as the average customer purchases, the number of nitrogen atoms, or the existence of a single bond with a fluorine atom.

Search techniques for relational data come primarily from inductive logic programming (ILP). By nature, they generate properties in the form of existential aggregates as "the customer has bought at least one such article" more often than cardinalities such as "the number of times the customer has purchased this article." Approaches that generate only cardinalities generally consider only one condition, such as "the number of times the customer bought a specific kind of wine." We are trying to learn complex aggregates, in other words, the cardinality of a conjunction of conditions, such as "the number of times the customer bought the wine at over 50 euros."

The only work we know on such a generation of complex aggregates suggests an order of aggregate functions [96], but the conjunctions of conditions which apply these functions are generated exhaustively. A high number of conditions severely limits the use of this approach.

We propose to use an evolutionary algorithm to explore the space of conjunctions of conditions. The quality of a set of aggregates is evaluated by the accuracy of a model extracted by a fast attribute-value learning algorithm as J48.

In this work, we consider a relational database table reduced to a principal and an annex table, with one one-to-many relationship. Indeed, the problem of combinatorial explosion is already happening with a single table. In addition, this configuration is common. This is the case between a dimension and a fact table in a star schema data warehouse. Many relational data mining test cases can be reduced to this configuration, such as financial data set in PKDD 1999, when a loan is associated with transactions in the account, or the set of data on atherosclerosis of PKDD 2004, when a patient undergoes several examinations. It is also the case of real data that we test. The cases of chained one-to-many associations are rare, and will not be considered here.

We use as an example the database shown in Figure 10.3 and the concept expressed by the *sql* query in Figure 10.4, indicating that a train is positive, if its power is less than 800 and if it pulls at least two cars, the weight is greater than or equal to 400 and having a length less than or equal to 1600.

A simple aggregate builds a feature based on a single column of an annex table, e.g. "the number of times the customer purchased wine," or "the minimal price of a bottle of wine purchased by the customer." A complex aggregate involves several columns of a subsidiary table.

10.2.2 Problem presentation

Most classical data mining algorithms do not support relational data. They are usually designed to deal with the attributes of an entry, but cannot use the fact that a relational attribute contains multiple values.

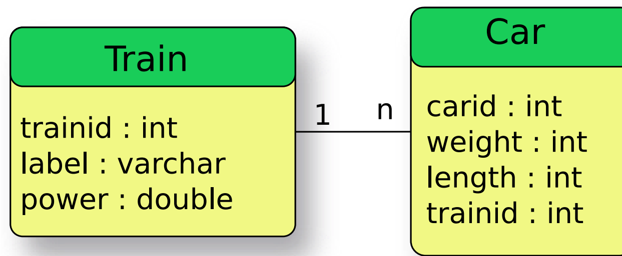


Figure 10.3: An example of a relational model.

```
select train id from train where power < 800 and train id in
   (select train id from car where weight>=400 and length<=1600
    group by train id having count(*)>=2)
```

Figure 10.4: Example of concept based on complex aggregates.

There are several ways to bypass this problem, one of which is to use aggregations. The aggregation can summarize all the values of a relational attribute into a single value. Figure 10.5 presents the use of the *average* function to aggregate the attributes of the relationship between train and car.

Aggregated this way, relational data can be processed by a standard data-mining algorithm. Many aggregation operators can be used together or not.

However, the usefulness of an aggregation operator depends strongly on the problem being processed.

Here, we would like to learn clauses such as: "the city block where the number of buildings the area of which is above 100 and elongation of less than 80 is greater than 5." This clause contains an aggregate function that is the enumeration of the number of buildings that satisfy a combination of conditions.

For all these reasons, we wish to describe aggregates with a more complex language. It must be able to perform aggregations, which are a count of the number of objects of the one-to-many relationship from a conjunction of belonging to an interval, as shown in Figure 10.6.

The feature being generated represents the number of objects belonging to the relationship in question, whose attributes fall into the set of genome intervals. By combining several features generated this way, we hope to help a data-mining algorithm type C4.5 to label the data with more accuracy.

10.2.3 Evolutionary Algorithm description

The features generated can be represented by a set of $2 \times n$ real numbers. A pair of numbers represents an interval on the attribute of the concerned appendix table.

An individual therefore contains $2nm$ real numbers, where n is the number of sub-attributes of the appendix table and m the number of attributes that must be generated. We call a set of $2n$ numbers: a meta-gene, which represents one of the generated features. The genome is also composed of $2nm$ boolean values that allow to deactivate a gene. This corresponds to using a ∞ (resp. $-\infty$) value for the upper bound (resp. lower).

The evaluation of such an individual is done using the genome and the two tables belonging to

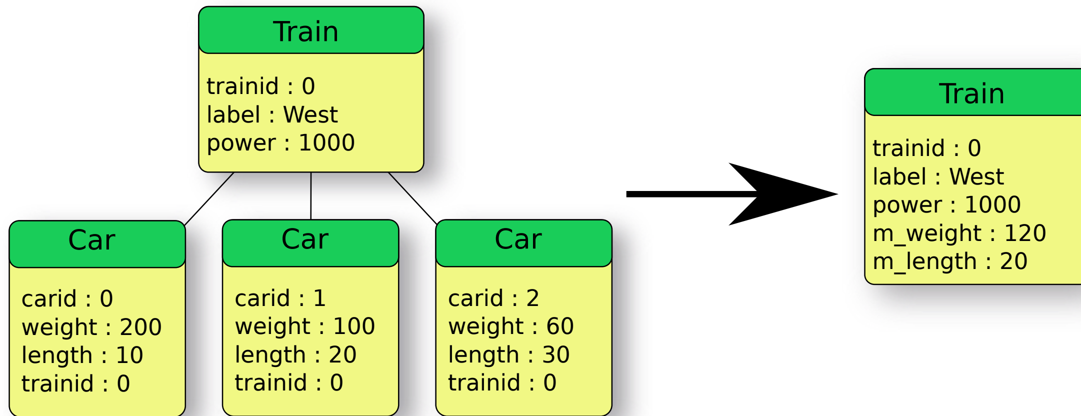


Figure 10.5: An example of aggregation using mean as aggregation operation

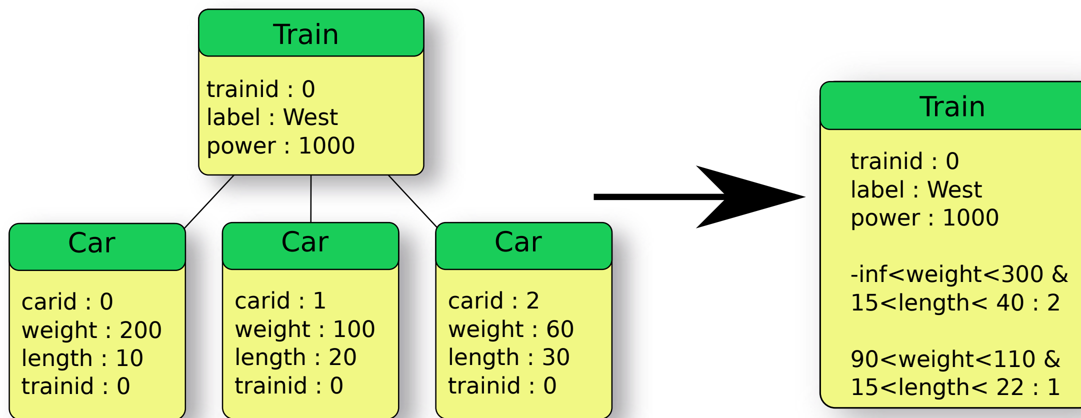


Figure 10.6: An example of aggregation using our language.

the relationship. The evaluation function generates a temporary table containing the attributes of the main table and the m features representing the aggregates from the genome. A C4.5 classifier [97] (its java implementation J48) is applied in cross-validation on the generated table. Finally, the fitness value of the individual is the sum of all the errors on the k test sets of the cross-validation.

The genes are initialized by drawing randomly, with a uniform distribution, values from the corresponding columns in the dataset. Those values are more likely to make sense than an arbitrary value drawn between the minimal and the maximal values, and automatically reflect the distribution of values of each attribute in the dataset. The selected thresholds must satisfy the constraint $L_i < U_i$ in a given number of trials, otherwise one of the thresholds is disabled to make the interval valid.

The variation step uses a uniform crossover operator, that does not respect the structure of the genome, because the feature order is not important, as features represent same kind of object. We used a Gaussian mutation, which allows to increase fitness by gradually shifting a bound of an interval. After the mutation, the value is adjusted on an existing value in the database, with a similar technique that is used during the initialization.

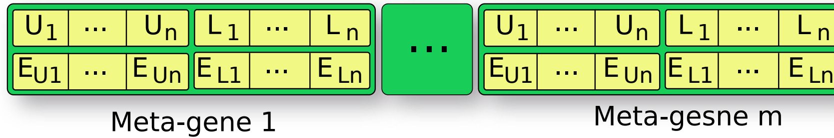


Figure 10.7: Example of a genome of m meta-genes.

Probability of copying a whole meta-gene	0.2
Probability of copying a gene	0.2
Probability of mutating a gene	0.5
Probability of disabling a threshold	0.1
Number of meta-genes per genome (m)	5
Number of folds for cross-validation (k)	5
Kind of elitism	weak
Elitism	1 individual
Replacement stategy	*ES+*
Replacement and selection tournament operator	Tournaments

Table 10.2: General parameters of the evolutionary algorithm.

Finally, the parameters of our algorithm are summarized in table 10.2.

The ECJ library [89] was used here, which simplified the development of an initial prototype. This library is coded in Java, it has been possible to use a part of the Weka software to create and evaluate the decision trees [98]. The Weka implementation of J48 is here regarded as a standard one. In addition, it is thread-safe and allows, in contrast with Quinlan's implementation [97], to use all the cores of a modern desktop machine.

10.2.4 Experiments

Artificial dataset

In order to validate the evolutionary algorithm, we use an artificial problem. For this purpose, a table with the structure shown in Figure 10.3 has been randomly generated and labelled by the same pattern as presented earlier in the SQL command 10.4.

The database contains approximately 5000 cars and 1000 trains, 300 of them have a positive label. The evaluation is done using a cross-validation in 5 folds and the overall algorithm uses a cross-validation in 10 folds.

Figure 10.8 presents the evolution of the best population individual for each of the 10 folds of the cross-validation, on an average of 10 runs using different initialization seeds. The algorithm converges to the optimal solution between the 6th and the 12th generation, whatever being the initialization and the fold partition.

Figure 10.9 shows an example of a decision tree that has been generated by J48 using a table produced by our evolutionary algorithm. One of these nodes uses the feature generated by an evolved meta-gene. The tree matches completely the concept being learned, proving that the algorithm is able to learn this type of clause.

Including the average size of the trees produced by C4.5 in the evaluation function leads to simpler solutions, hopefully increasing the generality of the solutions. However both objectives (reducing the size of the tree and reducing the number of errors) are contradictory in many situations, and defining a weighted sum of those objectives is therefore difficult. It has been possible on

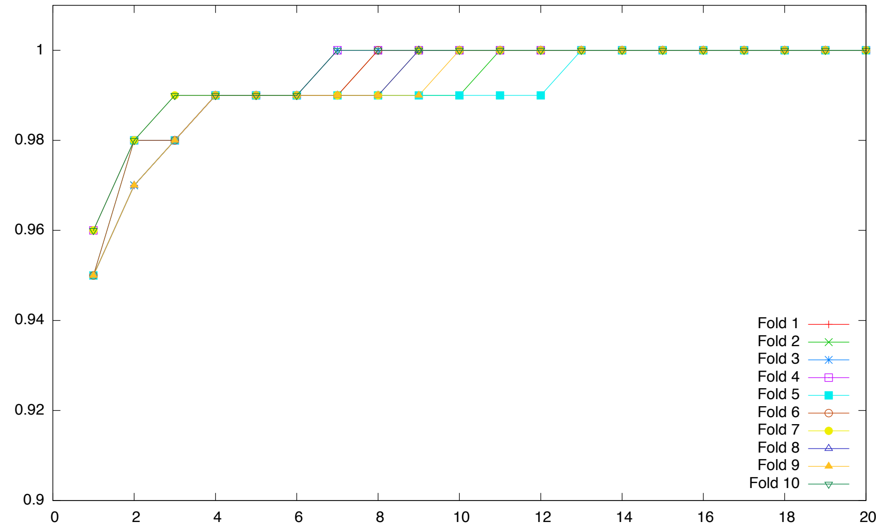


Figure 10.8: Convergence of the evolutionary algorithm on the artificial dataset.

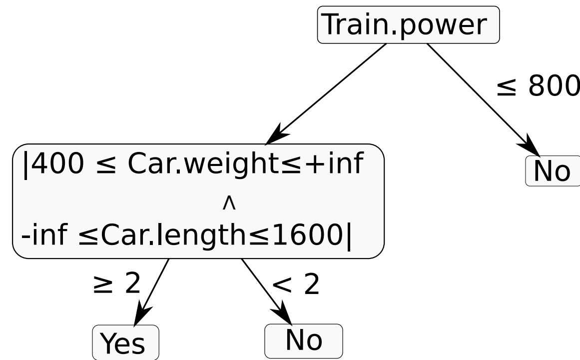


Figure 10.9: Example of decision tree learned as a solution of the concept of figure 10.4.

this artificial dataset to tune the weighting and to significantly increase the number of times the target concept was learned, because the size of the target concept is known. But in the case of real problems, the size and structure of the target concept are not known, which makes difficult to tune the weighting. Therefore, results presented in this article do not take the size of trees into account, for the real dataset as well as for the artificial dataset.

Table 10.3 shows the results of our algorithm, compared to Relaggs and a cardinalisation algorithm. Relaggs is based on the aggregation of the attributes of a relation, by the functions: average, max, min and sum.

Cardinalisation is a technique that allows a kind of data aggregation: for a given attribute, cardinalisation cuts its known definition interval in a given number of upper bounds. For each interval, it gives the number of items that are below this upper bound for all computed thresholds.

As expected, complex aggregates allow to learn a better model than previous approaches, *cf.* table 10.3. Moreover, those aggregates allow to model the original target exactly. The expressivity of the other approaches is less fitted to this kind of hypotheses, thus they are forced to combine more conditions to get accurate but larger models.

	Relaggs	Cardinalisation	Cplx. Aggr.
Accuracy	96.8%	92.2%	**99.6%**
Number of nodes in the decision tree	31	7	**5**

Table 10.3: Results on the artificial dataset.

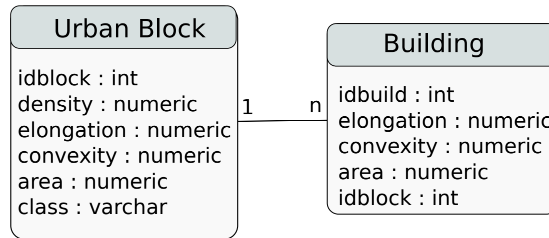


Figure 10.10: The relational model of our geographic databases.

Geographical datasets

In this section we consider several geographical tables. These tables describe urban blocks with domain related attributes (building density, elongation, convexity, area) and a label among six classes (individual houses, housing blocks, mixed housing, specialized high density area, specialized low density area, and mixed area). Each city block is related to a set of building, also described by their geometric properties (elongation, convexity and area). Figure 10.10 shows the schematic of this relational database. The three areas used in these tests share this diagram, but describe different geographic areas.

In these tests, we also use 10 folds for the cross-validation as before and compare with the Relaggs and the cardinalisation algorithm.

	Relaggs	Cardinalisation	Cplx. Aggr.
Zone1	**93.2%**	91.9%	91.5%
Zone2	93.1%	93.1%	**94.5%**
Zone3	91.0%	**94.3%**	91.7%

Table 10.4: Accuracy on the geographical datasets.

Here, the results are less encouraging than for the test on artificial data. No language bias does show a clear advantage over the others. In addition, the algorithms show signs of over-fitting, with very high precision scores. Our technique provides all in all a new and complementary technique of propositionalisation.

10.2.5 Conclusion

We presented a propositionalisation technique that can generate complex aggregates, despite the combinatorial number of those complex aggregates, thanks to an evolutionary algorithm optimizing the accuracy of an attribute-value learner. Its efficiency has been checked on an artificial dataset, showing that our approach allows to learn more expressive, therefore smaller and more accurate models than other existing techniques, still using a reasonable amount of time and memory. The comparison on real datasets shows that our approach indeed offers a different expressivity than existing techniques, and gets a better accuracy on some problem. Thus it complements the diversity

of the available propositionalisation techniques.

Two contradictory objectives (reducing the size and the number of errors) were identified. The use of a multi-objective optimization algorithm such as [99, 100] is a perspective for future work in order to balance both objectives in order to get more accurate and more comprehensible models. Another perspective is to replace the sequence of m meta-genes by a Parisian Approach [101]. Some meta-genes would be gathered during the evaluation to be used as the input of C4.5. Then, each meta-gene would get its own mark, proportional to its selection in the decision trees built by C4.5 during the cross-validation. Moreover, other metrics such as F-measure could be considered to evaluate an individual, and other classification algorithms than C4.5, *e.g.* a naive Bayesian learner could be used.

This work does not use GPGPU. The evaluation of an individual includes the construction of a temporary table and the execution of a C4.5 algorithm on it.

Two cases were considered for the parallelization of this algorithm on GPGPU, neither of them seeming feasible or interesting in terms of performance. For this problem, using a master-slave parallelization, as described in Section 6 would required a large amount of memory to allow the evaluation of a large number of individuals in parallel, mainly because the evaluation needs to create a temporary table for each individual. However, an evaluation is relatively heavy and it could be parallelised. But the algorithm that creates decision trees does not really seem parallelisable on a SIMD architecture.

Finally, the use of a parallelised algorithm on a multi-core processors (4 cores), and Java, was preferred for its ease of implantation.

10.3 Zeolite structure optimisation

10.3.1 Problem description

In materials science, knowledge of the structure at an atomistic/molecular level is required for any advanced understanding of its performance, due to the intrinsic link between the structure of a material and its useful properties. It is therefore essential that methods to study structures be developed.

Rietveld refinement techniques [102] can be used to extract structural details from an X-Ray powder Diffraction (XRD) pattern [103, 104, 105], provided an approximate structure is known. However, if a structural model is not available, its determination from powder diffraction data is a non-trivial problem. The structural information contained in the diffracted intensities is obscured by systematic or accidental overlap of reflections in the powder pattern.

As a consequence, the application of structure determination techniques which are very successful for single crystal data (primarily direct methods) is, in general, limited to simple structures (*cf.* fig.10.11). Here, we focus on inherently complex structures of a special type of crystalline materials whose periodic structure is a 4-connected 3 dimensional net such as alumino-silicates, silico-alumino-phosphates (SAPO), alumino-phosphates (AlPO), etc... These materials are microporous materials, whose structure allows to sort molecules based on a size exclusion process due to the presence of channels (fig. 10.11-c) and cages (fig. 10.11-b), as shown in fig. 10.11 for the framework called LTA [106]. The picture only shows a finite part of the structure as crystals are 3D periodic, i.e. the unit cell in white, (fig. 10.11-left), is repeated by simple translation in all the dimensions, (fig. 10.11-f) containing 27, e.g. 3*3*3, unit cells.

The determination of such kinds of structures is still very much dominated by model building.

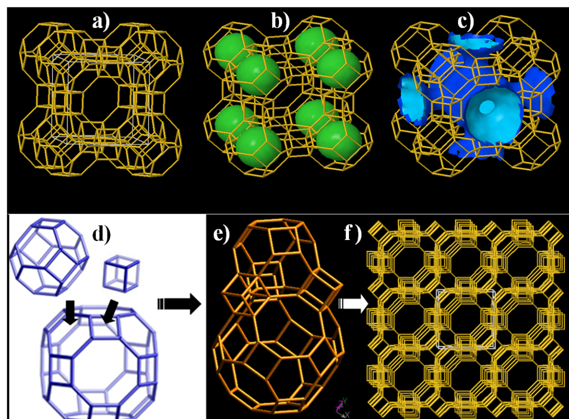


Figure 10.11: LTA crystal framework: a) the white cube is the unit cell, b) cages are represented in green, c) 3D channels are in blue, d) split of the structure LTA in building units, e) piece of LTA structure, and f) crystal structure with 27 unit cells.

10.3.2 EAs for zeolite structure optimization

The fitness function used for evaluation is based on the connectivity of atoms. As such materials are characterized by networks of corner-sharing TO4, with T a given tetra-coordinated element such as Si, Al, P but also Ge, a potential structure must fulfill this structural constraint. The genetic algorithm is employed in order to find "correct" locations, e.g. from a connectivity point of view, of the T atoms. As the distance T-T for bonded atoms, lies in a fixed range [dmin-dmax], the connectivity of each new configurations of T atoms can be evaluated. The fitness function corresponds to the number of defects in the structure, and Fitness=f1+f2 is defined as follows:

1. All T atoms should be linked to 4 and only 4 neighbouring Ts, so:

   f1=Abs(4-Number_of_Neighbours), and
2. no T should be too close, e.g. T-T<dmin, so:

   f2=Number_of_Too_Close_T_Atoms.

This very terse fitness function is used to quickly find candidates which are then submitted to a second optimization/evaluation algorithm.

10.3.3 Speedup on zeolite problem

The evaluation contains a step where the genotypic representation is transformed into a phenotypique one, that can be evaluated. Indeed, a structure should be constructed using only the unit cell, being contained into the individual's genome. After the structure is constructed, the evaluation function can be applied. These two phases are implemented on GPGPU, by including the phenotype generation into the evaluation function.

These preliminary results using evolutionary algorithms on GPGPU to solve a real problem still exhibit an interesting speedup. Using EASEA, it was possible to reach a $\approx 60\times$ speedup using a the GTX260 NVIDIA card. This is particularly interesting, because the evaluation does not show a complex algorithm.

Furtheremore, as the problem being optimised is a difficult one, it can directly benefit from optimising a big population during a large number of generations. It is altough a real problem, where the fitness landscape is not known, as it is for Weierstrass toy problem, being used previously.

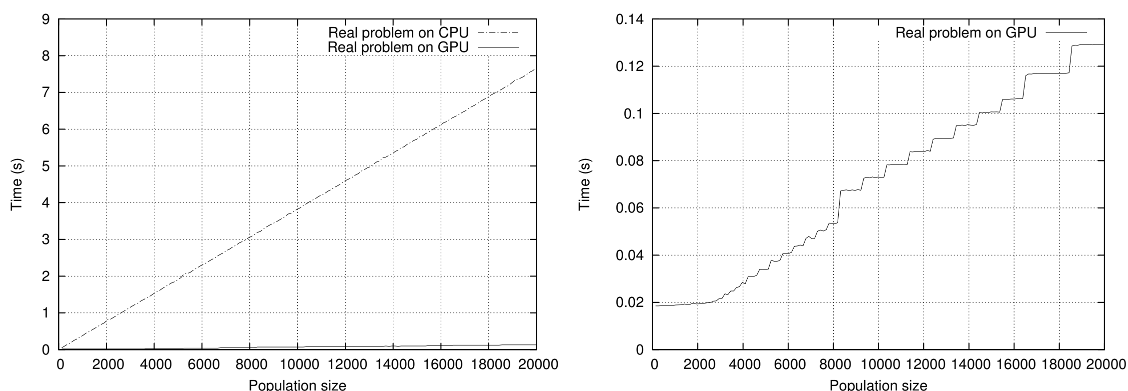


Figure 10.12: Left: evaluation times for increasing population sizes on host CPU (top) and host CPU + GTX260 (bottom). Right: CPU + GTX260 total time.

10.3.4 The zeolite search continues

Futher works were conducted after these preliminary results in our team. Using EASEA to implement prototype and libeasea to implement optimised algorithms, the authors applied EAs with different strategies to optimise the crystalline structure in Baumes et al. [87, 107], Krüger et al. [108]. Using the island model implemented into EASEA, a 40 hours run on 20 machines with ×120 speedup allowed to find 50 matching structures (*cf.* 10.13). One of the contained structure was the real zeolite. This work using EASEA is well described in the Supplied Online Materials of a *Science* paper [109]. It is interesting that the obtained speedups show that this 40 hours run would have taken more than 11 years to complete on a single machine, if the machine did not get stuck in a local optimum before getting there.

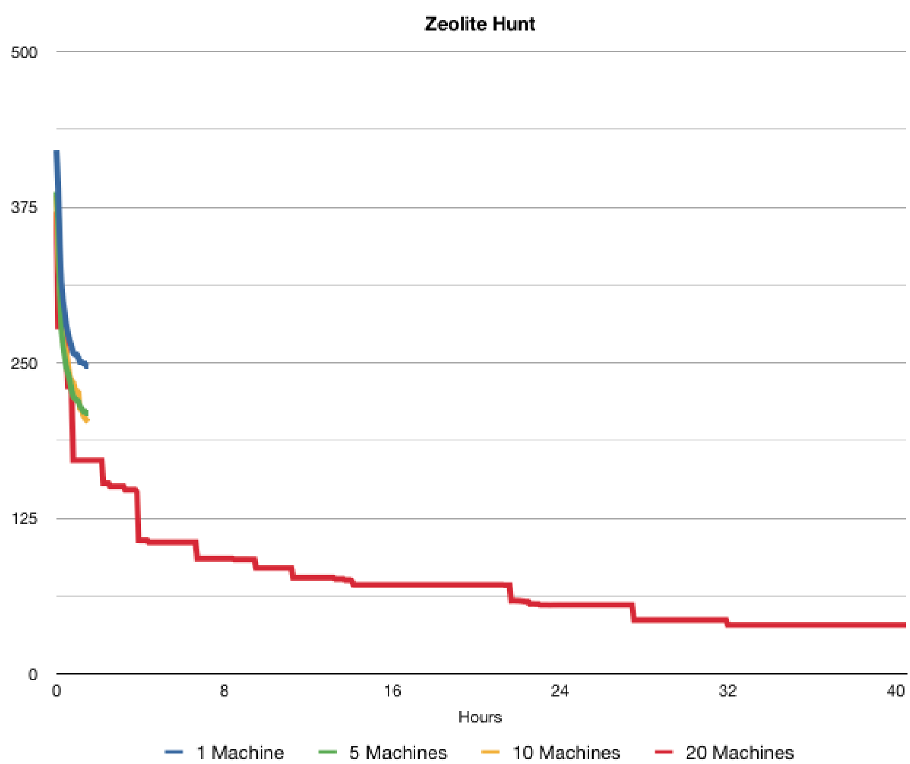


Figure 10.13: 40 hours run on a cluster of 20 GPGPU machines (equivalent to more than 11 years on one machine) that allowed to find 50 possible structures among which the correct structure was present.

Chapter 11

Discussion and perspectives

Contents

11.1 Porting EA on many-core architectures 129
11.1.1 Genetic algorithm and Evolution strategies 129
11.1.2 Population size adaptation 131
11.1.3 Distributed EAs on GPGPUs 132
11.1.4 Asynchronous evolutionary algorithms on GPGPUs 133
11.2 Metrics to compare parallel EA 134
11.2.1 Metric for GA/ES on GPGPU and CPU algorithms 134
11.2.2 Metric for Genetic Programming 135

11.1 Porting evolutionary algorithms on massively many-core architectures

11.1.1 Genetic algorithm and Evolution strategies

The master-slave model is a simple model of parallelization, that uses the parallel architecture only during the population evaluation phase. In this sense, it is the only true parallel implementation of evolutionary algorithms, as understood in high performance computing, where the algorithm works the same way whether it is executed on one core, with the sequential algorithm, or on many cores using the parallel version. This model is therefore different from the island model (dEA) or the fine-grained parallel cellular EAs (cEA). Indeed, if both retain the overall principle of the sequential algorithm, they modify it to match a given hardware architecture.

As a proof, changing the number of nodes and/or the topology in a cEA or dEA modifies the final result. Similarly, it is possible to obtain a speedup by running these alternative models on a sequential architecture, which would make no sense if they were identical to the sequential algorithm.

As said above, the master-slave parallel implementation yields an algorithm whose performances are identical to the sequential one: when using the same parameters, the algorithm returns the same results, whether it is run on one or multiple cores. However, we have seen that in order to get the most of a GPGPU card, it is important to use a population size that allows to use efficiently all the cores of the card. As a result, it may be necessary to use a much larger population size when using a GPGPU card than when using a CPU.

Because most algorithms have been designed to run on CPUs, they are optimized to run on population sizes ranging from 100 to 1000 individuals, where optimizing the load of a GPGPU card may require to use two to three orders of magnitude more individuals.

As a consequence, what may be a very efficient algorithm on a CPU may become very inefficient when run on a GPGPU card. NSGA-II, for instance, uses a ranking algorithm that sequentially calculates an exact Pareto front, whose complexity is in $O(mn^2)$ where m is the number of objectives and n is the population size. If the population size is between 100 and 1000, ranking only takes a fraction of a second on a CPU and is what makes NSGA-II efficient. If 100 000 individuals are used (the kind of population size that is needed to fully exploit GPGPU cards), sequential ranking takes 19 hours, not counting individuals evaluation.

Conversely, one can imagine that some algorithms will work optimally with few individuals only (this is the case of CMA-ES, for instance). With such an algorithm, what benefit is there to use a GPGPU card if it cannot benefit from very large population sizes ?

However, the GPGPU implementation of a master-slave model can be really interesting as could be seen with the examples of sections 6.4 and 10.3. In standard master-slave models, (Cantu-Paz [38], for instance), the evaluation function must be complex enough to justify the additional processing required by the parallelization, because any increase in population size or number of cores will induce additional overhead (mainly due to the transfer of the subpopulations to the computing nodes).

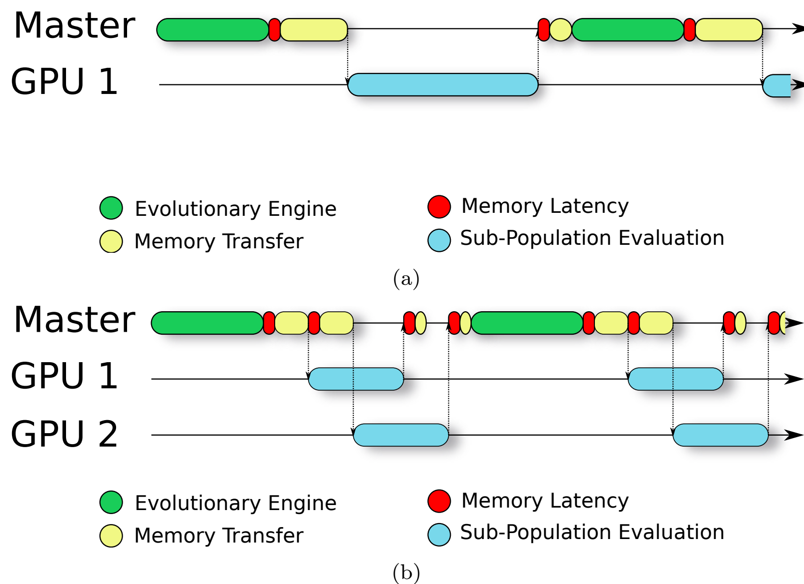


Figure 11.1: Master-slave model using either one (a) or two (b) GPGPUs. Total transfer time remains constant but memory latency increase *w.r.t* to the number of cards.

On GPGPUs (that were not available when this study was made), things are different: once the initial overhead of transferring the population on the GPGPU is absorbed, using an additional core is done at no extra-cost until the GPGPU is completely saturated. On a GPGPU card, because the transfer time between CPU and GPU is so fast, evaluating a population of 10 000 individuals may only take an epsilon more time than evaluating 30 individuals.

Then, once a GPGPU card is saturated, a second one can be used for an additional overhead cost, but this will only consist in memory latency because the transfer time is proportional to the

size of the population, as shown in Figure 11.1(a) and 11.1(b).

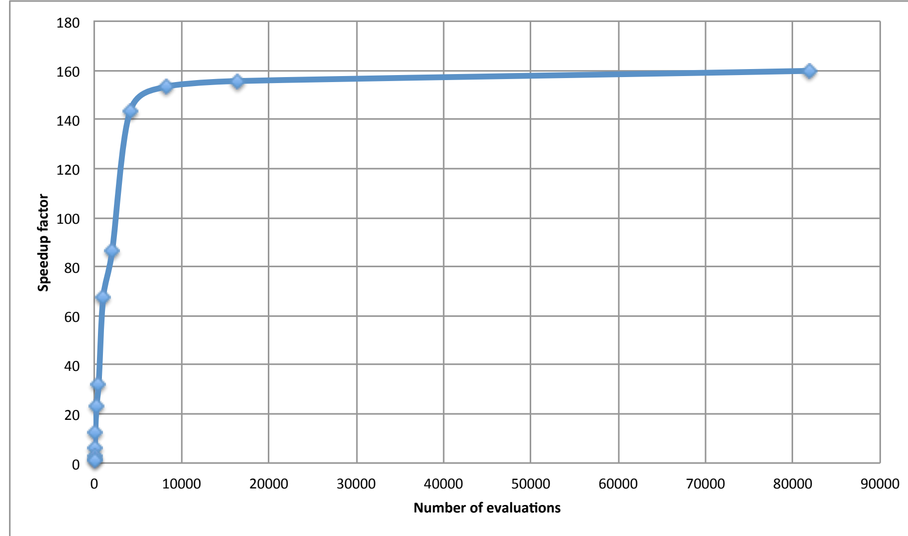


Figure 11.2: GPU-CPU speedup on a 240 cores GTX275 *vs* Xeon 5500 2.57Ghz Core i7 CPU on Weierstrass function depending on population size.

Thus, there is a large granularity in the number of cores to be used, which is the number of cores in one GPGPU processor (between 16 and 512 cores, multiplied by the number of threads that is necessary to load on each core so as to benefit from maximum scheduling efficiency). Using a population size smaller than this grain (around 16 000 individuals on a 512 cores chip) does not make much sense, as can be seen in Figure 11.2 which was obtained on a GTX275 card with 240 cores. A parallel can be drawn between the classical model and the one used with GPGPUs: a node is similar to a card and its memory, while the processor is related to a core. So there is no longer an equality between the number of nodes in the system and the number of processors. This applies to all multi-processor systems, but has more influence with GPGPGUs, as a chip embeds more computing cores.

11.1.2 Population size adaptation

We have seen in previous sections that the use of GPGPUs, especially in the GA/ES, requires to adapt the population size in order to benefit from a maximum speedup compared to the reference sequential algorithm. Population size used for a given hardware is relatively straightforward to calculate. Indeed, it is necessary to use all the cores of a card. But also, in order to take advantage of the scheduling mechanism, it is necessary to fill all the scheduling pools. All in all, the number of individuals must be at least equal to the number of cores $\times$ the size of their scheduling pool (hence the near-maximum speedup observed for 8192 individuals on a 240 cores GTX275 in Figure 11.2).

However, for many problems solved by EAs, there is an optimal population size beyond which efficiency will drop, when it is calculated based on the number of evaluations *vs* quality of the results. This population size is still difficult to calculate for any algorithm and any problem, as it is highly dependent on the fitness landscape and internal operators of the evolutionary algorithm.

A too small size could lead to a premature convergence, since exploration could be too low. Local maxima become very attractive and individuals are drawn up to them. From the hardware

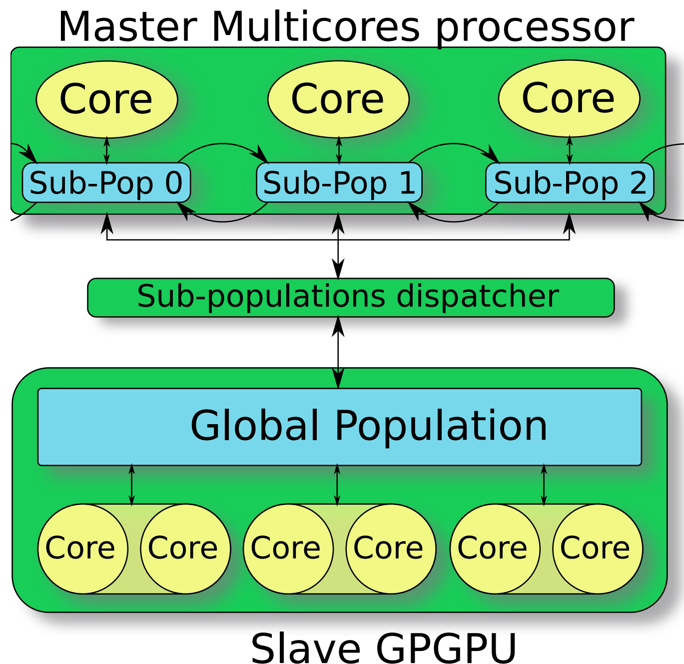


Figure 11.3: Implantation of a mixed dEA/MS onto one GPGPU

point of view, too few individuals results in an suboptimal use of the cores and/or of the memory bandwidth.

A larger population will of course increase diversity within the population which will result in a relative reduction in selection pressure. As the effective pressure is lower, exploitation of good spots is also lower, as well as the efficiency of the algorithm.

Effectiveness can be characterized in two ways. One compares the result of the algorithm based on the number of evaluations. This is particularly the orientation of the CEC contest [110].

However this type of constraint does not take into account the temporal aspect. We saw in section 6.4, that a GPGPU can perform the evaluation of a population of several thousand individuals at the same time-price than one.

This means that it is necessary to develop a new metric that is not based on the number of evaluations in order to rate the quality of massively parallel stochastic optimization paradigms such as massively parallel EAs.

Furthermore, new algorithms must be coined in order to maximize efficiency on massively parallel systems.

11.1.3 Distributed EAs on GPGPUs

Using of GPGPUs in the context of island models is possible along several principles: One or more GPGPU cards can be used as evaluation devices, by implementing a master-slave model on an island, this island being connected to others who themselves use one or several GPGPUs. This is a GPGPU implementation of a mixed dEA/Master-Slave model of parallelization.

However, because of the large population size used on GPGPU, several local islands could have their evaluation phases executed on the same GPGPU. The islands would then run synchronously

and their subpopulations could be gathered in a global population before being evaluated by one or several GPGPUs, the results being distributed to respective islands after the evaluation. This type of implementation would be able to use more conventional population sizes because it would be possible to implement several islands on a single GPGPU processor, as well as using the several cores present on modern CPUs.

As (except in GP, of course) the evaluation time of an individual is generally not related to the content of an individual, using a mostly synchronous evaluation step should not induce performance problems at this level. This implementation would allow a mixed dEA/master-slave on a single GPGPU. This type of implementation would work on an architecture organized as in Figure 11.3.

Moreover, such an implementation could allow to install a low cost and highly complex meshed network, taking advantage of the high speeds communication network available between the GPGPU cards and the central processors. Many configurations could be tested as well, without the need to have access to a cluster-type machine. Without taking into account the necessary synchronization between the various islands, such an implementation could reach the same kind of speedup as the master-slave model using GPGPU.

EASEA already allows to use one or several GPGPU as a slave in a mixed dEA/Master-Slave configuration. Thus, in an island, population evaluation is executed in parallel on one or several GPGPUs. In a similar way, an island can be composed of a single GPGPU and if the machine contains several of them, each GPGPU can be assigned to an island. It is therefore possible to implement several islands on the same machine, with individual migration using the same mechanism as the standard model, by exploiting the network loopback as communication layer.

The DISPAR-GPU implementation and GenGPU can completely parallelize an evolutionary algorithm on a GPGPU card. However, scaling to more cards is more complicated than for the master-slave model. Using an island model on top of these algorithms, would allow to take advantage of such a hardware.

In all cases, these algorithms introduce differences with the standard one, making it difficult to compare these algorithms, and especially to quantify the gain provided by a particular algorithm implementation over another.

11.1.4 Asynchronous evolutionary algorithms on GPGPUs

In Golub and Budin [39], Golub et al. [40], Golub and Jakobovic [41], the authors present an asynchronous evolutionary algorithm, that is adapted to multi-core architectures. It uses the facilities provided by the shared memory between the cores, which allows to remove synchronisations from the code.

This algorithm could be ported on a GPGPU card, even if some adaptation are expected to be useful. Indeed, the number of cores of Golub's implementation is a quite smaller than what is available on GPGPU chips. Some functionalities of GPGPUs could be exploited by such an algorithm, as the SIMD behaviour of SPs inside an *MP*. The probabilistic model should probably be adapted to this last behaviour. Furthermore, the memory structuration should be taken into account, in order to offer good access speed to the algorithm.

A mixed model, where evaluation of each individual is executed by the threads of a warp, in an SIMD manner, when the warps are each executing an asyncrhonous genetic programming algorithm. A SIMD genetic programming implementation could reduce the number of parallel asynchronous algorithms while allowing to use a gretter population size. Both parameters, i.e. the number of parallel algorithm and the population size being included in the probability calculation of wasting computations, these modification could help to port such a model of evolutionary algorithm on a GPGPU.

An asynchronus parallel algorithm could take into account many features of GPGPU cards, that could lead to greater execution speed, compared to current synchronous implementations.

However, as for DISPAR-GPU and GEN-GPU, the resulting algorithms models will be different from standard sequential algorithms. It would be necessary to conduct important studies in order to deeply understand the behaviour of these new algorithms.

11.2 Some metrics to compare parallel evolutionary algorithm implementations

11.2.1 Metric for GA/ES on GPGPU and CPU algorithms

We have seen that the comparison between different types of evolutionary algorithms is not trivial. Even though their general principle remains the same, their exact behaviour is different for various reasons (performance, adaptation to hardware, parallelization ...).

Even the master-slave model which is an exact parallelization of a sequential algorithm and therefore should be completely comparable, loses this property as soon as the settings are adapted to some external constraints such as population sizes larger than 10 000 individuals.

However, it would be interesting to compare these implementations between them, for instance on a given problem, in order to determine the more adapted implementation to solve it. HPC scientists use a metric for comparing sequential and parallel implementations, in terms of time saving. This metric is the speedup and is calculated by T_s/T_p, where T_s is the time of the sequential version of the algorithm and T_p is the time to execute the parallel implementation. For an exact parallel implementation of an algorithm, this metric allows to know the time factor saved when using a parallel machine. With such a speedup value, it is possible to determine the efficiency of the parallelization, by $Speedup/n$ where n is the number of processors used by the parallel algorithm. However, this metric can not be used to compare two different algorithms. Indeed, even an algorithm that scales well could in fact perform perform poorly compared to another one that would not scale as well, but that would be much more efficient. It would then be still necessary to compare the sequential versions, in order to draw any conclusions from the efficiency measurement.

A suitable metric would be interesting because for instance, when reading the available literature, it is difficult to get a clear idea of the real performance of the presented algorithms. Indeed, performance is highly dependent on the implementation (type of operators, sequential part of the algorithm, executed and interpreted programs ...), but also on the hardware used for the experiments. It is therefore very difficult to compare two parallel implementations, without recoding them and running them on the same hardware. But this poses the problem of reproducibility of a published algorithm: how can one know that the implementation of an algorithm presented in a paper is as efficient as the one done by the author, as it is impossible to thoroughly describe a complex algorithm ? These problems are real, as only few implementations are publicly available.

In this sense, the EASEA platform brings a real advantage, as it can be recompiled on many different systems. It would then be possible to evaluate the performance of a published *.ez* algorithm on local hardware.

In Alba [111], the author faces the same kind of problems and defines a taxonomy for speedup in order to measure EA's relative performances. Among this taxonomy, it is possible to cite the main categories, which are strong and weak speedups.

- Strong speedup refers to a comparison with the best-so-far algorithm known to solve the current problem. The author cites the difficulties to find this best algorithm. On parallel implementations, the difficulty is even more important, because the problems used in order to evaluate the implementations are simply benchmarks that do not reflect the problems found in real-world.

- Weak speedup compares a parallel implementation with its sequential counterpart. All the speedups used in this document are based on this principle. Even it the comparison between master-slave algorithms and their sequential counterparts is made easier because of the comparable aspect of their workflow, this is not the case when it is necessary to implement a different paradigm to match the underlying hardware.

Still, two metrics can be found for this type of comparison. It is possible to define a qualitative speedup by choosing a fitness threshold (this is what is done in section 9.3. For each algorithm to be compared, one measures the time required for the best individual to reach a particular threshold. The ratio between the time thus obtained by each algorithm can be considered as a qualitative speedup. This speedup is referred to as "Weak speedup with solution-stop" in Alba [111].

This metric is particularly useful to measure the performance of dEAs, which are clearly not comparable with any sequential algorithms, and for which there are multiple different possible configurations (number of islands, subpopulation size, topology and migration frequency...). However, it is necessary to adjust the threshold to a significant value, or to use several thresholds. If a single threshold is too easy to reach, all algorithms will succeed in a short time. The "best" performance of the algorithms may be underestimated. On the contrary, if the threshold is too difficult, the most basic algorithms will never reach it. Then, the speedup value of the algorithm is unknown or infinite, which gives no information about the quality of the algorithm compared to the others. Another point is that the fitness of the best individual often progresses step-by-step, thus the measured time is usually an overestimation of the real time. Finally, this metric is dependent on the fitness landscape, and a statistically reliable test set must be defined.

The second metric takes fitness into account. For this metric, a time threshold is given, and when the time is up, the fitness values obtained by the best individuals of each algorithm are compared. As with the previous metric, it is necessary to set a parameter and again a wrong setting causes an over or under approximation of the algorithms' performance. The measure is also strongly dependent on the fitness landscape of the problem being considered. In addition, this metric does not provide any informations on the speedup of an algorithm and it is not possible to determine the effectiveness of a parallelization.

It is also possible to use an evaluations/s unit to measure the effectiveness of a parallelization. Indeed, a poor parallelization tends to let the computing nodes be more idle and observed evaluations/s drops down, compared to the theoretical peak performance of the machine. However, this measure is highly dependent on the equipment. It is more a quantitative measure, which gives the ratio of the hardware which is used by the algorithm and not the qualitative efficiency of the entire system, problem-wise.

11.2.2 Metric for Genetic Programming

Usually, genetic programming algorithms use heavy evaluation functions. They are therefore well suited to use of a master-slave model as the first level of parallelization.

Comparability with sequential models are then possible. In addition, compared to previous algorithms, GP generally use very large populations and as the GPGPU implementation presented in chapter 7 implies several levels of parallelization, more parallel tasks have to be performed during the evaluation. This allows to compare a sequential implementation and a GPGPU implementation with the same parameters, without artificially overloading the sequential implementation or underloading the GPGPU hardware.

It is still notable that calculations made on single precision GPGPU cores do not offer the same precision as an *x86* processor, precision being further degraded when using approximated *SFUs*. As the evaluation step consists of a large number of operations (for an individual, it is the number

of nodes multiplied by the number of training cases), the errors can accumulate and evolution can takes a different path, depending on the architecture of the processor performing the evaluation.

Nevertheless, it is possible to compare the evaluation time for two GP implementations: as the master-slave GPGPU and CPU implementations are comparable, it is possible to calculate a speedup as for a conventional parallelization application. However, as is described in section 7.4, different sets of initial parameters can lead to different behaviour. The tree type should be carefully chosen, as the shape of the trees can affect the speed of the evaluation phase. The different criteria involved are: the depth of the trees, their balance and their similarity. Trees of the same size improve load balancing and similar trees can improve the performance of an SIMD system, as for example, the size of the learning function set influences the performance of the implementation in Langdon and Banzhaf [75].

Therefore, the test conditions have tremendous influence on the results obtained by different implementations. It is essential to characterize them completely, especially because only few of these implementations are publicly available.

What is timed must also be defined rigorously. Indeed, comparing the phases of evaluation only is not necessarily representative of the performance of the entire system. The use of an external computing facility for population evaluation (*i.e.* a GPGPU card) requires the addition of translation and transfer phases that could or could not be excluded from the time computation. Both solutions are possible, but for a fair comparison, this needs to be described clearly. Furthermore, a fair comparison should include transfer time, as this is a mandatory phase, in order to compare a master-slave algorithm to its sequential counterpart.

The comparison can also be performed on the complete algorithm. This approach provides a more global perspective, closer to what the final user is expecting. Indeed, for such a user, the speedup is above all a means of obtaining a result faster or a means to achieve better results. But this approach fails to take all the various parameters mentioned above into account, as for example the shape of the fitness landscape, that will influence the shape of the trees in a manner that may adversely affect the performance of a particular implementation.

Finally, the testing method used in section 7.4 takes into account the sequential part of the algorithm. This means that the results are less impressive in terms of speedup, because only one part of the algorithm is accelerated by the use of the master-slave parallelization model. The distribution of computing time between the evolutionary engine and population evaluation is a key point in the final result.

As before, it is necessary to note that the use of speedup as a metric, compares the two implementations but also different underlying hardware. This means that the measure is strongly dependent on the performance ratio between the two computation devices.

In Langdon [112], the author used the speed of evaluation in nodes per second. This metric is comparable and can calculate the evaluation speedup of two implementation. It has the advantage of not being a ratio between two implementation and avoids the authors have to implement a method for performing a comparison. It is also focused on the user's point of view, as it is necessary to take into account the architecture of the machine which is running the algorithm.

As for the use of GPGPU in genetic programming, we are currently seeking the most effective evaluation method, the latter metric centered on evaluation speed seeming particularly appropriate.

Finally, the use of the metrics presented in the previous section 11.2.1 is also possible. The same limitations and areas of application remain true.

Conclusion

The work carried out in this PhD thesis makes it possible to efficiently run evolutionary algorithms on one or several GPGPU cards. Several algorithms have been implemented:

- a master-slave massively parallel evolutionary algorithm for genetic algorithms or evolution strategies,
- a master-slave massively parallel genetic programming algorithm,
- a Disjoint Set PARallel (DISPAR) tournament that implements a parallel selector without replacement for population reduction,
- a complete ES algorithm that runs on a GPU (DISPAR-GPU),
- a complete generational genetic algorithm that runs on a GPU.

The obtained speedup (between 100× and 400×) compared with an identical (or comparable) sequential algorithm means that problems that were not tractable on standard PCs are now within reach.

However, these speedups are only accessible by drastically increasing population sizes. Where most EAs have been designed to run with 100 to 1000 individuals, massive parallelism requires at least 10 000 invididuals to load a single card, and if 4 are installed in the same machine, computing power would be wasted if less than 40 000 individuals were used. These numbers are valid in 2011, but because Moore's law seems to continue to apply, it will be necessary to double them every other year, meaning that it will be necessary to design evolutionary algorithms that will soon need to evolve millions of individuals in parallel.

Fortunately, John Koza has already explored these numbers with the Genetic Programming paradigm, showing that he could routinely obtain human-competitive results with such population sizes.

Another nice feature of parallel EAs is that they allow to efficiently use parallel machines on sequential inverse problems, as rather than needing to parallelize the evaluation of one individual, they run thousands of sequential evaluations in parallel.

However, efficiently using GPGPUs remains a challenge, as porting an algorithm directly on such architectures is conditioned by many constraints, as these computing devices are very different from a standard processor (even a multi-core one). Programming techniques are different and are unlikely to be within reach of applied scientists such as chemists or physicists.

Indeed, programming complexity makes it is very unlikely to be able reproduce the algorithms described in many published papers on GPGPUs.

In an effort to allow non expert programmers to use the elaborated algorithms, a good part of this PhD was spent in adding the developed algorithms into the EASEA platform that is now capable of automatically parallelizing an EA or GP on one or several GPGPU cards.

This means that a same *.ez* source file can be compiled for CPU, ezGPU (the master-slave EA and GP algorithms), DISPAR-GPU (a complete EA / ES algorithm on GPU) or genGPU (a complete generational Genetic Algorithm on GPU), which allows to compare the different algorithms.

An island model has been added to the EASEA platform by another PhD student, allowing to obtain linear and supra-linear speedups when several machines are used, allowing to obtain within a couple of hours some results that should normally have needed years of computation if they had been run on a single non-GPU machine. On a complex problem in chemistry, a cluster of 20 GPU machines running EASEA found 50 zeolite structures among which the good one was found. As a result, the EASEA platform was described in the Supporting Online Material of a Science paper [109].

Finally, a 389k EURO ANR Emergence grant was obtained from the French government to port the EASEA platform on the cloud and the grid (project EASEA-CLOUD) and the EASEA

platform was chosen to be part of a very large 12M EURO French project submission to simulate, predict and optimize Complex Systems (project RAPSODY). The new EASEA platform that was developed in this PhD should therefore be further developed.

As detailed in the discussion section 11.2.1, it would be nice to also extend this work by defining a metric for the comparison of implementations and their settings. Concerning the island model, the promising scalability should be studied on various real-world problems. Finally, an accurate characterization, as found in Cantu-Paz [38] could be defined.

In the mid 2000s, the frequency of personal computers processors has stopped increasing with a peak 3.8GHz on an Intel Pentium IV chip around 2004. In order to still increase the power of their processors, manufacturers chose to benefit from Moore's law (that still seems to be valid) by going the multi-core way, with the side effect that still due to heat dissipation problems, the first multi-core chips saw their frequency drop drastically because two active cores were dissipating more heat than one. Since then, CPU frequencies has started rising again, but is still not up to 2004 frequencies (3.46GHz for Intel Core i7 in January 2011).

So it seems that the evolution path for chips is to continue to increase their number of transistors (rather than their frequency) and as a consequence, their number of cores.

Due to the massively parallel character of graphic rendition (on millions of pixels or vertices), graphic cards manufacturers have chosen to develop specialized chips with the aim of implementing as many cores as possible. By exploiting the specificities of graphic rendition algorithms, they managed to embed in their chips up to two orders of magnitude more tiny graphic cores than there are complex versatile ones in CPUs (6 to 8 cores for CPUs, *vs* up to 512 cores on the latest 2011 GF110 GPGPU chips). Even though these chips run at a lower frequency than CPUs (1.3GHz), the number of cores they implement yields a monstrous 1.6 TFlops (GTX580 / Tesla M2050) *vs* around 50 Gflops for a contemporary Intel Core i7.

This many-core paradigm is carried further by High Performance Computing (HPC), that is nowadays embodied by hierarchical massively parallel machines. Tianhe 1a, the Chinese PetaFlop machine capable of a peak computing power of 4.7 PFlops, is made of 112 computer cabinets containing 3.584 quadri-processor blades for a total of 14.336 hexacore 2.93GHz processors, and 7168 GPU cards (two per blade) each containing 448 cores, for a total of 3 million GPU cores each clocked at 575 MHz.

In order to get a program to run on this super-computer, one should decompose it efficiently in 3.584 different and independent tasks (there are 3.584 computers) that should also divide in 4 (these machines are quadri-processors) and then in 6 (each processor is a hexa-core). Then, each task on each machine should be able to exploit the 896 cores of the two GPGPU cards, knowing that on each card, these cores are structured into 14 groups of 32 cores that must all execute the same instruction at the same time (SIMD parallelism).

Only a bottom-up inherently parallel approach can exploit such hierarchically massively parallel super-computers (but also personal computers of the future) to solve generic problems, and this is typically what evolutionary algorithms are. This PhD should therefore be seen as a contribution to the development of a massively parallel evolutionary computation, to match the new trend that is developing in hardware.

List of Figures

1.1 Two individual representations for two evolutionary algorithm problems. 17
1.2 The fitness landscape corresponding to a 2D "Ackley path" evaluation function as presented in Figure 1.2. 19
1.3 (μ+λ) cEA flow chart. 20
1.4 Uniform crossover between 3 parents, producing one child. 21

2.1 The population sizes configuration of EASEA, accessed using the GUIDE graphical user interface. 26
2.2 The EASEA code generation process. 27

3.1 Flynn's taxonomy of parallel systems. 32
3.2 An execution scenario in master-slave model. 33
3.3 Evaluation in M-S model. 34
3.4 Master-Slave model for EA parallelization 35
3.5 Asynchronous panmictic EA principle. 37
3.6 Island model algorithm. 38
3.7 Example of connection tolopogy for distributed EA. 39
3.8 An example of cEA population, organized as a 2D torus, with a 4-neighbouring around the center individual. 41
3.9 Examples of neighbouring used in cEA. 42

4.1 A schematic view of a CPU-GPGPU system. 45
4.2 SIMD cores performing a memory load from a base address A_b and core id. 46
4.3 Schematic view of multi-processors of two major architecture designs. 47
4.4 GPGPU memory hierarchy, organized by decreasing speed (bandwidth) and increasing size and latency. 48
4.5 Memory bus width and its importance in memory speed. 50
4.6 An example of threads software organization. 51
4.7 SIMD threads executing different execution paths. 52
4.8 An example of scheduling. 53

5.1 Recall from section 1.3.4: EA flow chart. 57
5.2 Amount of computing time for all the steps of an ES optimization of the Weierstrass function with 120 iterations on a CPU, depending on the population size. (Note the logarithmic scale on the Y axis.) 59

6.1 Master-Slave model for EA parallelization. 65
6.2 The Weierstrass function using a 2D (2 real values) genome. 68
6.3 Execution time distribution of an ES algorithm onto CPU 68

6.4 Execution time for the Weierstrass problem, for different numbers of iterations, *w.r.t.* the population size. 69
6.5 Speedup *w.r.t.* the population size, on different NVIDIA GPGPU cards. 70

7.1 Time elapsed into different steps of a GP algorithm, *w.r.t.* the learning set size. . . . 72
7.2 An example of a GP-tree individual and a problem-related learning set. 72
7.3 Execution of the first learning set cases in SIMD-mode. 73
7.4 An example of a GP-tree population and a problem-related learning set. 74
7.5 Execution of the first three nodes of the example tree in an SIMD manner. 74
7.6 Global scheme for GP simulation and timing process. 78
7.7 Evaluation time for depth 7 trees for different implementations with a sampling size of 8 fitness cases. 79
7.8 Resulting speedup between GPGPU and CPU. 80
7.9 Resulting speedup for depth 7 trees on GPGPU against CPU for different function sets. 81
7.10 Resulting speedup for depth 7 trees on GPGPU against CPU for different tree construction methods. 82
7.11 Resulting speedup for depth 7 trees, function set FS2SFU and 32 fitness cases on GPGPU against CPU *wrt* the population size for different implementations of GP evaluations. 83
7.12 Maximal speedup with the tree-based GP implementation on the $\cos 2x$ problem. . . 84
7.13 Speedup with respect to the population size and the number of fitness cases. 84
7.14 State and control variables of an airplane. 85
7.15 Speedup obtained on the airplane problem with EASEA tree-base GP implementation. 87
7.16 Simulated trajectories, from the original model and the evolved one. 88

8.1 Population distribution by fitness value for 1000 7-tournament selections “with” and “without” replacement, from a population of 100 and 1000. (number of individual *w.r.t.* rounded fitness values.) 91
8.2 Mean fitness distribution and repredocution rate over 1000 experiments for with and without replacement selection. (*w.r.t.* rounded fitness values.) 92
8.3 Selection intensity for a tournament selector with and without replacement for 2000 $\rightarrow$ 1000 reduction. The theoretical curve (top curve) plots the equation found in Blickle and Thiele [22]. It is superimposed with the experimental tournament with replacement curve. 93
8.4 Loss of Diversity for (1000+1000)-ES selection using tournament selection. 94
8.5 DISPAR-Tournament principles. 96
8.6 Selection intensity for (1000,1000)-ES reduction, with standard tournament without replacement and DISPAR-tournament. Tournament size is $t \rightarrow 1$ for tournament without replacement, but for DISPAR-tournament, the value of k is calculated depending on the number of individuals that are needed for the next generation. Because 2000 individuals must be reduced to 1000, multi-tournament parameters chosen for DISPAR-tournament are respectively $2 \rightarrow 1$, $4 \rightarrow 2$, $6 \rightarrow 3$, $8 \rightarrow 4$, $10 \rightarrow 5$, ... 97
8.7 Mean fitness on the sphere problem. 98
8.8 A schematic view of a full GPGPU DISPAR-EA. 99
8.9 A schematic view of a full GPGPU generational EA. 100
8.10 Individuals organization for genGPU algorithm. 100
8.11 Rosenbrock function. 102
8.12 Speedup factor for Rosenbrock problem. 102
8.13 Computing time distribution in presented algorithms (s). 103

8.14 Computing time distribution in presented algorithms (% of execution time). 103
8.15 Evolution of the best individual for the three GPGPU parallel algorithms (average on 20 runs). 104

9.1 Overview of the complete LibEASEA internal loop. 111
9.2 Recall from Figure 3.3 in section 3.3.1: evaluation in master-slave model. 112
9.3 Speedup factor achieved with dEA on weierstrass benchmark. 113

10.1 Efficacy of carbonate leaching models using various strategies. 116
10.2 Pareto-frontiers for carbonate leaching. 118
10.3 An example of a relational model. 120
10.4 Example of concept based on complex aggregates. 120
10.5 An example of aggregation using mean as aggregation operation 121
10.6 An example of aggregation using our language. 121
10.7 Example of a genome of m meta-genes. 122
10.8 Convergence of the evolutionary algorithm on the artificial dataset. 123
10.9 Example of decision tree learned as a solution of the concept of figure 10.4. 123
10.10The relational model of our geographic databases. 124
10.11LTA crystal framework: a) the white cube is the unit cell, b) cages are represented in green, c) 3D channels are in blue, d) split of the structure LTA in building units, e) piece of LTA structure, and f) crystal structure with 27 unit cells. 126
10.12Left: evaluation times for increasing population sizes on host CPU (top) and host CPU + GTX260 (bottom). Right: CPU + GTX260 total time. 127
10.1340 hours run on a cluster of 20 GPGPU machines (equivalent to more than 11 years on one machine) that allowed to find 50 possible structures among which the correct structure was present. 128

11.1 Master-slave model using either one (a) or two (b) GPGPUs. Total transfer time remains constant but memory latency increase *w.r.t* to the number of cards. 130
11.2 GPU-CPU speedup on a 240 cores GTX275 *vs* Xeon 5500 2.57Ghz Core i7 CPU on Weierstrass function depending on population size. 131
11.3 Implantation of a mixed dEA/MS onto one GPGPU 132

Bibliography

[1] G.J. Friedman. Digital simulation of an evolutionary process. *General Systems Yearbook*, 4: 171–184, 1959.

[2] A.S. Fraser. Simulation of genetic systems by automatic digital computers vi. epistasis. *Australian Journal of Biological Sciences*, 13(2):150–162, 1957.

[3] I. Rechenberg. Evolutionsstrategie: Optimierung technischer systeme nach prinzipien der biologischen evolution. *Frommann-Holzboog Verlag, Stuttgart. German*, 1973.

[4] H.P. Schwefel. *Numerical Optimization of Computer Models.* Wiley, Chichester, 1981.

[5] J.H. Holland. *Adaptation in natural and artificial systems.* MIT Press, Cambridge, MA, USA, 1992.

[6] D.E Goldberg. *Genetic algorithms in search, optimization and machine learning.* Addison-Wesley, 1989.

[7] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial intelligence through simulated evolution.* John Wiley, 1966.

[8] D.B. Fogel. An analysis of evolutionary programming. In D.B. Fogel and W. Atmar, editors, *First Annual Conference on Evolutionary Programming*, pages 43–51, 1992.

[9] N.L. Cramer. A representation for the adaptive generation of simple sequential programs. In John J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms*, pages 183–187, 1985.

[10] J.R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems).* MIT Press, Cambridge, CA, 1992.

[11] P. Collet, J. Louchet, and E. Lutton. Issues on the optimisation of evolutionary algorithms code. In *Proceedings of the Evolutionary Computation on 2002. CEC '02. Proceedings of the 2002 Congress - Volume 02*, CEC '02, pages 1103–1108, Washington, DC, USA, 2002. IEEE Computer Society.

[12] K.A. De Jong. *Evolutionary Computation: A Unified Approach.* MIT Press, 2006.

[13] A. Burke, D. Elliman, and R. Weare. A genetic algorithm based university timetabling system. In *East-West Conference on Computer Technologies in Education, Crimea, Ukraine*, pages 35–40, 1994.

[14] K. Deb and R.B. Agrawal. Simulated binary crossover for continuous search space. *Complex Systems*, 9:115–148, 1995.

[15] J.R. Koza and J.P. Rice. *Genetic programming II: automatic discovery of reusable programs.* MIT press Cambridge, MA, 1994.

[16] J.R. Koza, Forrest H.B. III, Andre D., and Keane M.A. *Genetic programming III: Darwinian invention and problem solving.* Morgan Kaufmann, 1999.

[17] J. R. Koza and al. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence.* Kluwer Academic Publishers, 2003.

[18] Thomas Bäck. *Evolutionary algorithms in theory and practice - evolution strategies, evolutionary programming, genetic algorithms.* Oxford University Press, 1996.

[19] H.P. Schwefel. *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie.* Birkhäuser, 1977.

[20] Hans-Paul Schwefel. Collective phenomena in evolutionary systems. In P. Checkland and I. Kiss, editors, *Problems of Constancy and Change – The Complementarity of Systems Approaches to Complexity, Proc. 31st Annual Meeting*, volume 2, pages 1025–1033, Budapest, 1987. Int'l Soc. for General System Research.

[21] Bäck T. Self-adaptation in genetic algorithms. In F J Varela and P Bourgine, editors, *Proceedings of the First European Conference on Artificial Life*, pages 263–271. MIT Press, 1992.

[22] T. Blickle and L. Thiele. A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, 4(4):361–394, 1996.

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

[24] P. Collet, E. Lutton, M. Schoenauer, and J. Louchet. Take it EASEA. In Marc Schoenauer, Kalyanmoy Deb, Günther Rudolph, Xin Yao, Evelyne Lutton, Juan Merelo, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature PPSN VI*, volume 1917 of *Lecture Notes in Computer Science*, pages 891–901. Springer Berlin / Heidelberg, 2000.

[25] P. Collet and M. Schoenauer. Guide: Unifying evolutionary engines through a graphical user interface. In Pierre Liardet, Pierre Collet, Cyril Fonlupt, Evelyne Lutton, and Marc Schoenauer, editors, *Artificial Evolution*, volume 2936 of *Lecture Notes in Computer Science*, pages 203–215. Springer Berlin / Heidelberg, 2004.

[26] M. G. Arenas, P. Collet, A. E. Eiben, Márk Jelasity, J. J. Merelo, Mike Preuss, and Marc Schoenauer. A framework for distributed evolutionary algorithms. In Juan J. Merelo Guervós, Panagiotis Adamidis, Hans-Georg Beyer, José Luis Fernández-Villacañas Martín, and Hans-Paul Schwefel, editors, *Proceedings of PPSN 2002*, pages 665–675. Springer, 2002.

[27] E. Lutton, P. Collet, and J. Louchet. Easea comparisons on test functions: Galib versus eo. In Pierre Collet, Cyril Fonlupt, Jin-Kao Hao, Evelyne Lutton, and Marc Schoenauer, editors, *Artificial Evolution*, volume 2310 of *Lecture Notes in Computer Science*, pages 219–230. Springer Berlin / Heidelberg, 2002.

[28] E. Cantú-Paz. A survey of parallel genetic algorithms. *Calculateurs paralleles, reseaux et systems repartis*, 10(2):141–171, 1998.

[29] J.H. Holland. A universal computer capable of executing an arbitrary number of sub-programs simultaneously. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, IRE-AIEE-ACM '59 (Eastern), pages 108–113, New York, NY, USA, 1959. ACM.

[30] J.H. Holland. Iterative circuit computers. In *Papers presented at the May 3-5, 1960, western joint IRE-AIEE-ACM computer conference*, IRE-AIEE-ACM '60 (Western), pages 259–265, New York, NY, USA, 1960. ACM.

[31] A.D. Bethke. Comparison of genetic algorithms and gradient-based optimizers on parallel processors: Efficiency of use of processing capacity. Technical report, University of Michigan, 1976.

[32] J.J. Grefenstette. *Parallel Adaptive Algorithms for Function Optimization:(preliminary Report)*. Computer Science Dept., Vanderbilt University, Vanderbilt University. Dept. of Computer Science, 1981.

[33] B.L. Miller and D.E. Goldberg. Optimal sampling for genetic algorithms. *Urbana*, 51:61801, 1996.

[34] J.R. Koza, S.H. Al-Sakran, and L.W. Jones. Automated re-invention of six patented optical lens systems using genetic programming. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, GECCO '05, pages 1953–1960, New York, NY, USA, 2005. ACM.

[35] E. Cantú-Paz and D.E Goldberg. Parallel genetic algorithms with distributed panmictic populations, 1999.

[36] E. Alba and M. Tomassini. Parallelism and evolutionary algorithms. *Evolutionary Computation, IEEE Transactions on*, 6(5):443 – 462, oct 2002.

[37] E. Cantú-Paz. A summary of research on parallel genetic algorithms. Illigal report no. 95007, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana Champaign, 1995.

[38] E. Cantu-Paz. Designing efficient master-slave parallel genetic algorithms. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, page 455, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[39] M. Golub and L. Budin. An asynchronous model of global parallel genetic algorithms. In C. Fyfe, editor, *Second ICSC Symposium on Engineering of Intelligent Systems EIS2000*, pages 353–359, Scotland, UK, 2000.

[40] M. Golub, D. Jakobović, and L. Budin. Parallelization of elimination tournament selection without synchronization. In B. Patkai and I.J. Rudas, editors, *5th IEEE International Conference on Intelligent Engineering Systems INES 2001*, 2001.

[41] M. Golub and D. Jakobovic. A new model of global parallel genetic algorithm. In D. Kalpic and V.H. Dobric, editors, *Information Technology Interfaces, 2000.*, pages 363–368. IEEE, 2000.

[42] D. Andre and J.R. Koza. A parallel implementation of genetic programming that achieves super-linear performance. In H.R. Arabnia, editor, *International Conference on Parallel and Distributed Processing Techniques and Applications*, volume 3, pages 1163–1174, 1996.

[43] Frédéric Krüger, Pierre Collet, and Laurent Baumes. Exploiting clusters of gpu machines with the easea platform. Technical report, LSIIT, 2011.

[44] D. Whitley, S. Rana, and R.B. Heckendorn. The island model genetic algorithm: On separability, population size and convergence. *Journal of computing and information technology*, 7 (1):33–47, 1999.

[45] E. Cantú-Paz. Migration policies, selection pressure, and parallel evolutionary algorithms. *Journal of Heuristics*, 7:311–334, 2001.

[46] D. Andre and J.R. Koza. Parallel genetic programming on a network of transputers. In J.P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 111–120, 1995.

[47] E. Alba and B. Dorronsoro. *Cellular genetic algorithms*, volume 42. Springer Verlag, 2008.

[48] G.G. Robertson. Parallel implementation of genetic algorithms in a classifier system. In J.J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, pages 140–147, Hillsdale, NJ, USA, 1987. L. Erlbaum Associates Inc.

[49] H Mühlenbein, M. Gorges-Schleuter, and O. Krämer. Evolution algorithms in combinatorial optimization. *Parallel Computing*, 7(1):65–85, 1988.

[50] Martina Gorges-Schleuter. Asparagos, a parallel genetic algorithm and population genetics. In *Workshop on Evolutionary Models and Strategies, Workshop on Parallel Processing: Logic, Organization, and Technology: Parallelism, Learning, Evolution*, WOPPLOT '89, pages 407–418, London, UK, UK, 1991. Springer-Verlag.

[51] Martina Gorges-Schleuter. Asparagos an asynchronous parallel genetic optimization strategy. In *Proceedings of the third international conference on Genetic algorithms*, pages 422–427, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[52] Heinz Mühlenbein. Parallel genetic algorithms, population genetics, and combinatorial optimization. In *Parallelism, Learning, Evolution*, WOPPLOT '89, pages 398–406, London, UK, UK, 1989. Springer-Verlag.

[53] Heinz Mühlenbein. Parallel genetic algorithms population genetics and combinatorial optimization. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 416–421, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[54] Martina Gorges-Schleuter. Asparagos a parallel genetic algorithm and population genetics. In J. Becker, I. Eisele, and F. Mündemann, editors, *Parallelism, Learning, Evolution*, volume 565 of *Lecture Notes in Computer Science*, pages 407–418. Springer Berlin / Heidelberg, 1991.

[55] L. Darrell Whitley. Cellular genetic algorithms. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 658–, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[56] Marco Tomassini. Cellular evolutionary algorithms. In Jiri Kroc, Peter M.A. Sloot, and Alfons G. Hoekstra, editors, *Simulating Complex Systems by Cellular Automata*, volume 0 of *Understanding Complex Systems*, pages 167–191. Springer Berlin / Heidelberg, 2010.

[57] G. Dick. A comparison of localised and global niching methods. In *17th Annual Colloquium of the Spatial Information Research Centre (SIRC 2005: A Spatio-temporal Workshop)*, pages 91–101, 2005.

[58] Sven E. Eklund. Empirical studies of neighborhood shapes in the massively parallel diffusion model. In *Proceedings of the 16th Brazilian Symposium on Artificial Intelligence: Advances in Artificial Intelligence*, SBIA '02, pages 185–194, London, UK, UK, 2002. Springer-Verlag.

[59] Jayshree Sarma and Kenneth A. De Jong. An analysis of the effects of neighborhood size and shape on local selection algorithms. In *Proceedings of the 4th International Conference on Parallel Problem Solving from Nature*, PPSN IV, pages 236–244, London, UK, 1996. Springer-Verlag.

[60] Enrique Alba and José Troya. Cellular evolutionary algorithms: Evaluating the influence of ratio. In Marc Schoenauer, Kalyanmoy Deb, Günther Rudolph, Xin Yao, Evelyne Lutton, Juan Merelo, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature PPSN VI*, volume 1917 of *Lecture Notes in Computer Science*, pages 29–38. Springer Berlin / Heidelberg, 2000.

[61] J. N. England. A system for interactive modeling of physical curved surface objects. *SIGGRAPH Comput. Graph.*, 12:336–340, August 1978.

[62] Michael Potmesil and Eric M. Hoffert. The pixel machine: a parallel image computer. *SIGGRAPH Comput. Graph.*, 23:69–78, July 1989.

[63] Mark J. Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra. Physically-based visual simulation on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '02, pages 109–118, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[64] Jed Lengyel, Mark Reichert, Bruce R. Donald, and Donald P. Greenberg. Real-time robot motion planning using rasterizing computer graphics hardware. *SIGGRAPH Comput. Graph.*, 24:327–335, September 1990.

[65] Gershon Kedem and Yuriko Ishihara. Brute force attack on unix passwords with simd computer. In *Proceedings of the 8th conference on USENIX Security Symposium - Volume 8*, pages 8–8, Berkeley, CA, USA, 1999. USENIX Association.

[66] Qizhi Yu, Chongcheng Chen, and Zhigeng Pan. Parallel genetic algorithms on programmable graphics hardware. In *Advances in Natural ComputationICNC 2005, Proceedings, Part III*, volume 3612 of *LNCS*, pages 1051–1059, Changsha, August 27-29 2005. Springer.

[67] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM New York, 1967.

[68] Ka-Ling Fok, Tien-Tsin Wong, and Man-Leung Wong. Evolutionary computing on consumer graphics hardware. *Intelligent Systems, IEEE*, 22(2):69–78, 2007.

[69] JIAN-MING LI, XIAO-JING WANG, RONG-SHENG HE, and ZHONG-XIAN CHI. An efficient fine-grained parallel genetic algorithm based on gpu-accelerated. *Network and Parallel Computing Workshops, IFIP International Conference on*, 0:855–862, 2007.

[70] Ogier Maitre, Laurent A. Baumes, Nicolas Lachiche, Avelino Corma, and Pierre Collet. Coarse grain parallelization of evolutionary algorithms on gpgpu cards with easea. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1403–1410, New York, NY, USA, 2009. ACM.

[71] Ogier Maitre, Frédéric Krüger, Stéphane Querry, Nicolas Lachiche, and Pierre Collet. Easea: Specification and execution of evolutionary algorithms on gpgpu. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, pages 1–19, 2011. Special issue on Evolutionary Computation on General Purpose Graphics Processing Units.

[72] O. Maitre, N. Lachiche, P. Clauss, L. Baumes, A. Corma, and P. Collet. Efficient Parallel Implementation of Evolutionary Algorithms on GPGPU Cards. *Euro-Par 2009 Parallel Processing*, pages 974–985, 2009.

[73] Darren M. Chitty. A data parallel approach to genetic programming using programmable graphics hardware. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1566–1573, New York, NY, USA, 2007. ACM.

[74] Simon Harding and Wolfgang Banzhaf. Fast genetic programming on gpus. In *EuroGP'07: Proceedings of the 10th European conference on Genetic programming*, pages 90–101, Berlin, Heidelberg, 2007. Springer-Verlag.

[75] William B. Langdon and Wolfgang Banzhaf. A SIMD interpreter for genetic programming on GPU graphics cards. In Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, Anna Isabel Esparcia Alcazar, Ivanoe De Falco, Antonio Della Cioppa, and Ernesto Tarantino, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 73–85, Naples, 26-28 March 2008. Springer.

[76] Denis Robilliard, Virginie Marion-Poty, and Cyril Fonlupt. Population parallel GP on the g80 GPU. In *Genetic Programming*, pages 98–109. Springer, 2008.

[77] Denis Robilliard, Virginie Marion, and Cyril Fonlupt. High performance genetic programming on GPU. In *Proceedings of the 2009 workshop on Bio-inspired algorithms for distributed systems*, pages 85–94, Barcelona, Spain, 2009. ACM, New York.

[78] Arijit Biswas, Ogier Maitre, Debanga Nandan mondal, Syamal Kanti das, Prodip Kumar sen, Pierre Collet, and Nirupam Chakraborti. Data driven multi-objective analysis of manganese leaching from low grade sources using genetic algorithms, genetic programming and other allied strategies. *Materials and Manufacturing Processes*, 26(3)(26):415–430, Apr 2011.

[79] S.E. Lyshevski. State-space multivariable non-linear identification and control of aircraft. *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, 213(6):387–397, 1999.

[80] Ogier Maitre, Nicolas Lachiche, and Pierre Collet. Fast evaluation of gp trees on gpgpu by optimizing hardware scheduling. In Anna Esparcia-Alcázar, Anikó Ekárt, Sara Silva, Stephen Dignum, and A. Sima Etaner-Uyar, editors, *EuroGP 2010, Genetic Programming*, volume 6021 of *Lecture Notes in Computer Science*, pages 301–312. Springer, 2010.

[81] Nicolas Lachiche et Pierre Collet Ogier Maitre, Stéphane Querry. EASEA parallelization of tree-based genetic programming. *IEEE CEC 2010*, 2010.

[82] T. Motoki. Calculating the expected loss of diversity of selection schemes. *Evolutionary Computation*, 10(4):397–422, 2002.

[83] Artem Sokolov and Darrell Whitley. Unbiased tournament selection. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, GECCO '05, pages 1131–1138, New York, NY, USA, 2005. ACM.

[84] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, 6(2):182 –197, apr 2002.

[85] M. Wall. Galib: A c++ library of genetic algorithm components. *Mechanical Engineering Department, Massachusetts Institute of Technology*, 1996.

[86] M. Keijzer, J. Merelo, G. Romero, and Marc Schoenauer. Evolving objects: A general purpose evolutionary computation library. In Pierre Collet, Cyril Fonlupt, Jin-Kao Hao, Evelyne Lutton, and Marc Schoenauer, editors, *Artificial Evolution*, volume 2310 of *Lecture Notes in Computer Science*, pages 829–888. Springer Berlin / Heidelberg, 2002.

[87] Laurent Baumes, Frédéric Krüger, and Pierre Collet. Easea: a generic optimization tool for gpu machines in asynchronous island model. In *KomPlasTech*, page 12, 2011.

[88] F. Pettersson, N. Chakraborti, and H. Saxén. A genetic algorithms based multi-objective neural net applied to noisy blast furnace data. *Applied Soft Computing*, 7(1):387–397, 2007.

[89] S. Luke. ECJ 16: A Java evolutionary computation library, 2007.

[90] F. Pettersson, A. Biswas, P.K. Sen, H. Saxen, and N. Chakraborti. Analyzing leaching data for low-grade manganese ore using neural nets and multiobjective genetic algorithms. *Materials and Manufacturing Processes*, 24(3):320–330, 2009.

[91] Silvia Poles, Paolo Geremia, F. Campos, S. Weston, and M. Islam. Moga-ii for an automotive cooling duct optimization on distributed resources. In Shigeru Obayashi, Kalyanmoy Deb, Carlo Poloni, Tomoyuki Hiroyasu, and Tadahiko Murata, editors, *Evolutionary Multi-Criterion Optimization*, volume 4403 of *Lecture Notes in Computer Science*, pages 633–644. Springer Berlin / Heidelberg, 2007.

[92] S. Mostaghim and J. Teich. Strategies for finding good local guides in multi-objective particle swarm optimization (mopso). In *Swarm Intelligence Symposium, 2003. SIS'03. Proceedings of the 2003 IEEE*, pages 26–33. IEEE, 2003.

[93] Arijit Biswas. *Optimization of process flowsheets for extraction of non ferrous metals from lean manganese bearing ores.* PhD thesis, Indian Institute of Technology, Kharagpur, 2011.

[94] Carlos M. Fonseca. *Multiobjective Genetic Algorithms with Application to Control Engineering Problems.* PhD thesis, Department of Automatic Control and Systems Engineering, University of Sheffield, Sheffield, UK, 1995.

[95] Luc De Raedt. Attribute-value learning versus inductive logic programming: The missing links (extended abstract). In David Page, editor, *ILP*, volume 1446 of *Lecture Notes in Computer Science*, pages 1–8. Springer, 1998.

[96] Celine Vens, Jan Ramon, and Hendrik Blockeel. Refining aggregate conditions in relational learning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *PKDD*, volume 4213 of *Lecture Notes in Computer Science*, pages 383–394. Springer, 2006.

[97] J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo, CA, 1993.

[98] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques, Third Edition (The Morgan Kaufmann Series in Data Management Systems).* Morgan Kaufmann, Burlington, MA, 3 edition, January 2011.

[99] Deepak Sharma and Pierre Collet. An archived-based stochastic ranking evolutionary algorithm (asrea) for multi-objective optimization. In *GECCO '10: Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 479–486, New York, NY, USA, 2010. ACM.

[100] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, April 2002.

[101] Pierre Collet, Evelyne Lutton, Frédéric Raynal, and Marc Schoenauer. Polar ifs + parisian genetic programming = efficient ifs inverse problem solving. *Genetic Programming and Evolvable Machines Journal*, 1(4):339–361, 2000. October.

[102] R. A. Young. *The Rietveld Method*. OUP and International Union of Crystallography, 1993.

[103] L. A. Baumes, M. Moliner, N. Nicoloyannis, and A. Corma. A reliable methodology for high throughput identification of a mixture of crystallographic phases from powder x-ray diffraction data. *CrystEngComm*, 10:1321–1324, 2008.

[104] Laurent A. Baumes, Manuel Moliner, and Avelino Corma. Design of a full-profile-matching solution for high-throughput analysis of multiphase samples through powder x-ray diffraction. *Chemistry – A European Journal*, 15(17):4258–4269, 2009.

[105] A. Corma, M. Moliner, J. M. Serra, P. Serna, M. J. Diaz-Cabanas, and L. A. Baumes. A new mapping/exploration approach for ht synthesis of zeolites. *Chemistry of Materials*, pages 3287–3296, 2006.

[106] A. Corma, F. Rey, J. Rius, M.J. Sabater, and S. Valencia. Supramolecular self-assembled molecules as organic directing agent for synthesis of zeolites. *Nature*, 431:287–290, 2004.

[107] Laurent Baumes, Frédéric Krüger, Santiago Jimenez, Pierre Collet, and Avelino Corma. Boosting theoretical zeolitic framework generation for the determination of new materials structures using gpu programming. *Physical Chemistry Chemical Physics*, 13:4674–4678, 2011.

[108] Frédéric Krüger, Ogier Maitre, Santiago Jimenez, Laurent Baumes, and Pierre Collet. Speedups between x70 and x120 for a generic local search (memetic) algorithm on a single gpgpu chip. In Cecilia Di Chio, Stefano Cagnoni, Carlos Cotta, Marc Ebner, Anikó Ekárt, Anna Esparcia-Alcazar, Chi-Keong Goh, Juan Merelo, Ferrante Neri, Mike Preuß, Julian Togelius, and Georgios Yannakakis, editors, *EvoNum 2010*, volume 6024 of *LNCS*, pages 501–511. Springer Berlin / Heidelberg, 2010.

[109] J. Jiang, J.L. Jorda, J. Yu, L.A. Baumes, E. Mugnaioli, M.J. Diaz-Cabanas, U. Kolb, and A. Corma. Synthesis and structure determination of the hierarchical meso-microporous zeolite itq-43. *Science*, 333(6046):1131–1134, 2011.

[110] K. Tang, X. Li, P.N. Suganthan, Z. Yang, and T. Weise. Benchmark functions for the cec'2010 special session and competition on large scale global optimization. *NICAL, USTC, China, Tech. Rep*, 2009.

[111] Enrique Alba. Parallel evolutionary algorithms can achieve super-linear performance. *Information Processing Letters*, 82(1):7 – 13, 2002.

[112] W. B. Langdon. A many threaded CUDA interpreter for genetic programming. In Anna I Esparcia-Alcazar, Aniko Ekart, and Sara Silva, editors, *EuroGP 2010*, Istanbul, 7-9 April 2010.

Appendix A

Personal Bibliography

A.1 Book chapters

1-MKSQxx O. Maitre, F. Krüger, D. Sharma, S. Querry, N. Lachiche, P. Collet
Evolutionary Algorithm onto GPGPU using EASEA, Programming Multi-core and Many-core Computing Systems Sabri Pllana and Fatos Xhafa (Eds.), John Wiley & Sons

A.2 Journal articles

2-BMNK11 A. Biswas, O. Maitre, D. Nandan Mondal, S. Kanti Das, P. Kumar Sen, P. Collet, N. Chakraborti
Data Driven Multi-objective Analysis of Manganese Leaching from Low Grade Sources Using Genetic Algorithms, Genetic Programming and Other Allied Strategies Materials and Manufacturing Processes, pp. 415–430, Vol. 26(3), Num. 26, april 2011

2-MKQLxx O. Maitre, F. Krüger, S. Querry, N. Lachiche, P. Collet
EASEA: Specification and Execution of Evolutionary Algorithms on GPGPU Soft Computing - A Fusion of Foundations, Methodologies and Applications, Special issue on Evolutionary Computation on General Purpose Graphics Processing Units

A.3 Conference papers

4-MCLxx O. Maitre, P. Collet, N. Lachiche
Propositionalisation as complex aggregates thanks to an evolutionary algorithm, Artificial Evolution 2011 Angers, France, To appear

4-MLCxx O. Maitre, N. Lachiche, P. Collet
Two ports of a full evolutionary algorithm onto GPGPU, Artificial Evolution 2011 Angers, France, To appear

4-MSLC11 O. Maitre, D. Sharma, N. Lachiche, P. Collet
DISPAR-Tournament: A Parallel Population Reduction Operator That Behaves Like a Tournament, EvoApplications 2011, Torino, Italie, pp. 284–293, C. Di Chio et al. (Eds.), Springer, Lecture Notes in Computer Science, Vol. 6624, Heidelberg, april 2011

4-KMJB10 F. Krüger, O. Maitre, S. Jimenez, L. Baumes, P. Collet
Speedups between x70 and x120 for a generic local search (memetic) algorithm on a single GPGPU chip, EvoNum 2010, Istanbul, Turquie, pp. 501–511, Di Chio, Cecilia and Cagnoni, Stefano and Cotta, Carlos and Ebner, Marc and Ekárt, Anikó and Esparcia-Alcazar, Anna and Goh, Chi-Keong and Merelo, Juan and Neri, Ferrante and Preuß, Mike and Togelius, Julian and Yannakakis, Georgios (Eds.), Springer Berlin / Heidelberg, Lecture Notes in Computer Science, Vol. 6024, 2010

4-MLC10 O. Maitre, N. Lachiche, P. Collet
Fast Evaluation of GP Trees on GPGPU by Optimizing Hardware Scheduling, EuroGP 2010, Genetic Programming, Istanbul, Turquie, pp. 301–312, Esparcia-Alcázar, Anna and Ekárt, Anikó and Silva, Sara and Dignum, Stephen and Etaner-Uyar, A. Sima (Eds.), Springer, Lecture Notes in Computer Science, Vol. 6021, 2010

4-BKJM10 L. Baumes, F. Krüger, S. Jimenez, O. Maitre, P. Collet, A. Corma
Boosting zeolite structure prediction, IZC-IMMS2010 (Sorrento - Italy, 4-9 July 2010) France, 2010

4-MQLC10 O. Maitre, S. Querry, N. Lachiche, P. Collet
EASEA Parallelization of Tree-Based Genetic Programming, IEEE CEC 2010 pp. 1–8, Fogel et al. (Eds.), IEEE, 2010

4-MLCB09 O. Maitre, N. Lachiche, P. Clauss, L. Baumes, A. Corma, P. Collet
Efficient Parallel Implementation of Evolutionary Algorithms on GPGPU cards, Europar 2009 France, pp. 974–985, Springer, Lecture Notes in Computer Science, Vol. 5704, 2009

4-MBLC09 O. Maitre, L. Baumes, N. Lachiche, A. Corma, P. Collet
Coarse Grain Parallelization of Evolutionary Algorithms on GPGPU Cards with EASEA, 11th Annual Conference on Genetic and Evolutionary Computation (GECCO0́9) France, pp. 1403–1410, Franz Rothlauf (Eds.), ACM, 2009

7-BJML09 L. Baumes, S. Jimenez, O. Maitre, N. Lachiche, P. Collet, A. Corma
Graphical Cards for Scientific Calculation, European Conference on Combinatorial Catalysis Research and High-Throughput Technologies Espagne, Selected on abstract, 2009

Evolution Artificielle sur GPGPU : Résumé

1 Introduction

Les algorithmes évolutionnaires permettent de trouver des réponses satisfaisantes, mais non-nécessairement optimales à des problèmes complexes. La puissance de ces algorithmes est directement corrélée à la puissance de calcul disponible pour leur exécution. En effet ces algorithmes réalisent une exploration en parallèle de l'espace de recherche, par le biais de l'évolution d'une population d'individus plus ou moins adaptés à la résolution du problème. La puissance de calcul disponible contraint la taille de la population et donc la capacité d'exploration ou d'exploitation qu'offre un algorithme.

Parallèlement, nous assistons au développement des architectures de processeurs multi-cœurs. Ces processeurs peuvent contenir jusqu'à plusieurs centaines de cœurs dans une puce, mais possèdent des contraintes structurelles qui imposent une adaptation des algorithmes utilisés. Parmi ces processeurs se développent depuis 2007 les processeurs de type GPGPU (pour General Purpose Graphical Processing Unit), qui sont des versions généralisées de puce de rendu 3D. Ces processeurs possèdent jusqu'à plusieurs centaines de cœurs par puces et permettent d'obtenir des accélérations de plusieurs centaines de fois, sur certaines applications.

2 Algorithmes évolutionnaires

Les algorithmes évolutionnaires réalisent une exploration stochastique biaisée et parallèle d'un espace de recherche, à l'aide d'une population d'individus. Le but étant d'adapter au fur et à mesure des générations ces individus aux problèmes en cours d'optimisation. Une progression de la valeur d'adaptation des meilleurs individus de la population est attendue. Cette valeur d'adaptation est donnée par une fonction dite de « fitness » qui note les individus. L'évolution se fait en créant des individus à partir d'individus dits « adaptés » dans l'espoir que cette descendance soit encore plus adaptée au problème à résoudre. La production de cette descendance est souvent réalisée par recombinaison de parents puis mutation de l'enfant produit par ce croisement. De par cette exploration en parallèle et donc grâce à ce principe d'algorithme à population, les algorithmes évolutionnaires sont dits « intrinsèquement parallèles ». Ils sont donc bons candidats pour être portés sur des architectures multi-cœurs. Dans cette thèse, les algorithmes évolutionnaires sont étudiés sous l'angle de la parallélisation avec en arrière plan, la conservation d'un comportement proche des standards acceptés par la communauté. Plusieurs variantes d'algorithmes évolutionnaires ont été crées depuis l'invention de cette discipline. Parmi ces variantes, nous analyserons les algorithmes classiques tels que les algorithmes génétiques et stratégies

d'évolution. La programmation génétique fait aussi l'objet d'une étude spécifique, car elle s'éloigne du modèle classique en réalisant une optimisation de la structure des solutions en même temps que de leurs paramètres.

3 Architectures multi-cœurs GPGPU

L'émergence des systèmes GPGPUs date du début des années 2000. Des programmeurs utilisent alors ces organes de calculs, initialement destinés aux calculs de rendus 3D (tels que les jeux vidéo, la DAO, etc.), pour exécuter des calculs génériques. L'utilisation dans les premières années impose la transformation de ces algorithmes vers des paradigmes de calcul graphiques, ce qui engendre des modifications algorithmiques, parfois non souhaitées. L'utilisation de ces périphériques de calculs 3D pour la réalisation de calculs plus standards, soulève l'intérêt des constructeurs de matériel qui en 2005 se lancent dans la modification de leurs architectures en vue d'unifier le calcul de rendu 3D avec le calcul générique d'une manière plus naturelle. Ainsi le constructeur NVIDIA produit la première carte à architecture unifiée et l'ensemble de logiciel destiné à son exploitation avec un paradigme de programmation parallèle plus classique CUDA. Néanmoins, de par la cible première de ces architectures, elles restent adaptées au calcul de rendu 3D. Le grand nombre de cœurs présents sur ces architectures impose un compromis sur la taille des mémoires cache associé à un cœur et une structuration SIMD/MIMD (Single Instruction Multiple Data et Multiple Instruction Multiple Data) qui implique une structuration logique similaire des processus exécutés. Le rendu 3D tel qu'utilisé dans les applications classiques des ces cartes, permet une forte structuration des accès mémoire, permettant l'exploitation d'un large bus de donnée. L'utilisation correcte de cette largeur de bus permet de très hautes vitesses de transfert entre processeur et mémoire. L'utilisation de ces architectures pour une application différente nécessite toujours une forte adaptation au matériel sous-jacent, sous peine d'une perte de performance par rapport à la puissance théorique, délivrée par ces processeurs.

4 Algorithmes évolutionnaires et GPGPUs

Néanmoins, comme nous le montrons dans cette thèse il existe des algorithmes qui s'adaptent à ces architectures, sans être directement liés au calcul de rendu 3D. Les algorithmes évolutionnaires appartiennent à cette catégorie. Leur principe de fonctionnement permet une parallélisation sur un grand nombre de cœurs, ce qui assure l'occupation d'une ou de plusieurs puces GPGPU. L'utilisation de GPGPU est analysée dans le cadre de algorithme génétique ou stratégies d'évolutions, en proposant un principe simple et intéressant de parallélisation. Ce principe permet l'utilisation de processeurs multi-cœurs pour le calcul parallèle d'algorithmes évolutionnaires sans que le comportement général de l'algorithme ne soit changé. Des résultats et des analyses de ces derniers sont présentés. Suivant ces analyses, un second modèle de parallélisation est proposé, qui permet un portage complet de l'algorithme sur la carte. Ce modèle impose cette fois des modifications algorithmiques plus profondes, par l'utilisation d'un opérateur de réduction de la population adapté au parallélisme. L'opérateur DISPAR Tournament (Disjoint Set Parallel Tournament) permet une réduction

par tirage sans remise d'une population parents et enfants ($\mu + \lambda$) vers une nouvelle population de parents (μ). DISPAR a fait l'objet d'une étude pour caractériser son comportement par rapport à l'opérateur de tournois classique avec et sans remise. Ce nouveau modèle de parallélisation d'algorithme évolutionnaire sur architecture parallèle permet un gain de temps intéressant, sur des problèmes qui ne bénéficiaient pas du premier type de parallélisation. L'applicabilité de ces méthodes est validée en utilisant des problèmes artificiels bien connus, qui permettent d'analyser les algorithmes d'une manière qualitative. Parmi ces problèmes nous utilisons l'optimisation d'une fonction de Weierstrass multidimensionnelle. Une validation expérimentale sur des problèmes non-encore résolus est faite, en collaboration avec le département de chimie de l'université de Valence, en Espagne, pour la découverte de structures de type zéolite par algorithme évolutionnaire. Cette approche permet d'obtenir l'accélération des calculs de plus de $100\times$ sur une classe de problèmes caractérisée, sans modifications comportementales de l'algorithme évolutionnaire. De plus des lignes de conduite pour l'utilisation efficace de ces architectures pour la résolution par algorithme évolution de problème ont été tracées. Une méthode d'équilibrage de charge automatique permet l'adaptation d'un algorithme à un modèle de carte quelconque. Enfin, la programmation génétique est étudiée. Nous avons mis en lumière l'intérêt de l'ordonnancement de threads dans ce contexte pour réduire les latences mémoire dont souffraient les algorithmes précédemment publiés. Ces travaux permettent un gain en temps d'exécution proche de $4\times$ par rapport aux autre travaux publiés. Cette approche est évaluée sur des problèmes artificiels, de type régression symbolique de polynôme. Mais aussi par la résolution d'un problème d'automatisme en collaboration avec Stéphane Querry doctorant au LSIIT. Dans ce dernier problème, la programmation génétique aide à trouver un modèle physique de drone. Ce modèle physique est un prérequis obligatoire à l'élaboration d'un pilotage automatique pour un véhicule autonome.

5 EASEA

Finalement, la plupart de ces travaux ont été intégrés dans une plate-forme logicielle distribuée publiquement, permettant l'exploitation de GPGPU pour l'évolution artificielle. Cette plate-forme est destinée à la recherche et à l'expérimentation par des utilisateurs non-spécialistes ni d'évolution artificielle, ni de parallélisation. Le langage EASEA préexistant à cette thèse a été modernisé, et adapté à l'utilisation de GPGPU, pour les algorithmes de type GA/ES et programmation génétique.